Automating Automation with Modular Robot Synthesis

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Mechanical Engineering

Julian S. Whitman

B.S., Mechanical Engineering, Cornell University

M.S., Mechanical Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, PA

August 2022

Acknowledgements

I would like to thank my thesis committee, Prof. Howie Choset (Chair), Dr. Matt Travers, Prof. Aaron Johnson, and Prof. Deepak Pathak. I am also grateful for the help and input from many past and present members of the CMU Biorobotics Lab. This work was funded in part by NASA Space Technology Research Fellowship NNX16AM81H "Automatic Specialization of Modular Robot Limbs," and funded in part by a Siemens Futuremaker Fellowship. This thesis is dedicated to my wife and parents.

Abstract

Robots have been used to automate many tasks, and yet designing robots remains largely a manual process. This thesis seeks to “automate automation” by developing methods to automate both design of the mechanism and control of the robot. To make these automation problems tractable, we use modular robots, as their components can be readily rearranged to form a customized robot for each new task. This work addresses two problems: control over many different modular designs, and selecting the best design for each task.

First, we present a control framework geared toward the case where a large number of designs are made from a set of modules, such that it would be prohibitively expensive to create a new control policy from scratch for each robot design. We introduce *modular policies* that learn to control a range of designs in one training process. In the policy architecture, information about the dynamics and controls is shared among modules of the same type via shared neural network parameters. We use deep reinforcement learning algorithms to train the policies on multiple designs and environments simultaneously. We then show that modular policies can transfer to new designs and environments not seen in training.

With a modular control framework in place, we apply it to select the best robot designs for specific tasks. We frame design optimization as a sequential decision making problem in which modular robots are incrementally constructed one module at a time. We then implement a deep reinforcement learning approach to train a *design value function*, an estimate of how each module added to the design will impact performance. The design value function serves as a search heuristic to efficiently identify the best design for each new task. This method is applied to two classes of robots: designing manipulators to reach locations in a workspace, and designing mobile robots to traverse a non-flat terrain.

Through these contributions, this thesis shows that *leveraging modularity in learning enables the creation and transfer of robot behaviors across tasks and designs.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

We have all dreamed of robots doing all kinds of jobs for us– Rosie the cartoon robot from “The Jetsons” cleans, repairs, walks the dog, and does many other tasks we may wish to avoid. But the reality is that a general-purpose robot that can handle a wide variety of tasks is still fiction. On the other hand, real robots can be found in modern automobile manufacturing plants performing an intricate ballet of tasks, yet these special-purpose robots can only perform a single task. We believe that modularity can break the trade-off between generality and specialization in robots, by allowing a designer to combine a small set of building blocks into specialized robots for a wide variety of tasks. In other words, a designer can use modules to create a variety of special-purpose robots.

The potential benefits of modularity are especially apparent when considering applications where the task to changes frequently. For example, in an urban disaster response scenario, the scenario that will be encountered cannot be predicted. One option to prepare for a variety of scenarios may be to transport a set of different robots to the disaster site. But, a fleet of different robots may be impractical due to the bulk, cost, and complexity involved in building, maintaining, and transporting the fleet. Instead, modularity could be employed, whereby a smaller portable “bag of modules” could act as general-purpose building blocks for customized special-purpose

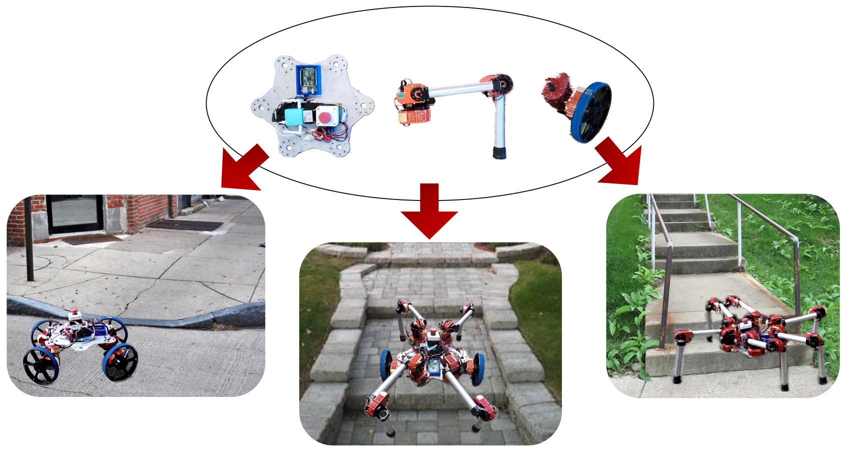


Figure 1-1: Modular robots made from body, leg, and wheel modules. A set of modular components (center, inside the black oval) can be made into different robots, each specialized for its environment (outside the black oval).

robots. If the first responders arrive to a disaster site and find they need to search a collapsed building, they could construct a modular legged robot design to climb through the rubble. If first responders instead need a robot to traverse a series of hallways and shut off a gas leak, those same modules could be reused to build a wheeled mobile manipulator. An example is depicted in Fig 1-1.

Another potential application for modularity lies in flexible manufacturing. A manufacturing facility that changes their production line frequently may benefit from specializing the manipulator designs to pick and place products, when the product and assembly line change frequently. Or, in a future factory, a broader set of modules could be re-combined to complete mobility and manipulation tasks that vary day-by-day (illustrated in Fig 1-2). A more futuristic application is extra-planetary exploration, where we could launch a set of robot modules without knowing what environment the robot will land in. The robot could predict which design will be able to explore the terrain most efficiently, then self-reassemble into that design.

In addition to the applications we envision, conceptually modularity provides a means for robots to generalize to new scenarios. Information gained (about the design, dynamics, controls, and/or performance) can be re-used, which will become especially important when constructing each new robot and/or controller from scratch is too

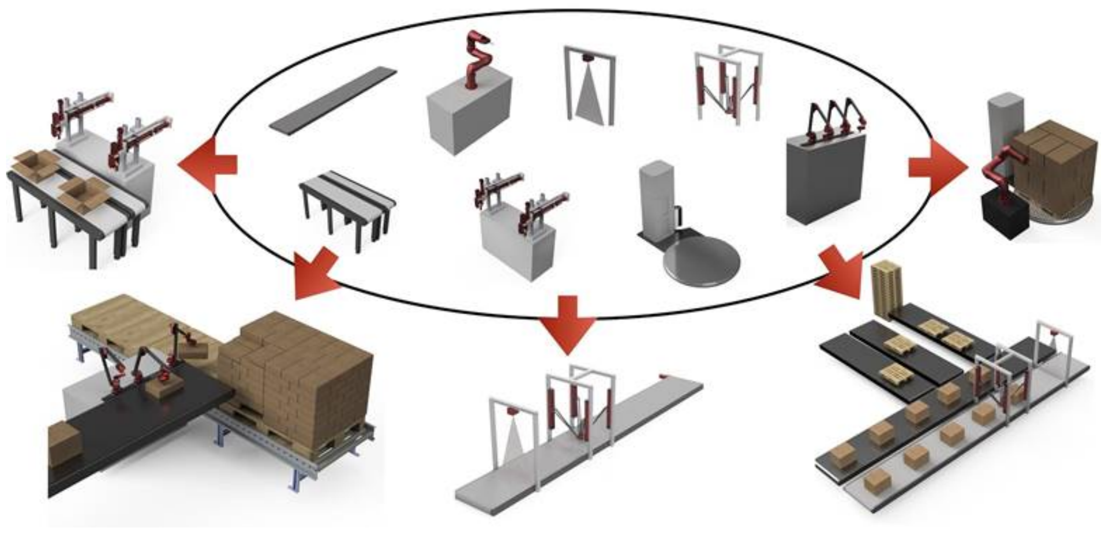


Figure 1-2: A rendering of a future modular factory, where a set of modular components (center, inside the black oval) such as manipulators, conveyor belts, mobile platforms, and sensors could be made into different manufacturing processes (outside the black oval). (Figure credit: Matt Dworman)

expensive or time consuming. One goal of this thesis is to develop methods where both the hardware and software components are modular, such that knowledge gained previously can be transferred to the development of new robot designs and controllers.

1.2 Contributions

This thesis develops new methods in "automating automation" by synthesizing modular control and design using two new concepts. First, we present a *modular policy* framework, in which one policy can learn to control a variety of designs, and readily transfer to new designs. Secondly, we present *design value functions*, a method to efficiently search for the optimal robot for a task. Through experiments and demonstrations of these contributions, this thesis shows that **leveraging modularity in learning enables the creation and transfer of robot behaviors across tasks and designs.**

During the process of selecting the best robot mechanism design for a given task, multiple candidate design prototypes may be iteratively created and evaluated. In our experience, this is true when the robot is engineered either with conventional or automation-assisted methods, and when the candidates are physically prototyped or tested in simulated. To the best of our knowledge, the best way to evaluate a design is to measure its performance at the task, which cannot be done without a controller. In other words, to test how well-matched a robot is for a task, we need to control that robot.

For some tasks and robots where control methods are computationally inexpensive and well-studied, such as quasi-static fixed-base manipulators operating in free space, controlling a series of candidate mechanism designs to evaluate each candidate's performance at the task is tractable. But for more complex tasks, such as legged locomotion, creating controllers for each candidate can be a bottleneck, as it would be computationally expensive to optimize a new policy from scratch for each candidate.

Our approach to this challenge is to first create a modular policy that can control a wide variety of designs throughout the design space. This policy in turn allows us to control the subset of designs that may be evaluated within an iterative design optimization process. By evaluating many different combinations of designs and tasks, we obtain data on their performance. From this data, our system learns how to select optimal designs for new tasks.

1.2.1 Modular policies

Given even a small set of modules, there is a combinatorial exponential explosion in the number of specialized robot designs that can be generated from that set [30, 144, 178]. Each of these many designs needs a control policy to coordinate motion among its constituent modules. For low-degree-of-freedom systems operating in open environments, such as fixed-base manipulators in open space, control may be possible in closed-form [57]. But, as the robot and environment become more complex, we are motivated to apply more expressive and complex control policies.

One possible control solution would be to learn a new control policy for each design and task. Deep reinforcement learning (RL) has been used in recent years to control a variety of robots both in simulation and in reality, producing impressive results

[28, 37, 65, 68, 71, 79, 80, 99, 100, 113, 121, 124, 141, 142, 151, 174, 176, 184]. These and related methods are improving at a rapid pace, both in reducing the computation needed and in augmenting the output policy capabilities. We draw inspiration from these works in this thesis, but despite the promise they hold, they must train a new policy for each new robot design.

The first contribution of this thesis is a *modular policy* learning framework, introduced in Chap. 3, geared toward the situation where a large number of designs are derived from a set of modules. Our approach leverages the fact that the kinematics of a modular robot can be represented as a *design graph*, with nodes as modules and edges as connections between them. Given a robot, its design graph is used to create a *policy graph* with the same structure, illustrated by Fig. 1-3). Each type of module has a neural network associated with it– i.e. there is one network used to control all of the "leg" modules, and a different one used to control all of the "wheel" modules. Each node in the policy graph uses the neural network associated with its corresponding module type's node to convert the inputs from that module's observations into the outputs for that module's actions. For example, a single neural network for leg-type modules is used to compute actions for each of the legs on a hexapod, one leg at a time.

The policy graph structure mediates how robots with different designs share knowledge, and enables modules to learn to modulate their outputs via a communication procedure in which they send and receive information over the graph edges [137, 162]. As a result, the policy can produce different behaviors for the same module depending on its location and neighboring modules within the robot. We find that graph-structured policies were able to learn to locomote more effectively than were non-graph alternative policy structures, both for designs that were seen, and those not seen, during training.

We train the modular policy with deep reinforcement learning, by collecting and learning from data from a set of different designs. We demonstrate the policy controlling a variety of designs to locomote with real robots. The modular policy generalizes to significantly more simulated designs not seen during training, without any additional learning (i.e. "zero-shot transfer"). We quantify policy generalization via experiments in which the policy is trained with 12 designs then transferred to 132 additional designs, and measure the difference between the commanded and observed locomotion heading and speed. In summary, as the first contribution of this thesis, we adapt graph neural networks as a modular policy architecture and show how modular policies can learn to control robots composed of various combinations of modules, then transfer without additional training to new robots with different designs.

Next, we develop two extensions to the modular policy learning framework. The policy developed in Chap. 3 is proprioceptive, that is, it takes internal joint encoder and body IMU measurements as its inputs, and is trained only in a flat uniform environment. In Chap. 4, the policy is given exteroceptive (vision-based) inputs, enabling it to learn to adapt its behavior to local terrain features such as stairs. We show that the policy can generalize to both new designs and environments simultaneously. Then in Chap. 5, the policy is trained in a novel reinforcement and imitation learning paradigm, where different designs can learn from each other. Our experiments show that including demonstrations from modular designs, even when those demonstrations are not from the same robot designs that are being trained, can accelerate learning. This method allow demonstrations from one set of designs can act as a sort of "prior" over behaviors for different designs. These chapters show how a modular policy can learn to control a set of designs and then transfer to new designs and environments, and how a modular policy facilitates the transfer of existing behaviors between designs.

1.2.2 Design value functions

With a framework in place for controlling modular robots, we next turn to design automation. In related work on robot design, a single task (e.g., locomoting in a given environment) is fixed, and a single design is specialized for that task [7, 19, 31, 32, 35, 45, 61–63, 81, 82, 98, 104, 107, 108, 110, 138, 163, 175]. These methods have been applied to a range of robot topologies, applications, and module types. However, these prior methods suffer from one of two limitations. Some require experts to customize

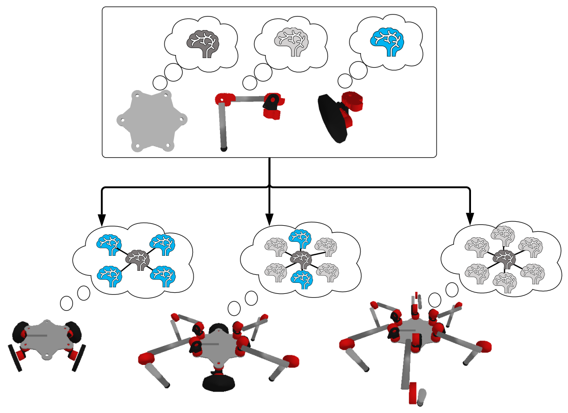


Figure 1-3: The first contribution of this thesis is a **modular policy** that can control a variety of robots. A module-level policy (depicted here as brains in the top box) is assigned to each module type (body, legs, and wheels). When the modules are assembled into robots (bottom row), the module-level policies are composed into robot-level policies (inside thought bubbles), where the policy structure matches the graph structure of the physical robot. Note the brain icons are drawn multiple times for each robot to acknowledge that there are multiple components in the policy, but in our method, all modules of one type are controlled via one neural network. The policy only has a total of *one of each grey, white, and blue brain,* and does not make copies of those three networks. For example, although we have drawn the blue brain four times in the left-most four-wheel design, each of those blue brain icons refers to the one wheel-type neural network. That wheel-type network is used to control each of the wheels not only in the four-wheel design, but also in the four-leg/two-wheel design.

the algorithm for each new task or type of robot; as a result, they cannot be easily applied to new tasks or robots. Others are computationally expensive; given that these methods must be restarted from scratch each time a new robot is optimized for a new task, this limits how quickly they can be applied to new tasks.

The second contribution of this thesis is a framework for robot design optimization. We frame design optimization as a decision-making process, where modules are selected one at a time to add to the robot design. We introduce *design value functions* characterizing the benefit of adding each module to the robot towards achieving the task. The design value function is represented by a deep neural network, which takes a partial design and the task as input, and outputs the estimated value of adding each module type to the robot. To create a task-specific robot, the design value function is applied iteratively to sequentially select modules. This iterative process is executed by a "design generator" agent, which takes the task as its input and uses the design value function as a search heuristic in optimizing the design. The design generator thereby functions as a mapping from task to design (illustrated by Fig. 1-4).

The design value function is trained using reinforcement learning, which learns the relationships among task, design, and performance offline in advance of deployment. Before the design value function can be applied, its training process must explore a large number of design/task combinations, making training computationally expensive. This training process is more computationally expensive than any individual design search using prior methods. But, by conducting offline training we gain a substantial computational benefit at run-time, when the design generator can search for the best design for multiple new tasks in an inexpensive evaluation process. This enables a user to rapidly progress from task specification, to design selection, to deployment, more efficiently than prior methods at run-time.

We apply this method to optimizing designs for two classes of robots. Chap. 6 addresses fixed-base manipulators, with manipulation tasks in cluttered environments, where the end-effector must reach a set of points in the workspace while avoiding obstacles and torque limits. Chap. 7 addresses mobile robots, with locomotion tasks in bumpy environments, where the robot must move as efficiently as possible across

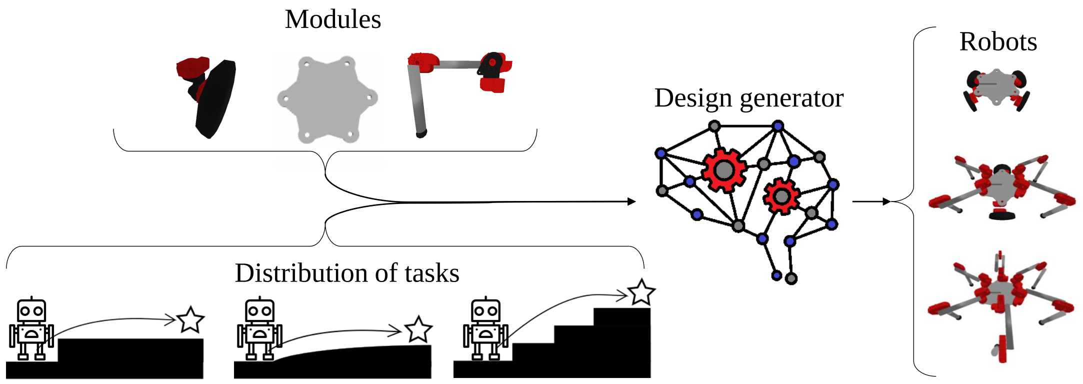


Figure 1-4: In this thesis, a robot design generator (indicated by the brain icon) uses a set of modules (top left) and a distribution of tasks (bottom left) to output task-specific robot designs (right). The tasks shown here are different terrains to traverse, and the robots generated use different combinations of legs and wheels that will most efficiently locomote on that terrain. The second contribution of this thesis is a design optimization method using **design value functions**. The design value function is trained to estimate the benefit of adding each module to the robot, and once trained, serves a search heuristic to efficiently identify the best design for each new task.

the terrain. We leverage our modular policy to evaluate the performance of different designs applied to a distribution of tasks. These chapters show how a design value function can learn the task-dependent value of modular designs in terms of the modules composing each design.

1.3 Modules and Tasks

In this thesis we need a consistent set of definitions, because on their own, words like “task” or “module” may be too vague for formal problem statements. In this section, we establish definitions of these terms so that they can then be used in later chapters.

1.3.1 Modules

Modular robots are defined as robots composed of readily interchangeable and interoperable parts [177]. The choice of what constitutes a module is itself an interesting question. One common preconception about modules considers them to be “simple” units, from which complex designs emerge. Like cells in a body, smaller units combine to form a larger system (hence, some modular robotics literature and conference sessions refer to them as “cellular” robots). However, even if modules, like biological cells, are used to form assemblies more complex than the individuals, that does not necessarily mean that each individual cell is simple. A module is not necessarily only a single-degree-of-freedom actuator, but rather a module can contain any combination of sensors, actuators, structure, power, and computation, as long as they share a set of standardized interfaces for power, communication, and mechanical connection. Modules can have a “simple” structure like rotary joints, prismatic joints, links, and brackets [33, 59, 63, 84, 101, 105, 107, 143]. Alternatively, modules may contain more complex multi-degree-of-freedom structures, like a leg or arm [36, 56, 89, 106, 172]. Modules may be all of the same *type,* i.e., homogeneous units [42, 102, 105, 106, 129, 146, 177], or they can be of different types, i.e., heterogeneous units [63, 84, 107, 143, 148, 167].

We use the terms *arrangement* and *designs* interchangeably in this thesis to refer to a combination of connected modules in a robot. These terms distinguish the design from the *configuration*, which is sometimes used in modular robotics literature to refer to a composition of modules. A configuration in broader robotics literature more often refers to an element of the *configuration space*, which is a vector of joint angles that can be used to determine the location of every point on the robot.

Similarly, the word “modular” is used broadly and loosely throughout robotics and computer science literature to refer to anything with an interchangeable hardware or software components. For example, Devin et al. [46] states that their “policies can be decomposed into task-specific and robot-specific modules;” their modules correspond to different parts of the policy, but not to any division of hardware components within an individual robot. In our usage of the word “modular,” we will refer to cases where the hardware is composed of interchangeable and interoperable components, and refer to modular policies as those divided up into the same number and arrangement of components as is the physical hardware.

Advantages and disadvantages of a modular system

Based on our literature reviews and experience, we see the following advantages, disadvantages, and challenges of using modular robots as compared to monolithic (e.g. non-modular) robots. Some advantages are:

- **Rapid design.** Modules can be quickly assembled into specialized designs for a variety of tasks.
- **Rapid re-design.** The ability to easily and quickly change the design if the task changes.
- **Individual design complexity.** Any individual modular design can be less complex than a monolithic design would need to be, if that monolithic design needed to perform many different tasks, but the modular design could be rearranged to perform each different task.
- **Re-use during engineering.** If multiple sections of the robot are the same (e.g. the left leg and right leg can be identical), the engineering time can be reduced relative to the time needed to design each part separately. The number of different parts to be made or purchased can be reduced, and purchasing parts in batches can reduce the total parts budget. For example, the hexapod used througout this thesis has six identical legs, such that all legs can be constructed and repaired using a single pool of common components.
- **Re-use for future projects.** Modules developed for one project can be reused for future projects, which can reduce time and money spent developing new robots in the future.
- **Maintenance.** Modules can be replaced if they break, reducing maintenance effort for an individual design.

Modularity in robotics comes with disadvantages as well:

- **Complexity.** The interfaces between modules add mass, complexity, cost, and potential failure points to the system. In our experience, the interfaces are the part of the module that break most often and require the most engineering effort to design. The interfaces also need a set of communication and power transmission standards applicable to any combination of modules.

- **Efficiency.** Any individual design may be less capable or efficient than a monolithic design created specifically for that task. For example, a mobile robot made from modules will be heavier than a comparable-size monolithic robot, and therefore will expend more energy locomoting the same distance using the same control policy. In our experience, this sometimes means that the modular robot's capabilities may not match the expectations one may have after witnessing the capabilities of comparably-sized monolithic robots.

- **Physical limits.** The use of modules restricts the design space. A modular robot sometimes has physical limits are more constrained than those of a comparable monolithic design, due to the constraints on the design space imposed by the size and mass of the modules. For example, consider using modules to make a fixed-base shoulder-elbow-wrist manipulator. If the module set contains only single-axis rotary joints, then one cannot exactly replicate a spherical joint, as would be desired for the wrist. To approximate a wrist, we could connect a chain of three joint modules with the smallest possible distance between them,. However, that wrist-like assembly might be too heavy for the base modules to support when extended out to reach positions far from the base.

Additionally, we see a number of challenges introduced by modularity. These are not necessarily fundamental drawbacks, but are problems to be addressed:

- **Control creation.** There is an additional engineering burden in creating controllers for multiple designs, and deciding how centralized/decentralized those controllers should be. We address this challenge in Chapters 3-5.

- **Design selection.** It may be difficult to decide which design to use for each task, requiring either robotics experience or design optimization computation. We address this challenge in Chapters 6 and 7.

- **Systems engineering.** It is more difficult to engineer, initially, a set of modules than it is to design a single monolithic robot. This is particularly challenging if there are demanding design requirements like waterproofing, extreme low-mass or low-size, or high force output.
- **Defining the module set.** In the module engineering process, it is not obvious what the contents of each module should be, and how many modules to make. These choices have ramifications when those modules are used later to make designs; for instance, in the "Physical limits" limitation above, it may be possible to engineer a spherical joint to use as a wrist, which would make additional manipulator designs feasible. But, when the modules are developed far in advance of deployment, it may not be obvious whether it is worth the additional engineering cost to develop and produce a spherical joint module.

1.3.2 Tasks

The word "task," like "modular," is used frequently throughout robotics literature with different definitions made by each author. In this thesis, we will define a *task* to be an environment and an objective to achieve. For the sake of semantic simplicity, if either the environment or the objective is changed at all, we will call that a new task, even if the other (environment or objective) have not changed. In this thesis, we primarily change the environment and/or desired goal locations, and keep the high-level objective (e.g. manipulation vs. locomotion) fixed, although this need not be the case for our methods to apply.

A vague task description may be "locomotion over rough terrain." Unfortunately, a description like this is insufficient to create a precise numerical representation of the environment and objective. In this thesis, we require tasks to be parameterized so that the relevant objective functions, constraints, and simulation environment can be created. This type of conversion of high-level task description into a precise representation for the robot is itself an open area of research [11, 154]. We set the scope of our work to exclude semantic task parsing, and instead assume we can manually parameterize the tasks.

For example, manipulation tasks $T_{\text{manipulation}}$ could be parameterized by their workspace, start, and goal. The workspace could be represented by a voxelized occupancy grid of size $N \times N \times N$, and an initial and final end effector pose to be reached, such that

$$T_{\text{manipulation}} \in \underbrace{\{0,1\}^{N\times N\times N}}_{\substack{\text{Workspace obstacle}\\\text{occupancy}}} \times \underbrace{\mathrm{SE}(3)}_{\substack{\text{End-effector desired}\\\text{initial pose}}} \times \underbrace{\mathrm{SE}(3)}_{\substack{\text{End-effector desired}\\\text{final pose}}} . \tag{1.1}$$

A set of positions and obstacles is an instance of this space of tasks.

Similarly, ground locomotion tasks $T_{\text{locomotion}}$ could be parameterized by a terrain height map size $M \times M$ with an initial and final goal waypoints,

$$T_{\text{locomotion}} \in \underbrace{\mathbb{R}^{M\times M}}_{\substack{\text{Terrain elevation}\\\text{map}}} \times \underbrace{\mathrm{SE}(2)}_{\substack{\text{Robot initial}\\\text{location}}} \times \underbrace{\mathrm{SE}(2)}_{\substack{\text{Robot desired}\\\text{final location}}} \tag{1.2}$$

A waypoint and a discretized terrain height map is an instance of this space of tasks.

Given these definitions for the space tasks, we can assign bounds and likelihoods to different instances of the objectives and environments. We introduce additional parameters like the max, min, mean, and variance of obstacle sizes, or bounds on the waypoint locations, to make up the *task distribution.* For example, in mobility tasks, the terrain may not be arbitrarily rough. We must limit the task distribution to environments we believe the robot is likely to encounter, for instance, the terrain could be parameterized by a bumps with uniformly distributed length, width, and height. We can train the design generator to infer the optimal robot design for tasks sampled from that distribution.

Chapter 2

Background

This thesis draws inspiration from a variety work reviewed in this chapter. We begin by describing the different forms of modular robots in Sec. 2.1. Then, Sec. 2.2 outlines relevant literature on model-based control and deep reinforcement learning. Sec. 2.3 summarizes methods for design automation, including both long-standing methods like evolutionary algorithm, branch-and-bound methods, and recent methods using machine learning.

2.1 Modular Robots

2.1.1 Modular robots in academic research

The design of modular robot mechanisms has been an active area of academic research since at least the 1990's [30, 39, 93, 140, 178], and a wide variety of different modular robot concepts have been developed since then [144]. Modular robots can be classified by whether the robot can alter their collective design autonomously without external intervention. Robots with this ability are known as *self-reconfigurable* robots [179, 180]. Self-reconfigurable modular robots offer the promised ability to change their design to complete multiple tasks without human intervention [42], and redundancy to automatically leave behind broken modules. Self-reconfiguration has been a long-standing open problem in modular robotics [179] as well as the subject of recent

(a)

(b)

(c)

(d)

(e)

(f)

Figure 2-1: Examples of recent modular robots. The modules sets in the left column are simpler, in that they contain at most one actuator. In the middle column has more complex modules, and in the right column, modules can be an entire interchangeable limb or tool. (a) S-series modules and (b) "Eigenbot" modules from the Biorobotics Laboratory, (c) Disney Research's Snapbot [56], (d) Hebi robotics [70], (e) University of Pennsylvania's SMORES [106], and Universal Robotics' quick-change end-effectors [157]

research [102, 106, 129, 146, 148, 152, 172]. One challenge in self-reconfiguration is that the modules need additional hardware features, such as batteries on each module and actuated interface mechanisms [48]. Another challenge for self-reconfigurable robots is *reconfiguration planning*, that is, how to control the modules to alter the overall design of the assembly [24, 55]. In contrast, *manually reconfigurable* modules require a user or external agent to alter the design [180]. While this means that the design is fixed during deployment, it allows for more robust interfaces, and fewer required electromechanical features per module. In the above self-reconfiguration prior works, identification of the initial and final designs are provided by a user.

2.1.2 Commercially available modular robots

Although modular robots have been of interest to industry since at least the 1970's [92], few modular robots have made the transition from the laboratory to the real world. That is not to say that no modularity has historically existed in robotics, because any interchangeable components and standardized interfaces can be considered

Table 2.1: Recent commercially available modular robot systems.

Company	Product	Year	Type of robot	URL
Schunk	Powerball	2012	Manipulators	`schunk.com`
Hebi	-	2016	Arms, legs and wheels	`hebi.us`
Modbot	-	2020	Manipulators	`modbot.com`
Robco	-	2020	Manipulators	`robco.de/en`
Modrobotics	-	2020	Educational toys	`modrobotics.com`
Keyirobot	Clicbot	2020	Educational toys	`keyirobot.com`
Beckhoff	ATRO	2022	Manipulators	[16]

to be a form of modularity. In particular, interchangeable end-effectors have become common in commercially available manipulators.

We have compiled a small table of some commercially available robots that have modularity in their overall design, and not just their end-effectors, in Table 2.1. This is by no means exhaustive, as new start-ups and products are being created frequently, and companies may go out of business or discontinue a product line.

2.2 Robot Control

A *robot control policy* is a function that takes as input the robot's observation of its state, and outputs actions for the robot's actuator. We will use the shorter term "policy" to refer to a robot control policy.. A policy can take many forms: it could be a hand-coded look-up table, or an algorithm using dynamic trajectory optimization, or it could be a neural network.

Our modular policy architecture and training algorithm build on a range of literature on model-based control, deep reinforcement learning, and methods that learn a single policy for multiple robot designs. Because the control methods we develop in this thesis are applied to ground robot locomotion, we focus our attention on related work using such systems.

2.2.1 Gait libraries

Previous work controlling modular locomoting robots used a table or library of policies that are either hand-crafted [36, 42, 84, 181] or optimized individually for each robot design [64, 68, 167]. While these methods have been successful for individual designs, conventional policy creation methods, where highly-trained experts carefully hand-tune the policy over long periods of time for individual robots, become expensive when the robot takes on a new design every day.

2.2.2 Model-predictive control

Model-predictive control (MPC) methods combine hand-engineered dynamics models with trajectory optimization. They are used widely in engineering processes, both within and outside robotics. In MPC, a trajectory is optimized for a finite horizon, a short sequence of the output actions are applied, then the trajectory is re-optimized, typically warm-starting using the unused portion of the previously output trajectory as an initial seed [125].

Recently, promising results in legged and leg-wheel robot locomotion have been made through MPC [20, 21, 53, 54, 171]. For example, they have enabled legged and leg-wheel robots to locomote through three-dimensional terrain while applying simultaneous localization and mapping (SLAM) to build up a map of the environment included in model predictions [20]. Though some impressive results have been achieved, these methods suffer from a few limitations.

One drawback of these methods is that they often rely on assumptions specific to an individual robot design, such as assuming the robot has centroidal dynamics, massless limbs, with a rectangular prism workspace or user-specified foot contact sequences [20, 21]. When working with robots with numerous combinations of legs and wheels, not all designs will have properties compatible with these assumptions– each design may have a different number of limbs, different types of limbs, and a different distribution of mass. Similarly, when adapting the controller to a new environment, even when holding the design constant, a different foot/wheel contact sequence may be optimal in different environments. These assumptions makes it difficult to quickly adapt the controller to the many designs that can be made from modules, and to new environments.

Another drawback of these MPC approaches is, in general, that they require efficient trajectory optimization subroutines, in order to react quickly to disturbances or unmodelled interactions with the environment [21]. This requirement for efficiency is further complicated by the curse of dimensionality, i.e., the tendency for the computational complexity of the optimization algorithms used in MPC to scale superlinearly with the size of the state space [86, 97]. As a result, these methods rely on large on-board computers to manage solution time, so that the policy can replan as fast as possible. Modular robots, in particular those with distributed computation, in our experience cannot always support a large computer.

The solution time can also be reduced via code optimization, and via pre-computed and engineered heuristics can be used [21, 22]. But in our experience, the subroutines in custom trajectory optimization are often limited to specific robot designs, and the heuristics customized to an individual design and contact sequence As a result, these methods do not provide a scalable solution to the wide range of potential designs and environments we encounter with modular robots.

2.2.3 Reinforcement learning

Reinforcement learning (RL) is a class of methods that train a policy using data collected iteratively from an agent's interaction with its environment [80]. RL has been used in recent years to control a variety of robots, included highly-articulated locomoting robots, both in simulation and in reality [9, 68, 71, 79, 80, 121, 174, 183]. These methods produce a "global reactive" policy, which takes as input an observation from any part of the state space and outputs an action that reacts immediately to the observation. The RL algorithm operates as an optimizer solving for policy parameters that maximize the robot's rewards (or equivalently, minimize a cost/objective function) when that policy is applied to the environment over a time horizon.

In recent years, these policies are often represented with deep neural networks.

These networks often contain millions of parameters, causing the networks to be computationally expensive to train. However, once trained, a deep neural network policy can be applied computationally efficiently due to the fact that the "forward pass" from inputs to outputs consists only of a short series of matrix multiplications and additions, in contrast to the comparatively expensive trajectory optimization inner loop processes in conventional MPC. The computationally inexpensive run-time of neural network policies makes them attractive for use on systems with limited on-board compute, such as modular robots. RL algorithms can be divided into two categories: model-free reinforcement learning (MFRL), and model-based reinforcement learning (MBRL).

Model-free reinforcement learning

In MFRL, trajectory data, i.e., sets of states, actions, and rewards, from either a physics simulation or a physical robot are used to reinforce policy action selections that result in high reward, while treating the robot and environment together as a black box [80]. That is, a policy is learned directly from an agent's interaction with its environment, collecting and learning from trajectory data without explicitly modelling the system dynamics. There are a variety of different methods within MFRL [150]. "On-policy" methods such Proximal Policy Optimization [142] train an "actor" network as the policy and a "critic" network. The critic is trained to estimate the *state value function*, which represents the maximum possible return (sum of rewards) that can be achieved when applying a policy from a given start state. "Off-policy" methods such as deep Q-learning [112] learn an estimate of the *state-action value function*, which represents the maximum possible return that can be achieved when applying a policy from a given start state with a given action. The state-action value function can then be used as a policy by selecting the greedily selecting the action with the highest estimated return at each time step. Note that RL can be applied to any sequential decision-making process, not only to robot control. We will apply Q-learning to our robot design synthesis in Chapter 6 and 7, and describe the Q-learning algorithm in greater detail.

MFRL has been used recently to create compelling demonstrations of many varieties of locomoting robots both in laboratory settings and deployed in the real world [1, 65, 68, 71, 79, 94, 111, 121, 151, 174]. The primary drawback of these approaches is that they often suffer from high sample complexity, (equivalently low sample efficiency), that is, the amount of data needed, where data consists of trajectories gathered from the policy interacting with the environment. Collecting data samples is typically the computational bottleneck in MFRL, and the sample complexity tends to increase with the size of the policy network, as well as with the number of degrees-of-freedom of the robot. Consequently, to direct a robot to locomote in multiple directions, often a separate policy is trained for each desired robot heading [68, 121, 174]. The sample complexity also tends to scale with the dimensionality of the observation, leading many of the above works to learn proprioceptive policies that use only robot internal state information as observations. However, recent developments in parallelized simulation have shown promise in mitigating MFRL's sample complexity drawback [109].

Recent work has also shown that MFRL can learn visual-motor policies, i.e., those that take exteroceptive (externally-measured) inputs in addition to proprioceptive inputs. The exteroceptive measurements usually take the form of a local terrain height map computed in conjunction with a SLAM system. For example, Yu et al. [182] trained a vision policy that outputs foot placements, then uses model-based contact-scheduled control. Miki et al. [111] trained a policy that takes in both local terrain map and proprioceptive, trained with a multi-stage process including both MFRL and imitation learning. Rudin et al. [132] showed that a stair-climbing quadruped policy using visual inputs could be trained rapidly with GPU-enabled simulations.

In addition to learning behaviors from scratch, MFRL has also been combined with imitation learning (IL). In conventional imitation learning, a dataset of demonstrations is used to train a policy without additional data gathering procedures; for example, the most long-standing IL method known as "behavioral cloning" trains a policy with supervised regression [122]. Hester et al. [73] combined RL and IL by training a value function approximation network from state transitions in a dataset.

Ding et al. [47] used expert demonstrations within a variant of hindsight experience replay [10]. Peng et al. [121] constrained the policy to be near the states in the dataset. Although these methods allows for small differences in the design of the system in the demonstration and the robot design, they require the design in the demonstrations have the same topology as the design being trained.

Model-based reinforcement learning

Where MFRL treats the robot-environment interactions as a black box, MBRL differs by explicitly learning a *model* of the environment dynamics [8, 9, 37, 67, 96, 96, 113, 124, 124, 176, 183]. This approximate dynamics model can be used efficiently within policy optimization, because the model can "imagine" how the robot would respond to a large number of possible control inputs without having to execute them with the simulation or on a real robot. MBRL has been repeatedly shown to be more sample efficient than MFRL [8, 37, 113, 124], that is, uses fewer trajectory data samples to gain equivalent proficiency. This means that MBRL is often more computationally efficient than MFRL, as gathering data is usually the main computational bottleneck in RL control algorithms.

The model learned in MBRL can be used in place of a hand-engineered dynamics model within MPC. The same model can be used to optimize control sequences for multiple tasks, such as directing the robot to locomote at various headings and speeds [113, 176]. But running MPC in real-time creates similar limitations as described in Sec. 2.2.2. As an alternative, one can train a reactive policy neural network, which can compute control actions is comparatively inexpensive at run-time, even for high degree-of-freedom systems. Guided Policy Search (GPS) [28, 99, 100, 184] is a form of MBRL that can produce such a reactive controller. GPS iteratively re-fits dynamics models, uses local trajectory optimization off-line to find a series of local trajectories, then merges those trajectories via imitation learning into a global reactive control policy.

One limitation of MBRL methods is that the policy relies on the accuracy of the learned dynamics model. If the dynamics model is not precise enough, the policy can exploit the model, a phenomena known as model bias [38]. To work well, the model must be able to gather data from the “right” parts of the state space that would be used by an optimal policy, which can be difficult in high-dimensional or complex/unstructured environments. MFRL does not have this problem because in MFRL, the agent is always interacting directly with the environment, and does not need to model the dynamics. However, “incremental” MBRL methods such as GPS mitigate this problem by iteratively re-fitting the model as new data is gathered [124], such that some recent MBRL methods have reached the performance of MFRL [9].

Most existing MBRL work operates in uniform environments without exteroceptive (e.g. visual/depth map) inputs. However some recent work [160, 161] adds a local terrain measurement to the model input, and operates on rough terrain. These works have to date been limited to low degree-of-freedom (DoF) car-like robots, and use the approximate model for MPC rather than learning a reactive policy. Given that computation time to compute actions with MPC scales exponentially with the number of degrees-of-freedom, this would prove difficult to implement on an onboard computer for more articulated robots. A summary of recent MBRL algorithms can be found in [96].

2.2.4 Learning decentralized and multi-design control

RL can not only apply a single algorithm separately to many designs– in some cases it has been used to train one policy that can transfer to multiple designs. This is accomplished by applying one set of neural network parameters to a range of designs, and training the network using data gathered from those designs. One method to do so is to condition the policy network input on the robot design parameters encoded as a vector [34, 108, 138]. Then, the policy can be applied directly to, or fine-tuned to transfer to, a design with a design not seen during the initial training period. However, this has only been shown previously where the set of designs all share the same topology.

When robots contain repeated structures, but not necessarily the same overall topology, another method to transfer a policy among multiple designs is to train a decentralized policy. The policy is applied to each repeated part of the robot, e.g. the joints or limbs reused multiple times within the designs. By assigning a policy component to each part of the robot, when that part of the robot is removed or an additional part added, the policy can transfer to the altered design. For example, Sartoretti et al. [136] trained a decentralized policy for the legs of a hexapod. This demonstrated that sharing policy information among the legs enabled accelerated training compared to a centralized policy for the full robot, and showed the policy could transfer to robot designs with fewer legs. However, each leg acted independently, without any internal coordination between limbs, ultimately limiting its capabilities.

In order to allow internal coordination between decentralized policy components, a more complex function than a conventional deep neural networks is needed. *Graph neural networks* (GNNs) are one such class of functions. GNNs are a form of neural network that operate over graphs. They can encode graphical structures into neural networks and share learned knowledge among repeated components [173]. Unlike more conventional neural networks, which have a fixed input and output dimension, GNNs allow a single set of neural network parameters to process inputs and produce outputs with variable dimensions. For robotics applications, this means that a single GNN can be applied to a range of robots with different numbers of sensors and actuators, as long as those components can be represented as nodes in a graph. [162] introduced "NerveNet," which used GNNs as a control policy. Each joint on the robot formed a node in the policy graph. NerveNet was able to generalize to some simulated designs not seen in training, such as from a centipede-like simulated robot to another centipede-like model with a different number of body and leg segments. Similarly, [78, 119] trained GNNs where each node controlled an individual robot joint, showing that GNNs can transfer to control systems with different topologies than were seen during training. However, these methods were only applied to simplified simulated robot models, and used a computationally expensive MFRL training algorithm.

GNNs have also been used to represent an approximation of the forward dynamics model, such that a single model can be applied to multiple designs [14, 134]. But, the GNNs and training procedures used by these works [14, 78, 119, 134, 162] encoded only the connectivity between joints into the graph, without the recognition that groups of joints repeated within the robot (e.g. limbs) or rigid components without joints (e.g. bodies or links) can be reused across different designs. Further, in order to make the model agnostic to the design, these works use maximal-coordinate state representations, for instance by including the full world-frame positions and velocities of each link in the robot to their state, which would make transfer to reality difficult in the face of uncertain state estimation.

2.3 Mechanism design synthesis

When using a set of robotic modules, a clear question facing the user is what design would be best for each task for a robot to complete. Expert users may be able to decide the best design either through intuition and experience, or if time allows, through trial and error informed by extensive simulations. The simplest solution, is to "enumerate and evaluate" by exhaustively simulating all possible designs. While this is feasible in relatively small search spaces, the space of modular designs grows exponentially with the number of modules, so such exhaustive approaches are not scalable. A number of approaches have been developed to search for designs, which we categorize into either evolutionary algorithms, branch-and-bound searches, and learning-based methods.

2.3.1 Evolutionary algorithms

The most popular class of methods to automate robot design are evolutionary algorithms (EA). A small selection of the plethora of such works include [19, 31, 32, 51, 82, 104, 110, 120, 163, 175]. EA search the design space by starting with a population of designs, randomly varying individuals in the population, over time selecting for those that perform best at the task. Each candidate design will need a controller synthesized for it, which can be created anew at each evaluation step [82] or developed incrementally alongside the population [163].

An EA typically proceeds as follows. EAs are initialized with a population of randomly selected designs. Each design is evaluated according to an objective function (in this literature, usually referred to as a “fitness” but also equivalent to an “episode return” or “net cost” in other literature). The designs with the highest fitness are propagated to the next iteration, and random changes are applied to the remainder of the population. These biologically-inspired operations usually include “cross-overs” in which components are traded between sets of designs, and ”mutations” in which small random changes are applied to individual designs. The new population is then evaluated, the highest-fitness designs are replicated, and the lowest-fitness designs are removed. The process is repeated until the population converges to a steady-state containing high-performing designs.

EAs have been successful in the field due to a number of inherent advantages. Our own observation has been that the method is, compared to other applicable methods, relatively simple to implement. This is still the case when the search space is complex, high-dimensional, mixed discrete and continuous, and even when the design and control are to be evolved simultaneously. To set up an EA, one must first craft an encoding that allows any design to be converted to a fixed-length “gene” vector. This vector should compatible with some mutation and cross-over operations; if a single value in the vector is altered (mutation), it should still correspond to a valid design, and if a section of the vector for one individual is exchanged with the same section of a vector from a different individual (cross-over), both vectors altered should correspond to a valid design. Next, one must craft a task-specific fitness function. An EA will then require a simulation environment able to convert a design variables into a robot model, and evaluate the objective function on that robot.

Another advantage of evolutionary methods is their ability to be parallelized. Evaluating of the designs in the population, by applying their policy to the task, can be conducted in parallel threads across many CPU cores. This allows large populations to be distributed potentially across multiple computers as well, although this does not change the total amount of compute used.

EAs are not without their drawbacks. First, the simulation environment is effectively treated as a black box– information about the relationship between design and performance is propagated between iterations only via the parameters of the designs in the population. Second, the results can vary substantially between runs, due in part to sensitivity to the designs drawn in the initial population. For example, if all of the initial designs in the population are, by chance, very low fitness, then the population may take many iterations to develop a high-fitness design. On the other hand, if the initial population happens to find a single high-fitness design by chance in an early iteration, that design can quickly reproduce and dominate the population, potentially excluding other designs with even higher fitness. Lastly, EAs can quickly become computationally expensive because many candidate robots must be simulated at each iteration.

Within the robot design problem, the optimal design depends on both the task and the control policy. In some cases, the control policy exists *a priori*, for instance, using conventional path planners and inverse dynamics control for fixed-based manipulators. When the control does not already exist in advance, one solution is to develop a new policy individually for each design. However, this can become computationally burdensome as the number of designs increases with the number of modules. Some evolutionary approaches (e.g. [19, 104, 110, 120]) address this by evolving policy parameters along with the design. The drawback of such an approach is that it results in a much larger, more complex, search space, exacerbating the aforementioned limitations of these algorithms. An additional side-effect of EA with both design and control is the tendency for designs that are easier to control to develop faster [61, 74]. This effect is found in biological evolution, where it is called the Baldwin effect. In other words, there is a larger evolutionary hurdle (an attractive poor local minima) to advance to a stage in an evolutionary process in which a complex control operates on a complex design. To mitigate this effect, modifications to the EA algorithms were made in [35] that allow for multiple control evolution steps at each design evolution step.

2.3.2 Branch-and-bound methods

An alternate deterministic method to improve upon an “enumerate and evalute” search method is to prune and prioritize regions of the search space using heuristics. In other words, iteratively identifying a section of the design space, and creating a bound on the objective function in that region. [7] performed a series of checks which progressively eliminate candidate designs based on simple criteria such as total length or static torques. This method requires the evaluation criteria be manually specified for each task and module set, and could become computationally expensive *en masse* given an exponentially large search space.

Another way to search the design space is by viewing the space of designs as a graph and applying graph search algorithms. The most prevalent way to do so, which is adopted in our work, is to view the space of modular designs as a tree. In this tree, each node is an arrangement of modules. The root node is an empty arrangement, and each child represents adding a module to the arrangement. Each attachment must be compatible with the physical hardware attachment points. Each step away from the root of the tree represents a module sequentially added to the robot, until either the maximum number of allowable modules is reached or no more modules can be added due to a lack of open attachment points. These leaf nodes are considered “complete” designs and can be evaluated in simulation.

In order to manage the search complexity on this tree, its branches can be pruned via a best-first tree search [44, 45, 63]. In these works, a heuristic was hand-crafted that estimated the ability of each partially complete arrangement in fulfilling the task, and was used to guide a search over a tree of different designs. These heuristics were carefully constructed by hand, and were meant to represent a reasonable estimate of the best-case potential performance of any designs stemming from that partially complete design. While a best-first search is guaranteed to be complete, the search heuristic was not shown to be admissible, i.e., it may over-estimate the cost-to-go and incorrectly de-prioritize the branch on which the true optimum lies. The evaluation of these heuristics involve solving an optimization subproblem, which becomes computationally burdensome as the number of possible module types and connections grows. Further, the heuristics do not consider obstacles, self-collisions, or torque constraints. These methods also assume the existence of a planning and/or control algorithm used to evaluate each candidate design.

2.3.3 Learning-based methods

Another variety of design optimization methods draw on machine learning, most often reinforcement learning (RL). Wang et al. [163] combined an evolutionary algorithm to optimize robot design with reinforcement learning to simultaneously optimize a control policy. Schaff et al. [138] 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. [62] 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 task, and each search conducted is computationally expensive. As a result, if the task is altered, these algorithms must be restarted, making them costly to use as a design space exploration tool in rapid-prototyping. These methods create a single policy and design for a single environment, but do not retain information about how various designs performed, which could be of use in future design automation problems for new tasks.

In addition to these learning-based methods for robot design, we find similar methods applied to non-robotics domains. Design synthesis problems in the chemical engineering literature bear some similarities to our modular design problem. Machine learning has been used to design novel molecules, selecting component atoms and bonds from a discrete set [43, 58, 135, 189]. In particular, Zhou et al. [189] used a deep reinforcement learning paradigm for molecule discovery by sequentially adding atoms to a molecule. This allows for an expressive space of designs (molecules) created while keeping the decision space limited to the number of discrete components (atoms and bonds) in the design. In a similar method, RL has also been used to design a deep neural network for image recognition [12], where the layer sizes and types are treated as sequential decisions to be made in neural network architecture.

The methods we have listed so far use evolution or RL as the optimizer for a single task; that is, they fix the task and environment then search for a design. Thus they suffer from the time it takes to optimize each design, and do not retain information about the relationships among task, design, control, and performance that are observed over the course of the optimization process. This limits their potential for use within robot reconfiguration and redeployment, when rapidly prototyping designs and when the task changes frequently.

Chapter 3

Learning modular robot control policies

Even with a small set of modules, there is a large number of different robot designs that can be generated from that set [30, 144, 178], and each design needs its own control policy. Further, there is an added nuance to this scaling: how the modules interact with each other also plays a role in optimizing policies. Each module needs to behave differently depending on its *context*, i.e., its functional role in the system. For instance, the legs in a hexapod must behave differently than those same legs need to behave in a quadruped. And, within any one robot, the location of the module impacts its desired behavior, e.g. a leg should function differently when used as a front or rear limb. Dynamic interactions among the different portions of a robot (e.g. coupling between legs) are important within the policy as well.

Given that context matters, with an eye towards mitigating computational complexity, we aim to create policies for the modules that can automatically determine their function within their given context. Just as one can install a new keyboard into a computer without reinstalling the operating system, we seek for the module's policies to have a similar "plug and play" feature. However, we seek a "plug and *adaptive* play" because in robots, a module's behavior must adapt to its context within the system.

Instead of optimizing separate policies for each individual robot design, this chap-

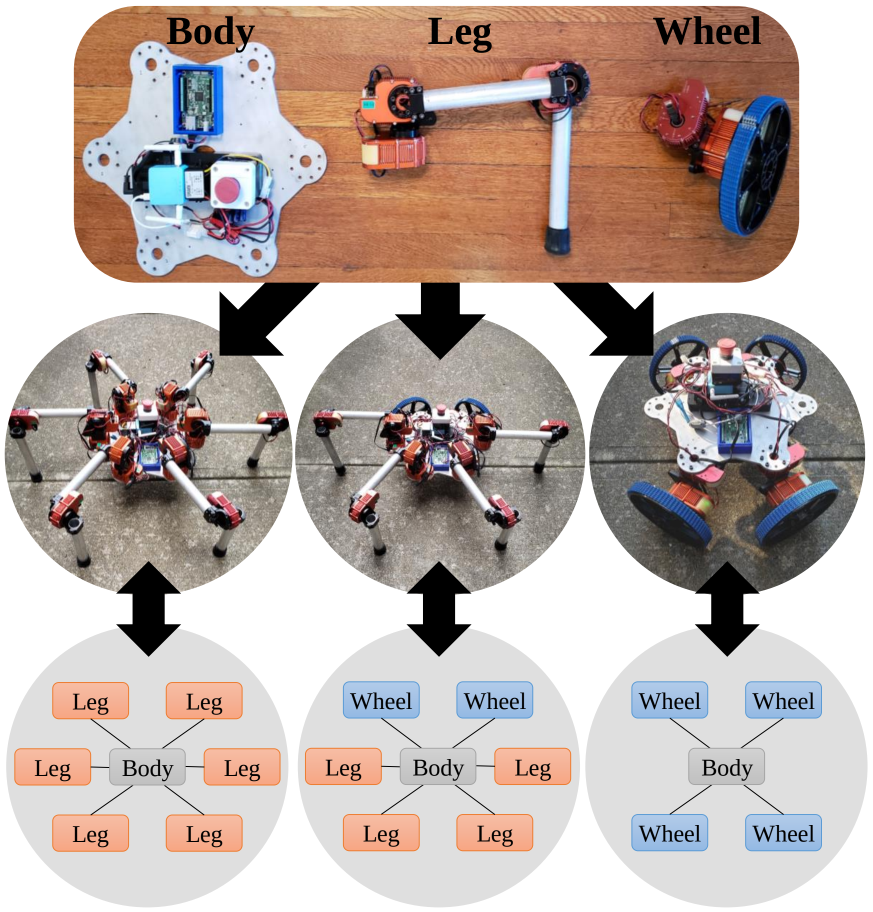


Figure 3-1: A set of modular components, a body, legs, and wheels, (top) can be combined to form many robot designs (middle). These designs can be represented by graphs (bottom). Our modular policy learning algorithm leverages the graph structure common to all such modular designs, enabling us to control a variety of robots composed of these modules.

ter develops a learning process that trains policies for the modules. The modules' policies learn to adapt to different contexts, such that they can be connected together into system-wide policies for various designs. In our architecture, which we call *modular policies*, a global policy (comprised of the union of the module policies) consists of a set of deep neural networks, where there is one network for each type of module. For instance, our hexapod has six leg-type modules, and each leg uses the same neural network to process its observations and produce actions. A leg-wheel hybrid design uses that same leg-type neural network for all of its legs, but applies a different wheel-type neural network to its wheels. Once trained, the policy can readily transfer to many different robots made from the same set of modules.

Our approach leverages the fact that kinematic structure of any modular design can be represented as a graph, where modules are nodes and electromechanical connections between them are edges. Fig. 3-1 depicts the relationship between modular designs and graphical structures which we adopt in this thesis. When a robot design graph is input, the module policies combine to act as a reactive feedback control policy for that robot, where the module policies are connected with a graph structure corresponding to that of the hardware. Making the policy graph structure the same as the hardware graph structure changes how robots with different designs share neural network parameters when compared to prior hardware-conditioned policies. We find our graph-structured modular policies learn to match a desired heading and speed more effectively than non-graph policy counterparts such as the "hardware-conditioned" policies of [34].

The policy graph structure also enables modules to adjust their behavior to their context. Module policies do not act independently, but rather learn to automatically adapt their outputs via a communication procedure [137] in which they send and receive information over the graph edges. Some prior work has applied graph structure to robot learning, treating each joint in the robot as a node to learn dynamics [134] or policies [119, 162]. However, these approaches were not transferable to distinct robot designs from those seen in training.

We are able to achieve some notion of scalability, and test on many additional designs, because each type of module, regardless of which robot it is situated in, shares the same dynamics model and policy neural network parameters. Each module type (e.g. a leg), uses the same parameters in all positions within a single robot, and over all robots that are generated; this prevents them from over-fitting to a single positions and design, forcing them to learn to communicate with other modules in the system and adapt to their context.

To train the policy to handle a variety of robot designs, we use reinforcement learning. The algorithm introduced in this chapter is inspired by the ideas of Guided Policy Search [28, 99, 100, 184], which iterates between learning an approximate model of the system dynamics, predicting optimal trajectories under the learned model, and distilling those local trajectories into a global policy. We achieve policy generalization to new robots: we show zero-shot transfer (application without additional learning or optimization) to an order of magnitude more designs than were in the training set. Then, we demonstrate our modular policy applied to designs with different combinations of legs and wheels locomoting on real robots.[1]

3.1 Problem Overview

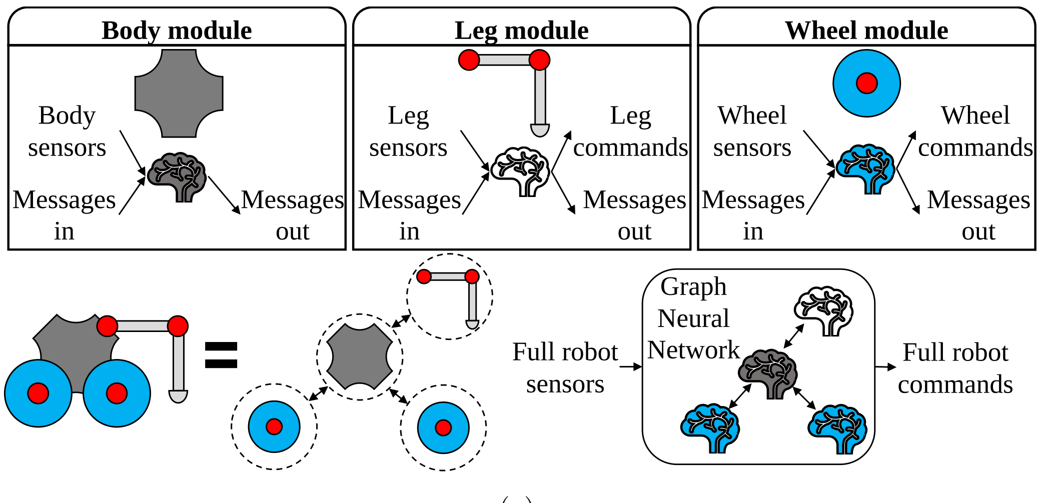
In this section, we first define the modules and design graph. Then, we describe the objective function used to optimize modular policies for locomoting robots.

3.1.1 Module and design graph

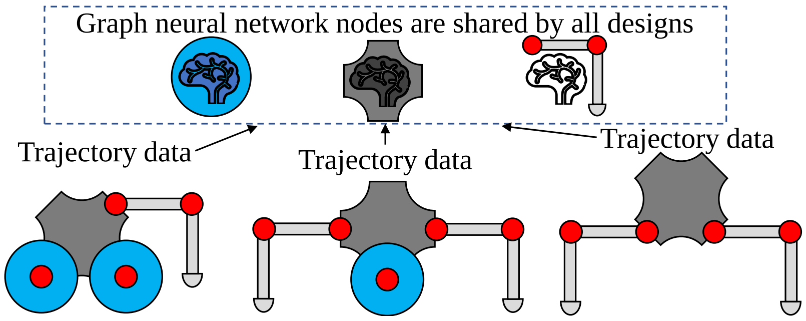
We represent each robot system with a design graph, where a node corresponds to a module, and an edge corresponds to the electromechanical connection between two modules. In this chapter, the example modules we use to ground the discussion are a two-DoF steered wheel, a three-DoF leg, and a rigid body with no actuation. A design d has N_d modules, and can contain multiple modules of the same type. For example, a robot made up of a body, two wheel, and four leg modules, uses all three module types and has $N_d = 7$ total modules in it.

Let M represent the number of types of module, where each type has an index $m = \{1, \dots, M\}$; here we will use $M = 3$ for the leg, wheel, and body modules. Note that this index refers to the type of module, and not an instance of that module in the robot; a single module type may appear multiple times within a design. Each module naturally has a state, makes observations through its sensors, and executes actions through its actuators. Let $x_m \in \mathbb{R}^{n_{x,m}}$ be the state of a module type m where $n_{x,m}$ is the size of that module's state vector. Likewise, let $o_m \in \mathbb{R}^{n_{o,m}}$ and $u_m \in \mathbb{R}^{n_{u,m}}$ be the observations and actions, respectively, for module type m where $n_{o,m}$ is the number of

[1]This work is currently under review. A preprint of the journal paper is available at `https://arxiv.org/abs/2105.10049`



(a)



(b)

Figure 3-2: Our modular policy architecture: (a) the modules, depicted by the three boxes in the upper left, can be composed to form different designs. Each module has a deep neural network associated with it, indicated by the brain icons, which processes that module's inputs (sensor measurements), outputs (actuator commands), and messages passed to and from its neighbors. Assembling those modules into a robot (bottom left) creates a graph neural network (GNN), with a structure reflecting the design, where nodes and edges correspond to the modules and connections between them. The architecture is decentralized in form, but due to messages passed over the edges that influence the behavior of the nodes, the graph of networks can learn to compute coordinated centralized outputs. Note that the blue brain is drawn twice in the graph neural network to indicate that it is used by each of the wheels, but the policy only has one "copy" of the wheel-type neural network which is used to control each of the wheel modules. An alternate depiction is shown in Fig. 3-3. (b) The GNN nodes (top right) are shared by all designs made from these modules. The modular architecture is trained using trajectory data collected from a variety of designs (bottom right) such that it can apply to any combination of the modules.

observations and $n_{u,m}$ the number of output actions for module type m. Observations contain partial and/or noisy state measurements. The full states $x \in X$, observations

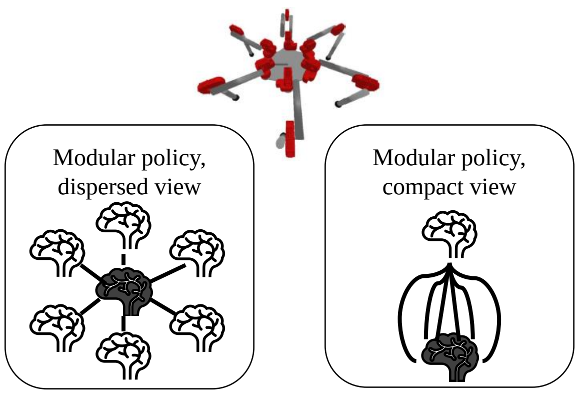


Figure 3-3: There are two equivalent ways of depicting the modular policy, shown in this figure with a hexapod as the example design. In both cases, we color-code the brains to indicate that the node type either corresponds to legs (white) or body (grey) nodes. One way, which we favor in most of this work for simplicity (left), is to show each node in the policy graph as a separate entity. The neural network parameters used by all nodes of the same color are the same: the top left leg uses the same set of neural network weights to process its inputs, messages, and outputs, as does the top right leg, and any other legs present. The second way, which we include here as an alternate perspective (right), is to show only the neural networks instantiated in the code. Only one "copy" of each node type exists. But, the body node sends and receives six different messages over the graph edges, and each of those messages is processed by the leg node to control each of the six legs.

$o \in O$, and actions $u \in U$ of any design are the union of states, observations, and actions of that design's component modules. Note that the dimensionality of these spaces, for any given design, vary depending on the modules in that design.

Each module type has a $N_{\text{ports},m}$ ports, which form connections between modules, and exchange power and data. Each port has at most one edge connecting to one neighboring module, or that port may be empty if no module is attached to it. In our module set, we created a body with six ports, such that the robot can have up to six limbs. The wheel and leg modules have one port each where they can connect to the body.

3.1.2 Modular policy optimization problem

During training, our objective is to obtain reactive policies optimized for a set of K modular robot designs $D = \{d_1, d_2, \dots d_K\}$. The K designs may be fewer than the total number of possible combinations of modules. [2]

In order to specify locomotion with various headings and speeds, we input a target velocity to the policy in addition to the robot's sensor observation inputs. We call this auxiliary input the policy "goal" $g \in G$, the target desired velocity for the robot to achieve, represented by a linear velocity (v_x, v_y) and yaw angular velocity (ω_z), so $g = [v_x^{\text{des}}, v_y^{\text{des}}, \omega_z^{\text{des}}], G \subseteq \mathbb{R}^3$. During training, these desired velocities are sampled from a distribution $g \sim \mathcal{G}$. $\mathcal{G}$ is a distribution over G, e.g., a uniform distribution over a bounded range of velocities. At deployment time, goals can come from a user (e.g. joystick teleoperation) or from a high-level planner. Without loss of generality, goals could also contain other desired state conditions. The policy $\pi : O \times G \to U$, conditioned on a design, takes an observation and goal as input and outputs actions for all modules in the design. We condition the policy on a desired body velocity, which has not been shown by previous MBRL methods. The policy π takes the form of a GNN with parameters θ, which will be described in detail in the next section.

The overall objective of policy optimization is to minimize a cost function $C : X \times U \times G, \to \mathbb{R}^+$. The cost penalizes deviations of the velocity from the desired velocity, e.g. $C(x, u, g) = ||[v_x, v_y, \omega_z] - g||^2 + c(x, u)$. The cost function also includes additional penalties c include regularizing the control input norm, rate of control variation, as well as the roll, pitch, and height of the body. Further cost function details are listed in the Appendix. The policy optimization problem can be written as

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \quad \underbrace{\mathbb{E}_{g\sim\mathcal{G}}}_{\substack{\text{Expectation}\\ \text{over goals}}} \left[\overbrace{\frac{1}{K} \sum_{d \in D} \underbrace{\sum_{t=1}^{T} C\big(x_t, \pi_\theta(o_t, g), g\big)}_{\text{Cost for individual design } d}}^{\text{Average over } K \text{designs in } D} \right]. \tag{3.1}$$

Over the course of a trajectory of length T, the state evolves according to an underlying forward dynamics transition $x_{t+1} = f(x_t, u_t)$. The dynamics f are different for each design. We assume f is not known analytically, but robot-environment interaction data can be accessed from a simulation. We develop a GNN architecture to

[2]In this chapter, these designs are assumed to be given. Future work will address automatic selection of the design training set.

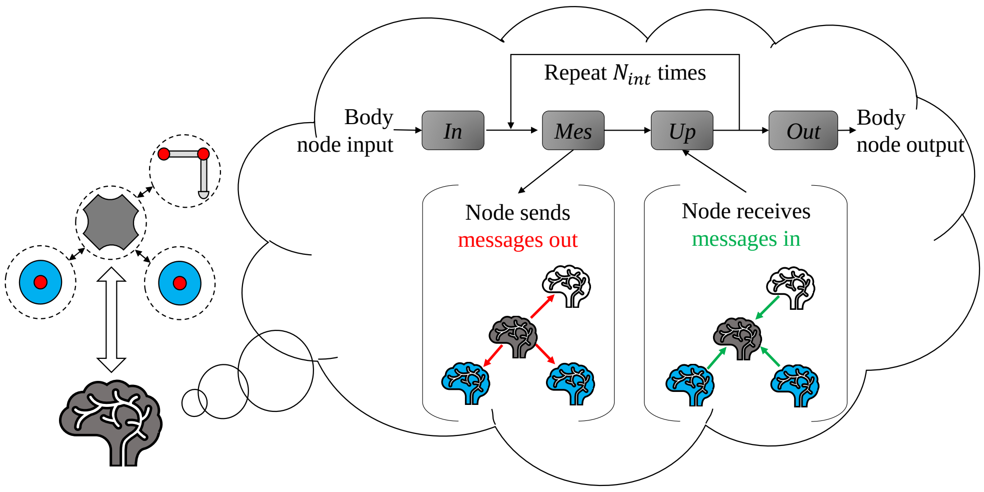


Figure 3-4: An illustration of the graph neural network from the point of view of the body module node, depicted as a dark gray brain icon. Each module in the robot (left side) has a graph node, which undergoes a “forward pass” indicated by the contents of the thought bubble. The node first obtains the relevant input (e.g. sensor observations from the body). Then, within the space of a single time step in the real world, all nodes compute a series of N_{int} internal propagation steps. During these steps, the nodes exchange messages to propagate information through the graph. All nodes undergo these steps at once, then compute outputs (e.g. control actions for each modules' actuators). See Sec. 3.2 for further descriptions of these functions.

approximate f and represent π.

3.2 Graph neural networks for modular robots

The modular policy and approximate model are implemented as GNNs [137, 162, 173], deep function approximators comprised of a network of neural networks. In our implementation, each node in the GNN has a node type corresponding to its associated module type. For any design d, the connectivity of the GNN is set to match the connectivity of the physical hardware graph with nodes $\{\nu_1, \ldots, \nu_{N_d}\}$. Fig. 3-2 illustrates the GNN architecture. The functions mapping inputs to outputs (a.k.a. the neural network “forward pass”) for a GNN are more complex than they are for conventional multi-layer perceptrons (MLPs, also known as dense neural networks). MLPs process inputs by sequentially passing vector-structured data through a series of layers. GNNs must use a more sophisticated series of internal functions to operate

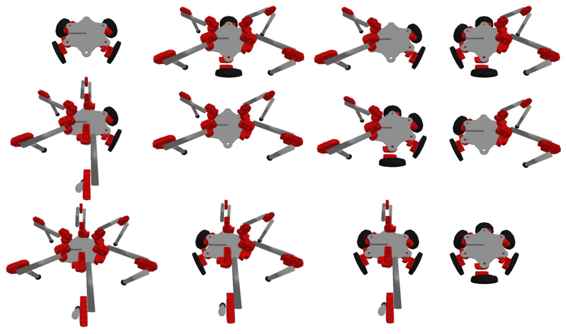


Figure 3-5: The twelve "training set" designs with different arrangements of legs and wheels used in our experiments. In our learning process, the approximate dynamics models and control policy parameters are shared among these designs, chosen for their bilateral symmetry. The policy can generalize to a much larger "test set" of asymmetric designs.

on graph-structured data.

3.2.1 Graph neural network internal functions

The GNN forward pass uses a series of functions: first an input function, then multiple internal steps with message-passing and internal update functions, then lastly an output function. The form of our GNN is inspired by [162]. Fig. 3-4 illustrates the process of a forward pass from the perspective of the body node, and Algorithm 1 describes it in pseudeocode.

Input function

At each time step t, each node ν receives an observation o_ν, which is passed through an input function F_{in} to produce an hidden state vector $h_\nu^0 = F_{\text{in}}(o_\nu)$. Here we use the subscript to indicate that a vector belongs to the node ν. Each node maintains its own hidden state h. Nodes take as input their local parts of the full robot's observation in minimal coordinates. For example, a leg module node takes in the local joint angles and velocities from a leg's three joint encoders, but does not require information about the Cartesian position of that limb. The body node takes in its orientation, linear, and angular velocities from the IMU sensors on the body.

Message-passing propagation

In the GNN, there are two notions of "time." The first is the standard time step discretizing the dynamics and controls, during which the GNN forward pass occurs. The second notion occurs inside the span of each real-world external time step, when multiple computation steps occur inside the GNN during a single forward pass. In the space of one external time step, the GNN computes a series of internal propagation steps to pass messages (real-valued vectors whose content will be learned) between nodes. This learned communication protocol occurs internally to the network during each time step; it provides a means for the nodes of the GNN to produce collective coordinated outputs. Each module learns to alter its behavior depending on the messages it receives, and learns to pass messages that inform other modules how to alter their behavior, to achieve the full robot's goals.

At each internal propagation step, each node sends outgoing messages, receives incoming messages, and then uses those messages to update its hidden state. Specifically, after the input function, the graph undergoes N_{int} internal propagation (message-passing) steps within a single time step. Let e represent an edge connecting ν to a neighboring node. At internal propagation step $i \in \{1 \ldots N_{\text{int}}\}$, each node converts its hidden state into outgoing message vectors $\mathbf{m}$ that will be sent over each of its edges using an output function, $\mathbf{m}_{e,\nu} = F_{\text{mes},e}(h^i_\nu)$. The superscript on h indicates the internal propagation step index. The message output function F_{mes} sends a separate message to each port. The content of the messages is a learned output of the node, and not directly human-interpretable.

After each node computes its outgoing messages, each node reads all messages received from its neighbors. Those messages are concatenated into a vector $\mathbf{m}_{\text{in},\nu} = [\mathbf{m}_{1,\nu} \ldots \mathbf{m}_{N_{\text{ports}},\nu}]$. When a module's port is unoccupied, the node receives zeros as messages over that port.

The maximum number of input and output ports on each node are fixed according to the ports on the modular hardware. Then, by concatenating incoming messages, the receiving node can easily learn to determine the source of incoming messages.

In contrast, recent related work [119, 134, 162] averaged incoming messages, which prevented the receiving node from determining their source. Fixing the number and order of messages allows the nodes to implicitly learn to send information about the relative location of the receiver module to the sender, and as a result, allows modules to adapt their behavior according to their placement on the body.

Update function

Each node uses the incoming messages to update its hidden state via an update function $h_\nu^{i+1} = F_{\text{up}}(\mathbf{m}_{\text{in},\nu}, h_\nu^i)$. The message computation and internal update functions are called repeatedly for N_{int} internal propagation steps, iteratively integrating information from incoming messages into the hidden states.

Output function

After N_{int} internal propagation steps, all of which occur within a single time step, each node computes an output from its hidden state via an output function $F_{\text{out}}(h_\nu^{N_{\text{int}}})$.

3.2.2 Implementation

Each module type has its own instance of the input, update, message, and output functions (F_{in}, F_{mes}, F_{up}, F_{out}). We use MLPs within each of these functions, although other function representations could be used as well.

An important feature in our application of GNNs to create modular policies is that all instances of a module type share the same network weights for the GNN internal functions. The policy parameters are divided by module type, $\theta = [\theta_1 \dots \theta_M]$. Then, θ_m are the parameters used in functions (F_{in}, F_{mes}, F_{up}, F_{out}) for all modules of type m. Each module type has the same parameters regardless of the design in which they are used, so the number of learned parameters scales with the number of module types, and not with the number of designs or number of total modules. When invoked, the GNN nodes are automatically connected to match the design graph, arranging the nodes into the same connectivity as the hardware. For example, in a hexapod robot

Algorithm 1 Message passing graph neural network forward pass described by Sec. 3.2 . Our algorithm uses one GNN as an approximate dynamics model and another as a policy.

```
 1: Collect graph-structured observation $o$ from robot for the current time step.
 2: for $\nu \in \{\nu_1, \ldots, \nu_{N_d}\}$ do
 3:     Apply input function $h^0_\nu = F_{\text{in}}(o_\nu)$
 4: end for
    Message passing internal propagation steps
 5: for $i = 0 \ldots N_{\text{int}}$ do
 6:     for $\nu \in \{\nu_1, \ldots, \nu_{N_d}\}$ do
 7:         for Each edge $e$ of node $\nu$ do
 8:             Compute message function $\mathbf{m}_{e,\nu} = F_{\text{mes},e}(h^i_\nu)$
 9:         end for
10:         Send messages to neighbors over graph edges
11:     end for
12:     for $\nu \in \{\nu_1, \ldots, \nu_{N_d}\}$ do
13:         Aggregate incoming messages,
            $\mathbf{m}_{\text{in},\nu} = [\mathbf{m}_{1,\nu} \ldots \mathbf{m}_{N_{\text{ports}},\nu}]$
14:         Apply update function $h^{i+1}_\nu = F_{\text{up}}(\mathbf{m}_{\text{in},\nu}, h^i_\nu)$
15:     end for
16: end for
17: for $\nu \in \{\nu_1, \ldots, \nu_{N_d}\}$ do
18:     Apply output function $F_{\text{out}}(h^{N_{\text{int}}}_\nu)$ to obtain either a next state $x$ (model net-
        work) or action $a$ (policy network).
19: end for
20: Return the graph-structured outputs.
```

(a body and six leg modules), the GNN contains six leg nodes which all share the same neural network parameters, and a body which has its own parameters. Each leg module uses the leg-type node parameters to compute their hidden states, messages, and outputs separately. To properly coordinate full-robot locomotion, the legs learn to alter their behavior according to the messages passed to them via the body module. An example of the modular policy structure is shown in Fig. 3-3.

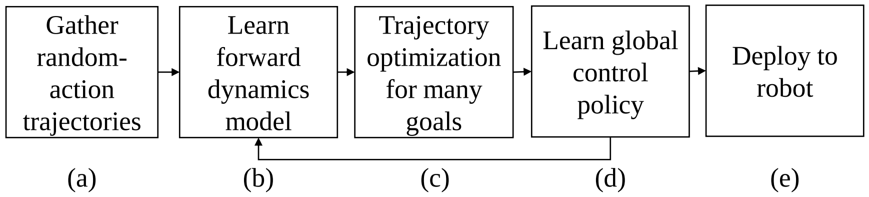

Gather random-action trajectories
Learn forward dynamics model
Trajectory optimization for many goals
Learn global control policy
Deploy to robot
(a)
(b)
(c)
(d)
(e)

Figure 3-6: An overview of our model-based reinforcement learning process, described in detail in Sec. 3.3. All steps are applied simultaneously to multiple robot designs, which share one set of graph neural network parameters. *(a)* First, data is collected for random control actions from which *(b)* initial dynamics approximations are learned. *(c)* Next, the learned model is used to optimize trajectories for the various designs to locomote in a range of headings and speeds. The resulting trajectory data is used to improve the dynamics approximation. *(d)* A global control policy is learned that distills the set of optimized trajectories. *(e)* Finally, the policy is tested in simulation, then validated on physical robots.

3.3 Training Modular Policies with Reinforcement Learning

We now turn to training the modular policy to produce effective behaviors for a range of designs. A large number of neural network policy optimization methods could be adapted to this modular policy learning problem, including, but limited to: evolving neural network weights [147], imitating an optimal control expert [88], model-free reinforcement learning (see Sec. 2.2.3), or model-based reinforcement learning (see Sec. 2.2.3). To solve the policy optimization problem (3.1), we use model-based reinforcement learning (MBRL). We choose an MBRL-based method because they have been shown to be more computationally efficient than alternatives [37, 113, 124]. The main difference between our algorithm and prior MBRL methods is that where prior MBRL learns for a single robot design, we include multiple designs simultaneously.

MBRL first learns an approximate model of the system dynamics for use within trajectory optimization (for brevity, henceforth referred to as "TrajOpt"). Then, similarly to Guided Policy Search (GPS), the well-known idea from the MBRL literature [28, 99, 100, 184], optimized trajectories are used within imitation learning, resulting in a global reactive control policy. Our training algorithm is shown at a high-level graphically in Fig. 3-6, and in more detail by Algorithm 2.

The most significant difference between our algorithm and prior MBRL/GPS methods is that we apply one set of neural network parameters to, and synthesize data from, a varied set of the many possible designs, shown in Fig. 3-5. Two separate GNNs– a model GNN and a policy GNN– are trained as part of our MBRL algorithm. The model GNN, $\tilde{f}_\phi$ with parameters ϕ approximating the forward dynamics, takes the robot state and action as input and outputs the estimated next state. The model is used in TrajOpt, then refit with data gathered from low-cost regions of the state space visited by the optimized trajectories, becoming more accurate in those regions, such that in the next iteration we obtain trajectories closer to those that would be optimal under the true dynamics [95]. The policy GNN, π_θ with parameters θ, takes the robot state as input and outputs actions used as control set-points for each module's actuators.

With MBRL, it is possible to learn from real-world robot data [176]. It is time-consuming and expensive, however, to gather such data from a variety of robot designs. We therefore collect all robot data in a simulation environment. This means training must take additional considerations, described throughout this section, to ensure that the resulting policies can be used on robot hardware.

Our method produces a reactive control policy mapping directly from robot sensor observations to actuator signals. This stands in contrast to recent MBRL approaches [37, 176] that use a learned approximate dynamics model for model predictive control (MPC) to produce actions applied to the robot. GPS methods [28, 99, 184] add an additional step, using optimized trajectories to learn a global reactive policy via imitation learning, which is then applied to the robot. We adopt the latter approach for the following reasons, which overlap with those recently noted by Kaufmann et al. [88]:

- We use full states within TrajOpt, which includes quantities like body position and velocity, and directly impose a cost on those quantities within the trajectory optimization. But, we learn a global policy that operates over partial observations, mitigating the need to accurately estimate those quantities on a physical robot.

- MPC in real-time can become computationally expensive to run on-board a robot when compared to using a single forward pass of a neural network at each time step.
- The global policy can be used to provide initial seeds to TrajOpt. In our method the approximate dynamics is relearned from the states gathered during TrajOpt. This causes the model to become more accurate in regions of the state space near the policy, which leads to better TrajOpt results in the next iteration.

In the remainder of this section, we present the process for each step in the algorithm, along with associate experiments demonstrating their efficacy when applied to multiple designs simultaneously. The hyperparameters for the various components in this method were tuned by hand and can be found in the appendix.

3.3.1 Initial trajectory collection

First, trajectories from random actions were gathered, similar to the data collection methods in [113, 134]. These trajectories are used to learn initial approximation of the dynamics model, and do not resemble the trajectories that are obtained by later stages of the algorithm. To create smooth random actions, 10 random values were chosen for each joint and splines fit to those values equally spaced over 100 time steps. Each design was simulated for a number of trajectories proportional to the number of joints in that design. If the robot flipped onto its side (roll or pitch magnitude exceeds $\pi/2$) then that trajectory was ended. Each trajectory $\mathcal{T} = (x_0, u_0, \ldots x_{100})$ was added to a dataset $\mathcal{T}$.

3.3.2 Learning modular forward dynamics approximations

The dynamics model approximation network $\tilde{f}_\phi$ was learned from the trajectories contained in $\mathcal{T}$. The model network learns to approximate the change in state between

Algorithm 2 MBRL for modular robots. Each step is conducted for all designs in the training set.

```
 1: Collect dataset $\mathcal{T}$ from random action trajectories.
 2: for $i = 1 \dots N$ do
 3:     Learn model $\tilde{f}_\phi$ from $\mathcal{T}$
 4:     $\mathcal{T}_{new} \leftarrow \emptyset$
 5:     for $j = 1 \dots M$ do
 6:         Trajectory optimization:
 7:         if $j > 1$ then
 8:             Sample initial state from $\mathcal{T}_{new}$
 9:         else
10:             Use nominal initial state
11:         end if
12:         for $k = 1 \dots R$ do
13:             Use current policy $\pi_\theta$ and model $\tilde{f}_\phi$ to predict the next $T$ actions, $u^0_{1:T}$
14:             Use $u^0_{1:T}$ as initial seed for trajectory optimization with dynamics $\tilde{f}_\phi$ to
                obtain control $u_{1:T}$
15:             Simulate $n_{ex} < T$ steps forward with control $u$
16:         end for
17:         Add trajectory from simulation to $\mathcal{T}_{new}$
18:     end for
19:     Learn policy $\pi_\theta$ from $\mathcal{T}_{new}$ with behavioral cloning
20:     $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}_{new}$
21: end for
```

time steps, similar to the models used by [113, 134, 176],

$$\tilde{x}_{t+1} = x_t + \tilde{f}_\phi(x_t, u_t) \tag{3.2}$$

where $\tilde{x}_{t+1}$ approximates the true next state $x_{t+1} = x_t + \Delta x_t$ for a fixed time step Δt. The dynamics of each design is different, but all designs share the same model GNN, trained with batches of data from the training set of designs.

This model approximation can be learned using standard supervised regression, but additional techniques can increase the accuracy of this approximation in making predictions over multiple time steps. We adapt two of these techniques to our modular model learning: probabilistic neural networks [37] and a multi-step loss [176].

Probabilistic graph neural networks

A probabilistic neural network is one whose output variables are interpreted as the parameters of a probability distribution rather than as a deterministic value [37]. In our case, this means that the GNN outputs at each node are a mean and a variance of a Gaussian with diagonal covariance, that is,

$$\tilde{f}_\phi \sim \mathcal{N}\big(\mu_f(x_t, u_t), \Sigma_f(x_t, u_t)\big) \tag{3.3}$$

Then, the corresponding log-likelihood loss function for a batch of N data samples is

$$\begin{aligned} L_f &= -\sum_{n=1}^{N} \log \tilde{f}_\phi(\Delta x_n | x_n, u_n) \\ &= \sum_{n=1}^{N} (\mu_f(x_n, u_n) - \Delta x_n)^\intercal \Sigma_f^{-1} (\mu_f(x_n, u_n) - \Delta x_n) \quad + \log\det \Sigma_f(x_n, u_n). \end{aligned} \tag{3.4}$$

This allows the learned model to capture heteroscedastic noise, and has been found to result in more accurate models even when the data, generated from simulation, is not inherently noisy [37, 96]. We found that in practice, it also allows the networks to properly scale the relative loss contributions from state components with different orders of magnitude such that batch normalization as used by [113, 134] was no longer necessary.

Multi-step probabilistic loss

A learned approximate model is not guaranteed to stay within physically meaningful states when used to predict dynamics over long time horizons [91, 113]. To mitigate such divergence effects, one recent approach is to penalize deviations from the ground truth over sequences of states [176], rather than from single state transitions as in (3.4). We adapt the multi-step loss from [176] for use with a probabilistic network,

$$L_{f,ms} = \sum_{n=1}^{N} \frac{1}{T} \sum_{t=n}^{n+T} \big[(\mu_f(\hat{x}_t, u_t) - \Delta x_t)^\intercal \Sigma_f^{-1} (\mu_f(\hat{x}_t, u_t) - \Delta x_t) + \log\det \Sigma_f(\hat{x}_t, u_t) \big], \tag{3.5}$$

where $\hat{x}$ are recursively predicted states $\hat{x}_{t+1} = \hat{x}_t + \mu_f(x_t, u_t)$, and when $t = n$ (the first state in the multi-step sequence) $\hat{x}_t = x_n$ is sampled from dataset $\mathcal{T}$. The recursively predicted sequence of states measures the deviation over time of the learned dynamics. We use sets of $T = 10$ sequential states to compute this loss.

One drawback to this approach is that the gradients of this loss become increasingly expensive to compute as the sequence length increases and the network is called recursively multiple times. This is compounded when using a GNN, which already has more complex gradients than a MLP. To reduce training time, we used a form of curriculum learning [17], in which the multi-step sequence length started is incrementally raised over the course of training. At the start of training, we set $T = 1$ in (3.5) and then periodically increase it up to $T = 10$. This adaptation resulted in the same trained model accuracy with significantly less computation.

Multi-design training

$\tilde{f}_\phi$ is trained to approximate the dynamics of the training set of designs. During training, we sample batches of N state-action sequences (i.e., short trajectories) $\mathcal{T} = (x_n, u_n, x_{n+1}, u_{n+1} \ldots x_{n+T})$ for each design, compute (3.5), and accumulate the gradients over multiple designs before taking an optimization step with an Adam optimizer [90]. Averaging the loss over multiple designs prevents the model from over-fitting to any specific design, and instead, to fit jointly to all designs.

To further reduce computational load, we applied a form of curriculum learning over the number of designs included in each step. At each training step, a subset of designs were sampled for a forward pass, rather than including all designs in the loss. The number of sampled designs was incrementally increased until all training set designs were used at each step. This adaptation also resulted in the same trained model accuracy with significantly less computation.

Translation and yaw invariance

The dynamics of motion, under a constant gravitational field and in a uniform flat environment, are invariant to the translation and yaw of the system. Prior work [113, 134] learned the model in the world frame, then subtracted out the body translation to compute network inputs. We extended this translation-invariance with an additional inductive bias by recognizing the symmetry of the dynamics with respect to not only the translation in the plane but also the yaw of the body.

The model dynamics were learned in a frame which we call the "planar body frame." This frame is different from the conventionally-defined body frame, as the height, roll, and pitch of the body are still relevant when the dynamics occur under the external force from gravity. The x-position, y-position, and yaw γ in the plane were removed from the state. Then the world velocities were rotated by the negative yaw; for example, the body world-frame linear velocity $v \in \mathbb{R}^3$ was rotated to the planar body frame velocity $v_B = R_z(-\gamma)v$, where $R_z(\cdot)$ represents an SO(3) rotation matrix about the z-axis. The state transitions (and also the global policy, discussed later) were learned with respect to the planar body frame. The change in x-position, y-position and yaw with respect to that frame was predicted by f_ϕ.

3.3.3 Trajectory optimization with a learned model

The next step in the algorithm is to use the learned model in TrajOpt. The goals and states visited by the optimized trajectories will ultimately be used to train a global reactive control policy. Note that each of these steps is applied at each iteration to all designs in the training set.

Optimization and Model Predictive Control

Each TrajOpt solves for a series of control inputs u that minimize an objective function C over a finite horizon length H. Each trajectory was given a constant body velocity matching goal g within the objective function, for a minimization problem,

$$\begin{aligned} u^*_{0:H} = \operatorname*{argmin}_{u_{0:H}} \sum_{t=0}^{H} C(x_t, u_t, g) \\ \text{s.t.} \quad x_{t+1} = x_t + \mu_f(x_t, u_t). \end{aligned} \tag{3.6}$$

In the TrajOpt, the dynamics evolve according to the mean predicted by the model GNN. The initial state x_0 in each trajectory has a significant impact on the full MBRL process and on transfer from simulation to reality, as it governs the states used to train the policy, and will be discussed later.

The objective of the TrajOpt process is to create a dataset $\mathcal{T}_{new}$ of "expert" demonstrations showing robots tracking various goals g from many initial states x_0. As such, for each trajectory, we sample a new body velocity from the bounded range of goal body velocities G. The cost function C penalizes deviations of the body velocity from the desired body velocity, as described by Sec. 3.1.2. Further penalties in the cost function include costs on the control input norm, as well as the roll, pitch, and height of the body. A cost term that we found to be critical is the "slew rate" penalty, which penalizes abrupt changes in the control inputs. While trajectories without this penalty perform well in simulation, they did not transfer well to physical hardware, where actuators perform poorly when commanded to frequently abruptly change direction.

Many TrajOpt methods exist to find locally optimal solutions to (3.6). Prior work [113, 176] used simple gradient-free random shooting methods. We found that such methods suffer from the curse of dimensionality when applied to high-dimensional systems like our hexapod. Instead, we turn to a gradient-based method, differential dynamic programming with input constraints [8, 153], which is able to exploit model linearization to efficiently find locally optimal control inputs. Other TrajOpt algorithms could be used as well. Batches of trajectories were optimized at once using batched forward passes of the learned model.

To create a trajectory, a start state x_0 and velocity goal g is sampled, a local solution to (3.6) is solved under the approximate model. However, the approximate model is not guaranteed to stay within physically meaningful states when used to predict dynamics over long time horizons [113, 176]. To mitigate such divergence effects, we combined the multi-step loss described above with TrajOpt in a model-predictive control fashion [113, 176]. That is, we set the horizon length H, solve (3.6), then execute the first n_{ex} steps of the optimized control in the simulation environment.

The remaining $H - n_{ex}$ steps are then reused as part of the initial seed for the next replan. This process was repeated R times for each goal, resulting in each trajectory $\mathcal{T}_{mpc}$ of length $n_{ex}R$. The trajectories $\mathcal{T}_{mpc}$ are stored in a dataset $\mathcal{T}_{new}$.

Initial seeds

The local TrajOpt requires an initial control input seed, which had a significant impact on the quality of the solution at convergence. During the first MBRL iteration, when the global policy is entirely untrained, we used zeros as initial control seeds. During subsequent iterations, we use the global policy rolled out on the learned model to create initial control seeds. This ultimately resulted in lower-cost trajectories than always using zeros as initial control seeds. More importantly, the optimized trajectories end up nearby the global policy, and the policy is then retrained from those trajectories. This process iteratively reinforces a consistent gait style. Without using the policy as the initial control seed, motions generated by TrajOpt were dissimilar between iterations, and cyclical locomotion patterns did not emerge.

Initial states

We found the initial state set in the TrajOpt to have a significant impact on the policy's ability to change locomotion heading on-the-fly both in simulation and reality. While prior GPS learned policies for forward locomotion [99, 184], our objective is to learn policies that move the robot in any direction in the plane, and to be usable with tele-operation. Consequently, policies must have the ability to quickly change direction and speed. The policy is learned via imitation, given optimal "expert" trajectories demonstrating the robot changing directions.

To create trajectories that contain rapid direction and speed changes, we follow the following steps. During the first M_{init} trajectories in each iteration, the initial state x_0 was set to a nominal state, standing upright at zero velocity, as shown in Fig. 3-5. These trajectories provide expert examples showing the robot starting from rest, stored in $\mathcal{T}_{new}$. Then, we sample additional initial states from states visited in $\mathcal{T}_{new}$. This was inspired by the methods of Zhang et al. [184], who noted that such a process creates overlap in the state distributions visited by the optimized trajectories. Since the velocity and heading goal of the new trajectory were sampled independently of the goal used to create that sampled state, this resulted in trajectories in which the robots abruptly change directions mid-step, an essential behavior to capture if the end user will be creating the heading on-the-fly with a joystick. To further enable transfer to reality, we injected a small amount of noise to each sampled initial state, in the form of small perturbations to the joint angles and velocities. This provided demonstrations for the global policy of how to optimally recover from disturbances.

Gait style objective

When creating controllers for legged robots, experts often inject their intuition or preferences about gait style. For instance, recent work in deep reinforcement learning forced cyclical motions (gaits) to emerge by using reparameterization of the actions space [176] or externally generated cyclical keyframes [121, 174]. While our method does not require keyframes for gaits to emerge, we introduce the option to impose a manually selected gait styling with an additional cost on leg joint angles.

The gait style objective was created by first selecting an amplitude, frequency, and phasing for hexapod joint positions that would result in an "alternating tripod" step-in-place pattern. Deviations of any joints in legs from this pattern are then penalized in C. These open-loop joint angles do not move the robot in any direction nor effect the wheels, and the TrajOpt process must discover how to produce locomotion to minimize the velocity matching cost. This cost resulted in gaits that follow the main body velocity-matching objective while also remaining near an alternating tripod gait style. In designs where the gait style makes locomotion more difficult (for example, in a quadruped), because we set it to a smaller weighting than velocity-matching objective, it can be overcome by TrajOpt. Note that this cost is not necessary for gaits to emerge with our method. When a simpler cost on angles deviating from their nominal stance is used, cyclical gaits still emerged which are equally effective in simulation as those learned with the gait style cost.

3.3.4 Learning the modular control policy

Given a dataset $\mathcal{T}_{new}$ of expert demonstrations, the next step in the MBRL process is to distill these local policies into a global policy via imitation learning. That is, given many samples of robots moving in many different directions, the imitation learning process acts to "interpolation" between the samples.

We use a reactive control policy for the reasons stated at the start of the section: it is simpler to implement on a physical real-time system than is running MPC with the learned model, it is able to operate on partial state observations while allowing the internal trajectory optimization operate on the full state, and it provides initial seeds for TrajOpt. We introduce some modifications to the policy inputs and outputs compared to those of related work [113, 184] in order to facilitate transfer from simulation to reality.

We command target velocities to the actuators, which are tracked by low-level PID loops at a higher frequency on-board the actuator. The actuators (X-series from Hebi Robotics [70]) perform more accurate tracking when provided with a feed-forward (FF) torque value τ. Thus, in addition to control outputs, we learn an additional output of the policy network that estimates the feed-forward torque needed for the actuator to track the desired velocity. The data for this output is obtained by tracking the torques experienced by the joints in simulation, contained in $\mathcal{T}_{new}$.

The global policy outputs control command u and FF torque τ, parameterized by means (μ_u, μ_τ) and diagonal variances (Σ_u, Σ_τ),

$$[\mu_u, \Sigma_u, \mu_\tau, \Sigma_\tau] = \pi_\theta(o, g), \tag{3.7}$$

given an observation o and body velocity goal g as input. The goal is appended to the body graph node input.

The policy is used deterministically, so at runtime, only μ_u and μ_τ are used. But, to avoid batch normalization, we found that interpreting the network outputs as a Gaussian (making it a probabilistic GNN, see Sec. 3.3.2) resulted in a more consistent learning process than we found when using a mean-squared error loss. The policy is learned using a log likelihood [37] behavioral cloning loss,

$$L_\pi = \sum_{n=1}^{N} \left[(\mu_{u,n} - u_n)^\intercal \Sigma_{u,n}^{-1} (\mu_{u,n} - u_n) + w_\tau (\mu_{\tau,n} - \tau_n)^\intercal \Sigma_{\tau,n}^{-1} (\mu_{\tau,n} - \tau_n) \right], \tag{3.8}$$

for a batch of N samples of (o, g, u, τ) drawn from $\mathcal{T}_{new}$. w_τ is a weighting hyperparameter controlling the importance of accuracy of the FF torque predictions relative to the control joint velocity set point outputs. A key feature of our method is that the global policy shares data from, and applies to, the full set of modular designs. That is, the loss over all designs are averaged at each training step.

To further facilitate sim-to-real transfer, we learn the policy with sensor noise and partially-observed inputs. At each iteration within the policy supervised learning process, we add white noise to the observations. The body velocity and height, while easily observable in simulation, require state estimation techniques to observe in reality. To avoid the added complexity of such state estimation, we remove the body velocity and height from the state observation input. We also learn the policy with respect to the planar body frame, as discussed in Sec. 3.3.2, such that the body planar position and yaw are not included in the observation. To account for latency [176], the delay between sensing and actuation, we use the observation from the previous time step as the policy input.

3.3.5 Updating the learned model

The trajectories seen in simulation during TrajOpt form a dataset $\mathcal{T}_{new}$ that provides "guiding samples" [100] to update the dynamics and learn the policy. We retrain the model using both $\mathcal{T}_{new}$ and $\mathcal{T}$, adding samples along trajectories relevant to perform effective locomotion without causing catastrophic forgetting of the dynamics in other states. After the policy is learned, the trajectories in $\mathcal{T}_{new}$ are merged into $\mathcal{T}$, and $\mathcal{T}_{new}$ is reset to empty. Then, training as described in 3.3.2 is continued, warm-started using GNN parameters from the previous iteration.

Note that the TrajOpt problem in (3.6) uses the approximate model $\tilde{f}$, so the resulting trajectories are optimal with respect to those dynamics and not to the true dynamics. However, the model is relearned from trajectories seen during TrajOpt in the previous iteration, increasing model accuracy in the vicinity of low-cost regions in the state space. Subsequently, the optimal trajectories in the next iteration will be closer to the optimum under the true dynamics, and the policy learned from those expert trajectories will be closer to the true solution to (3.1). In other words, iterative process of re-learning the model with data seen during TrajOpt is intended to combat the recently observed "objective mismatch" in MBRL: namely, learning a globally accurate dynamics model does not necessarily lead to higher-quality trajectories [95].

3.3.6 Evaluation metric

In order to evaluate the quality of the local trajectories and the global policy, we developed a metric quantifying the mismatch between the desired and achieved body velocity over a fixed time period. In the case of forward locomotion at maximum speed, as is commonly used in locomotion learning, the distance travelled or average speed serves easily as an evaluation metric. In the case of a multi-direction and multi-speed distribution of desired body velocities, (which we represent as a goal distribution $\mathcal{G}$ in (3.1)), this metric no longer suffices, and we must create a new evaluation metric.

As an evaluation metric we form a fixed "test set" of goal velocities that serve as a finite sampling proxy for the expectation over all possible goals $g \in G$ in (3.1). This test set of goal velocities include moving forward, backwards, left, right, and turning in place left and right. For each of these test goals, we execute the policy. At every n_{ex} step period over the resulting $n_{ex}R$ step trajectory we measured the difference between the desired and achieved average velocity. Under this metric bounded on $[-1, 1]$, higher values are better. A value of 1 would indicate that the robot always moved exactly in the commanded direction, and a metric of 0 indicates that the robot did not move at all. We did not know in advance how fast each design would be physically capable of moving, and therefore a metric value 1 was not achieved in our experiments, because the desired maximum speed in the goals set was chosen based on a rough estimate of the theoretical maximum robot speed (additional details and hyperparameters are described in the Appendix). Similarly, any given design might not achieve the top speed, even were it physically capable of doing so, since the multi-objective TrajOpt cost balances body velocity matching with other costs. Some designs have difficulty in locomoting in a given direction due to their design, for instance, if a goal velocity requires moving perpendicular to the direction of a wheel. We use this metric to track the progress of training over the outer MBRL loop and within policy transfer tests.

3.3.7 Zero-shot transfer to unseen designs

The policy and model GNNs were trained on a set of 12 designs, chosen out of the full set of possible module combinations because they are symmetric along their front-back axis and allowed to have the middle port unoccupied. However, these make up only a small fraction of the total possible space of designs from these modules– if we allow the designs to be asymmetric, there are an additional 132 possible designs, not seen during training. The policy can automatically be applied to each of the 132 asymmetric designs. We conducted an experiment to test zero-shot transfer (that is, without additional training or modification) of our policy to these designs.

3.3.8 Comparison to MLP weight sharing

In our GNN architecture, the model and policy are both hardware-conditioned because the structure of the learning representation matches the physical kinematic structure of the robot, and modules of the same type share information. This stands in contrast to the related hardware-conditioned policies used by [34], which shared all weights among all robots using an MLP. To test whether our learning architecture results in higher-quality policies than other weight-sharing architectures using MLPs, we created two baseline comparisons, which we call "hardware-conditioned MLP" and "shared trunk MLP."

The hardware-conditioned MLP is based directly on the architecture presented by [34]. When initializing the network, a fixed maximum number of modules and maximum dimension of the inputs and outputs are specified. For each robot design the network is applied to, the inputs are padded with zeros to reach this constant maximum dimension length before entering them into the network. For example, the leg module has a state dimension size six (position and velocity for each joint), and the wheel dimension three (position and velocity for the first joint, and wheel velocity for the second). When entering the state of a leg into the network, no zeros need be appended. When entering the state of a wheel module into the network, three zeros are appended to bring the input size up to the maximum length six. The output layer of the network is also set to a maximum output size, and the unused outputs for each module are ignored. The design used during each forward pass is encoded via a one-hot vector, with entries corresponding to the type of each module. This architecture allows transfer to new designs not seen during training.

The shared trunk MLP is a simpler weight sharing scheme loosely inspired by multi-task image recognition architectures [41, 186]. The bulk of the neural network weights are shared by re-use of the hidden layers across the 12 training set designs. To account for the different dimensions of states, actions, and observations among the designs, each design is given its own input and output layers that are not shared. Then at each forward pass, the design index is passed to the network, so that the corresponding input and output layers are used. This architecture does not transfer to new unseen designs without further training, as each design has its own input and output layer specific to its dynamics and dimensionality.

For each of these two network architectures, a separate network instance was used as a model and policy. The number of layers and depth were tuned to approximately match the capacity and depth of the set of GNN nodes. We then applied our MBRL algorithm to measure the policy efficacy via the velocity matching metric. The gait style objective was used in these experiments.

3.3.9 Application to real robots

Our MBRL algorithm was designed to create a control policy that allows a user to drive a variety of robot designs with a joystick in reality. The body contained a battery pack, Ethernet switch, WiFi router, and and IMU. A USB gamepad was connected to an off-board laptop, on which joystick inputs were converted to goals, appended to the control policy input, and joint-level commands were computed and sent via WiFi to the robot.

3.4 Results

Our method enabled us to control a variety of robot designs with a single set of GNN parameters. The modules behave differently when placed in different locations on the robot, even though the GNN node weights are the same for all modules of the same type, because the messages passed between nodes differ depending on the relative position of the modules. Further, the modules behave differently within different designs– for example, the gait pattern that emerges to control the quadrupedal design is different than that of the hexapod. We ran the algorithm for 3 iterations (alternating between batches of model learning, TrajOpt, and policy learning), which resulted in a policy trained for 12 designs, after approximately 10 hours on a large desktop with 18 Intel i9 cores and four NVIDIA RTX 5000 graphics cards. We believe with further code optimization time to train could be reduced, for example, by parallelization of functions that were conducted independently and sequentially for the 12 designs.

The following subsections describe the results of our experiments on model and policy learning with multiple designs, on zero-shot transfer, and demonstrate sim-to-real transfer.

3.4.1 Modular model learning

We conducted an experiment to validate the utility of training a shared model with data from multiple three-dimensional articulated robot designs. We created datasets using the random-action procedure from Sec. 3.3.1 for three designs (as shown in the left-most column of Fig. 3-5): four-wheel car, a six-leg walking hexapod, and a design with two wheels and four legs. We then divided the data into training and validation sets, and trained a GNN and a MLP for each design separately using two data regimes: either 100 trajectories or 1,000 trajectories. The total number of parameters in the GNN and MLP were made comparable in this experiment. We also trained a single shared GNN model using data from all three designs.

We compared the validation error of a forward dynamics approximation network learned via comparably sized GNNs and MLPs within high-data and low-data regimes. We compared further against a GNN trained using data from all three designs. We also compared the prediction validation error between the conditions to a "constant prediction baseline" [134]: the error value that would be obtained if the state change prediction was zero at each step. The results of this experiment are shown in Fig. 3-7. We found that shared weights between modules of the same type helps prevent over-fitting in low-data regime, and also results in lower validation error for the same number of parameters. Data from multiple modules of the same type contribute to the parameters of the corresponding graph node. Where [134] conducted a similar experiment, they used higher-capacity models (e.g. deeper networks) and thus was able to obtain lower validation error than we obtained. However, lower validation error has recently been shown to not necessarily correspond with the cost of trajectories obtained in model-based TrajOpt [95]. We also observed that using data from multiple designs had little impact on the validation error, indicating that parameters can be shared among designs to accurately predict the future states of multiple designs made from the same set of components.

3.4.2 Modular policy learning

Next we explored whether a policy created via our full MBRL process with data shared among 12 designs would perform as well as a policy trained with only one design at a time, with the same hyperparameters. In this experiment, we first use the full MBRL process on 12 designs at once, at each step using data from all designs. Then, we use the full MBRL process on the lowest and highest degree-of-freedom systems (car and hexapod) independently, at each stage using only data from a single design. Table 3.1 shows the evaluation metric applied to the car and hexapod designs;

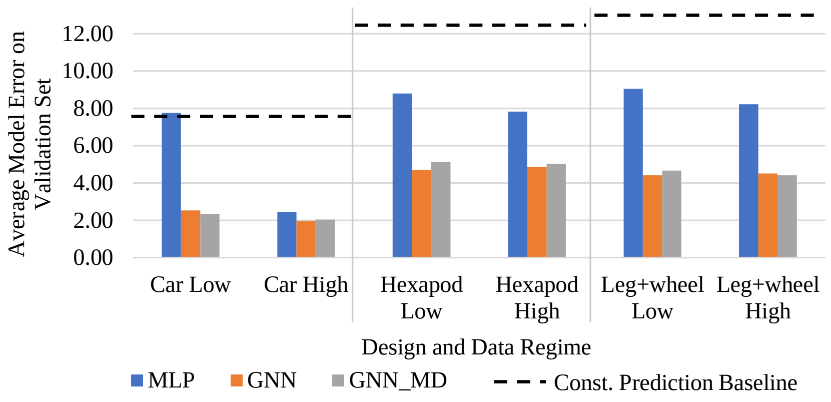


Figure 3-7: Results described by Sec. 3.4.1. The graph neural network (GNN) has a lower validation error than a multi-layer perceptron (MLP) with similar capacity and depth. It also is more data-efficient, achieving comparable validation error with 10 times less data. We tested model learning on three designs: a four-wheel car-like design, a hexapod, and a leg-wheel hybrid. Images depicting these three designs are in the left-most column of Fig. 3-5. Learning a model from shared data of the multiple designs (GNN MD) does not harm the validation error. The constant prediction baseline (dashed lines) indicates the the validation error for predictions of zero state change between steps.

this indicates that the policy trained with shared data between multiple designs is able to perform similarly to a policy trained on only one design.

Velocity matching metric	Hexapod	Car
Trained alone	0.63	0.80
Trained with 12 designs	0.73	0.80

Table 3.1: Modular policy training result for Sec. 3.4.2.

3.4.3 Generalization to unseen designs

We trained the control policy with data shared between 12 bilaterally symmetric designs, then tested the policy on 132 simulated asymmetric designs without further training or optimization. Fig. 3-8 shows the evaluation metric, as described in Sec. 3.3.6, applied to the test set (robots seen during training) and transfer set (not seen during training). While the average metric for the transfer designs is lower than the training set, we found that the policy was able to direct all designs in the commanded direction on average. The designs which performed worst in the transfer tests qualita-

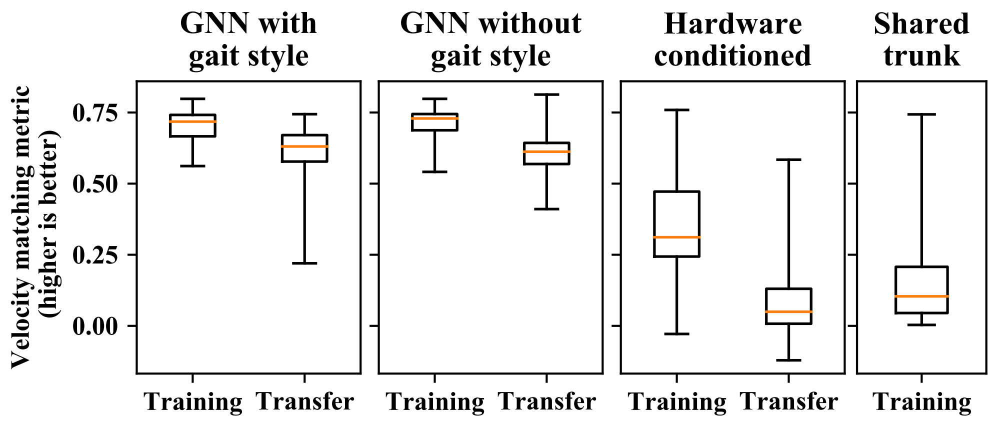


Figure 3-8: Results of applying the policy to the training and zero-shot transfer set of designs. The left and left-center plots show our modular GNN architecture with and without the gait style objective applied, described by Sec. 3.4.3. The right-center and right plots show the baseline comparisons with multi-layer perceptrons, described by Sec. 3.3.8. The training set contains 12 designs, and the transfer set 132 designs not seen during training. The mean of the set is shown in orange, the boxes show the first and third quartiles, and the maximum and minimum of the set are shown by the top and bottom whiskers for each set. The policies were measured using a velocity matching metric, where higher values indicate that the policy tracked the desired robot velocity well. We found that our modular policy results in effective locomotion for different headings and speeds on a range of different robots, and is able to generalize (without additional training) to an even larger set of designs.

tively appear to be those with fewer limbs and more asymmetries in limb placement, such that their dynamics differed most from that of the designs in the training set. The results are similar with and without the alternating-tripod gait style objective applied during training.

3.4.4 Comparison to MLP weight sharing

The results of our weight-sharing baseline comparisons with MLPs are shown in Fig. 3-8. We found that the GNN policy had both a higher mean performance and narrower range of performance among both the training and transfer set of designs. Although sharing neural network parameters centrally for all parts of multiple robots was demonstrated previously by [34] for fixed-base manipulators, we found a hardware-conditioned MLP to be less effective when applied to robots with multiple limbs. This experiment shows that the inductive bias we applied in learning, that is, matching the structure of the graph network to the structure of the robot, enables more effective learning than using a more generic architecture.

3.4.5 Zero-shot transfer comparison

In Sec. 3.4.4 with corresponding Fig. 3-8, we found that our GNN was able to generalize to unseen designs more effectively than a hardware-conditioned MLP. However, the average, max and min performance of the many designs does not reveal which designs the policy can transfer to. We plot the number of leg modules and wheel modules against the transfer results in Fig. 3-9. We can see that the MLP is able to transfer most effectively to designs with wheels, which agrees with our intuition that wheels are "easier" to learn to control.

3.4.6 Physical robot validation

We tele-operated the twelve "training set" robots using the modular policy outdoors on a sidewalk. Fig. 3-10 shows a time-lapse of this demonstration. Qualitatively, most designs performed well, although differences in ground interactions between simulation and reality appear to hamper some of the designs where slipping or dragging contacts occurs.

3.5 Discussion

Model-based trajectory optimization for legged or leg-wheel hybrid robots, such as [20, 21, 53, 171], typically make the assumption that the contact sequence is known a priori. An emergent feature of our work is that the contact sequence can be discovered automatically. This becomes particularly important when we are tasked with creating trajectories for multiple robot designs, because each design may have different combinations of legs and wheels, resulting in a different optimal contact sequence. We added a gait style objective for the case where the user biases learning towards a particular contact sequence. An example of the difference in contact sequences

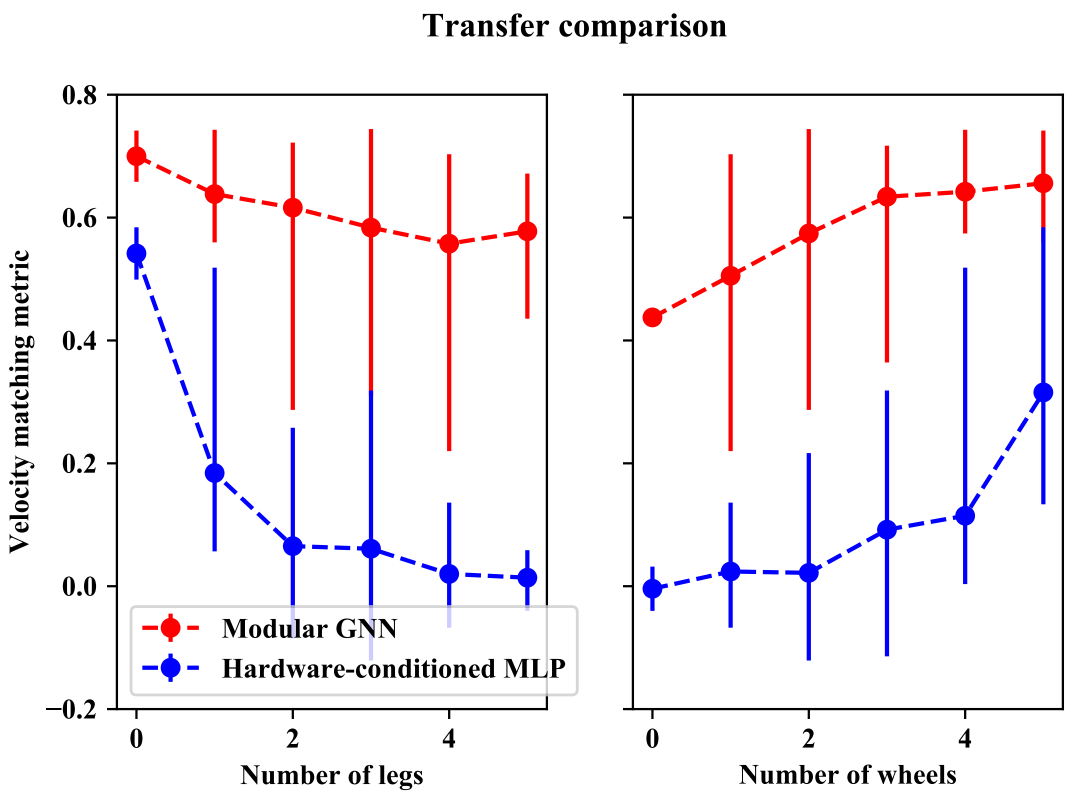


Figure 3-9: The zero-shot transfer test results from Sec. 3.4.4 and Fig. 3-8, broken down by number of leg and wheel modules in each robot design. Multiple designs may have the same number of legs and/or wheels, arranged in different ways. The circle markers show the mean of the designs with a given number of legs or wheels, and the vertical bars indicate the max and min. The velocity matching metric measures how well the robot can match a desired heading and speed using the learned policy. As the number of legs increases and number of wheels decreases, the MLP policy performance degrades significantly more than does the GNN policy. In all cases, our GNN policy is able to transfer more effectively to new designs.

that emerge with and without the gait style objective is shown in Fig. 3-11. When the gait style impedes learning, it is overcome; we found this was the case with the quadruped, which qualitatively has a similar behavior, and quantitatively has a similar performance measure, both with and without the gait style.

The contact sequences that emerge differ among the designs, even though they are the result of a single policy. We found that the interplay between trajectory optimization and global policy imitation learning played a key part in enabling the policy to be effective on many designs. Training the modular policy using the dataset

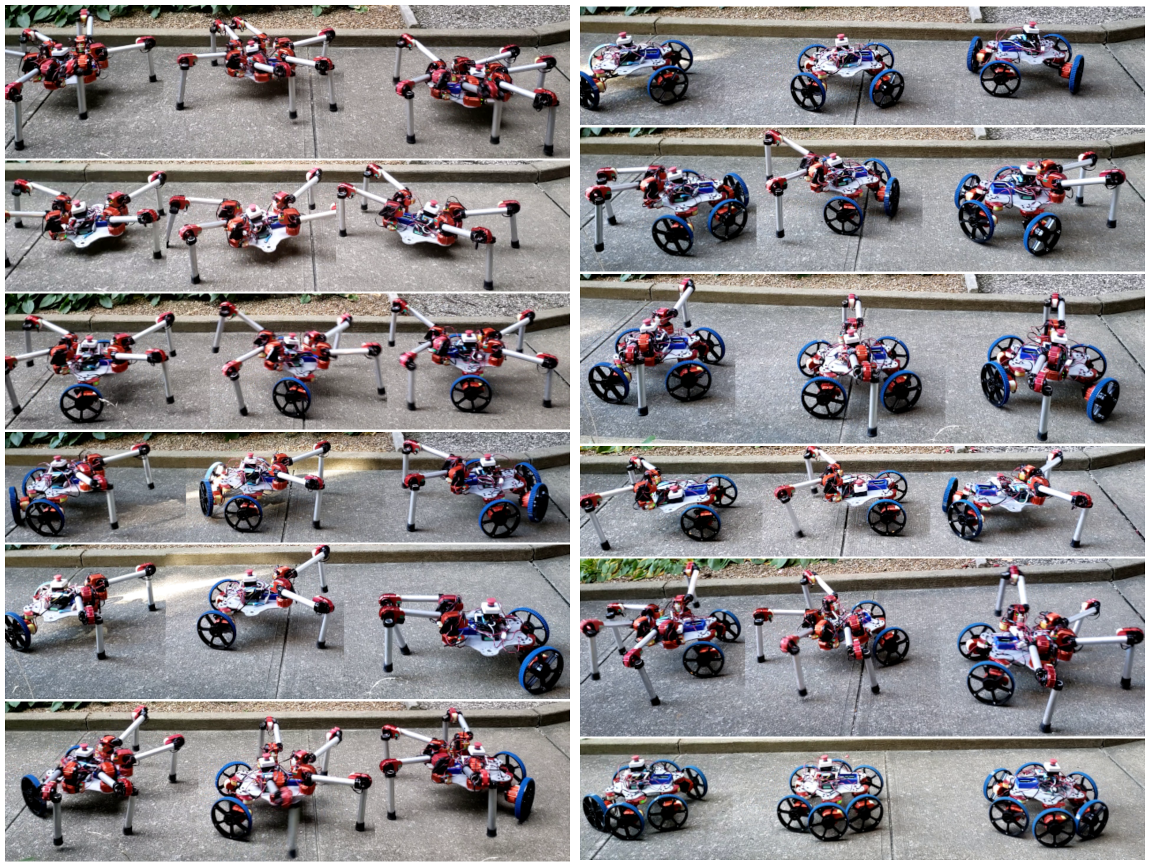


Figure 3-10: Timelapses of the twelve modular designs controlled with the learned policy. The robots were teleoperated with a joystick demonstrating forward and turn-in-place locomotion on a sidewalk.

created by each iteration of trajectory optimization creates module-level behaviors that apply increasingly well to the full range of designs. In the first iteration of the pipeline, before the policy has been trained at all, the initial control seed in TrajOpt is zero. The local trajectories that arise for the different designs appear dissimilar, and some are low-quality local optima. The policy learns to imitate the collective dataset of trajectories, so in the first iteration, the policy may not be effective on even a single design. However, that policy provides an intial seed for the next iteration of TrajOpt, resulting in lower-cost local minima that are more similar across designs. We observe that this effect compounds until ultimately the policy is effective on the full training set of designs.

A number of choices in the algorithm are taken to aid computational efficiency. We allow for intra-limb coordination of multiple joints encapsulating multiple actu-

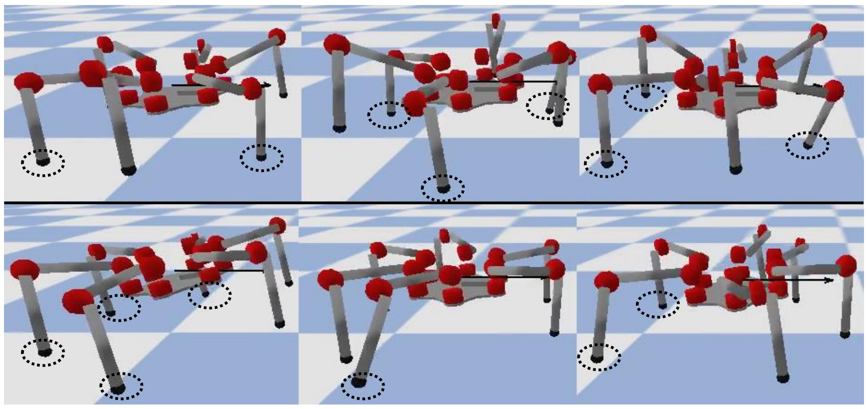


Figure 3-11: A time lapse of the simulated hexapod motion with (top) and without (bottom) the alternating-tripod gait style objective, walking from left to right. The feet in contact with the ground are circled within dotted lines. Both gaits move at a similar speed, but without the gait style objective, a different contact sequence emerges.

ated joints within a graph node, which reduces the number of message passing steps compared to related work [78, 134, 162] in which each node controls, or approximates the dynamics of, one joint. We use smaller capacity models than [134] to learn the dynamics, but find that these are still effective when used within trajectory optimization, especially when combined with the multi-step loss function introduced by [176]. We also showed that the additional inductive bias introduced by shared weights between limbs of the same type helps prevent over-fitting in the low-data regime, resulting in sample efficient training. With these choices, in addition to some strategic curriculum learning as described throughout Sec. 3.3, we were able to conduct training on a single computer without cloud compute resources. We believe this to be an important feature in making deep learning accessible and reproducible.

This chapter introduced a model-based reinforcement learning method to control a variety of modular robot designs with a single policy. Both a dynamics approximation and a global control policy are learned with graph neural networks (GNN) that share parameters among distinct designs and learn from a combination of data from those designs. Our GNN formulation embodies a novel inductive bias [15] in the learning representation and training process relative to prior works [78, 134, 162]: not only is a robot made up of a tree of joints, but there are multiple types of modules repeated in the graph structure of modular designs, and the structural modules without joints impact the dynamics and control as well. The GNN learns how each module type (body, wheel, or leg) should behave within the context of the other modules present in the design. As a result, we observe emergent behavior wherein limbs with the same neural network weights behave differently for different designs and locations on the body. The policy allows a user to drive a range of robots with a joystick, or for the policy to be used as motion primitives within a high-level path planner.

We showed that our policy transfers readily to designs, composed of those same modules that were not seen during training, without additional learning or optimization. We were inspired by the computer vision research community, which has found that the right learning representation (convolutional neural networks) and a diverse set of training examples (dataset of images) enables generalization to images not seen in training [133]. Similarly, we find that for modular robots, a learning representation that stores knowledge about dynamics and controls in the module graph nodes (a GNN), and a diverse set of training examples (designs with various combinations of modules) enables policy generalization to designs not seen in training.

In the development of our methods, we noticed a number of limitations. As noted by [95], there is a fundamental mismatch in the functions being optimized in model-based reinforcement learning– a more accurate model does not necessarily result in a better policy. Our use of Guided Policy Search techniques [100] appear to mitigate this problem, at each iteration increasing the accuracy of the model in regions of low trajectory cost. Future work will thus consider convergence analysis, as well as further study on the effect of the number and size (number of joints) of the modules, as well as the effect of the many other hyperparameters on convergence, in particular when applied simultaneously to many robot designs at once.

Another limitation of our work lies in the simulation to reality transfer. We showed that the policy transfers to reality, but the performance of robots in simulation appears better than in reality. In future work, we will investigate learning from data collected on a combination of simulation and physical hardware. Our experiment, with results in Sec. 3.4.1, found that a GNN can learn from smaller datasets more efficiently than a comparably sized MLP when data from modules of the same type are shared, which shows that our method has the potential to learn from physical robot data. We are also investigating combining our methods with existing sim-to-real techniques like simulated latency [176] or domain identification [121].

In this work, we learn a model from simulation, and perform trajectory optimization using that learned model. One reason we do so is the potential to learn a model using data from reality either in place of, or in addition to, simulation data. A reasonable alternative would be to use the simulation directly for trajectory optimization. Our initial attempts to do so using Pybullet [40] failed, which we attribute to the difficulty in using finite differences to compute dynamics linearizations while frequently making and breaking contacts. Further, we found it computationally less expensive to compute batches of trajectories in parallel with neural networks on GPUs than with parallel physics simulations. But, this may be possible using a differentiable physics simulator [26], or using gradient-free TrajOpt [170] in tandem with physics simulations that run in parallel on GPUs [103]. To account for sim-to-real transfer, the resulting policies could be used to gather real-world data from a robot, and that data used to learn an error correction term to create a model that is hybrid of a simulator and a neural network [2], potentially rivaling our current sample efficiency by learning only terms defining how reality differs from the simulation.

We presented generalization of the policy to designs not seen during training; however, we noticed that the worst-case from those designs, while still moving in the right direction on average, performed worse than the lowest-performing design from the training set under our evaluation metric. One way to address this would be to include some bilaterally asymmetric designs in the training set, such that the policy learns to coordinate limbs in asymmetric designs. However, such an approach would likely not be scalable in the general case, as even our small set of components can be used to form over 100 designs. In future work, we plan to scale the method up to larger design spaces by sampling designs at each training iteration, rather than using every design in a fixed set at each iteration. Further, we recognize that not all designs have the physical capability to move effectively, and so we intend to interleave design optimization with policy training to simultaneously identify high-performing designs and create their policies.

Chapter 4

Learning modular visual-motor policies

Modular robots have the potential to be customized to each new environment. For instance, a robot that will locomote on flat ground can be constructed with wheels, but if it will need to climb stairs, it can be re-constructed with legs. To control modular robots effectively, the controller should consider both the design and the environment. However, there can be a large number of designs one could construct with the modules, which may need to operate in many different environments. As such, it is inefficient to create a new control policy from scratch for each new design and environment. Building on the methods of Ch. 3, in this chapter we develop a modular visual-motor policy that trains with and applies to both multiple robot designs and multiple environments. The policy can learn from designs and environments seen previously, and share knowledge among robots with different designs. We train the policy with a set of three designs and two environments, then show that the policy generalizes to new designs and new environments similar to those seen in training.

Chapter 3 showed that given an appropriately "modular" learning architectures and algorithms, a single policy can be trained using reinforcement learning (RL) to control multiple robot designs and generalize to new designs. That is, for a given set of modules (e.g. leg, wheel, body), assume a large number of "feasible designs" D_{feas} can be constructed. We can train a policy π for "training designs" $D_{\text{train}} \subset D_{\text{feas}}$, where $|D_{\text{train}}| < |D_{\text{feas}}|$ so that the policy can generalize to D_{feas}. However, the policies presented in Chapter 3 operate in flat obstacle-free environments, using only proprioceptive sensor inputs.

This chapter alters our previous methods to produce modular visual-motor policy that can control multiple robot designs to locomote through multiple environments, and to generalize to both new designs and environments. We do so by augmenting the policy learning algorithm of Chap. 3 to include visual inputs drawn from three-dimensional terrains. Once trained, the policy can generalize to both new designs and environments. We test how training jointly with different combinations of designs and environments impacts generalizability. Finally, we demonstrate the policy both in simulation and on real robots.

4.1 MBRL with exteroceptive inputs

4.1.1 Problem formulation

We assume training designs D_{train} are first chosen from the feasible designs D_{feas}[1]. The objective we optimize maximizes the reward a policy would accumulate when applied to multiple designs and environments,

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{e\sim\mathcal{E}} \left[\overbrace{\frac{1}{|D_{\text{train}}|} \sum_{d\in D_{\text{train}}} \underbrace{\mathbb{E}_{a\sim\pi} \sum_{t=1}^{T} r(s_t, a_t, d)}_{\substack{\text{Expected reward} \\ \text{for each design}}}}^{\text{Average over designs}} \right] \tag{4.1}$$

$$\text{s.t.} \quad s_{t+1} = f(s_t, a_t, e, d) \quad \text{Dynamics} \tag{4.2}$$

$$a \sim \pi(a|o, \theta, d) \quad \text{Policy} \tag{4.3}$$

$$o \sim O(s) \quad \text{Observation function} \tag{4.4}$$

$$s_o \sim p(s) \quad \text{Initial state distribution} \tag{4.5}$$

[1]Here, training designs are selected by the user, but future work will consider automatically selecting training designs based on design optimization, described further in Sec. 8.2.3

Here the policy π is a distribution over actions conditioned on observations and parameters θ. The policy is applied to the training designs D_{train} on a distribution of environments $e \sim \mathcal{E}$ from the space of three-dimensional environments $e \in E$. The reward $r : S \times A \times \mathcal{D} \to \mathbb{R}$ is computed over a time horizon T. The observation function O adds noise to, or removes, parts of the state vector. As in Ch. 3, our observations remove the x/y position, yaw, and the linear velocity of the body from the state, as these can be estimated by an odometry system, whereas the remainder of the state (joint positions and velocities, body orientation and angular velocity) can be readily obtained from joint and IMU sensors. We assume that a low level of Gaussian noise is added to all sensor readings, with variance tuned to approximate the noise levels of hardware sensor readings. Exteroceptive observations (e.g. from a vision system) are incorporated into the observations for the body module, making π a visual-motor policy, rather than a purely proprioceptive policy.

This problem formulation is similar to that used in the previous chapter, with the addition of marginalization over multiple environments. We approach this problem with an MBRL algorithm, which learns an approximate model $\tilde{f}_\phi$ with GNN parameters ϕ and then optimizes the policy parameters with that model as a differentiable stand-in for the true dynamics.

4.1.2 Algorithm overview

Our MBRL algorithm alternates between multiple phases of model learning, policy optimization, and data collection, within each iteration. The algorithm in this work differs from that of our prior work, and differs from related work, by including both multiple designs and multiple environments in each phase. A block diagram of the phases is shown in Fig. 4-1, and psuedocode in Algorithm 3.

In the first phase, an approximate dynamics model is trained with supervised learning from a dataset of randomly generated trajectories in the simulation environment. Next, the model is used within policy optimization, leveraging the fact that the a neural network-based model is differentiable and can be applied in large parallel batches. The policy is then applied to robots in simulation to generate more

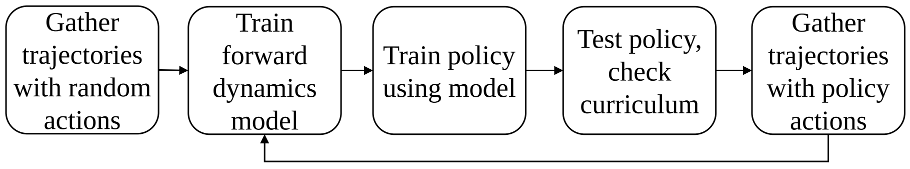

Gather trajectories with random actions
Train forward dynamics model
Train policy using model
Test policy, check curriculum
Gather trajectories with policy actions

Figure 4-1: A diagram depicting the phases of the reinforcement learning algorithm for modular visual-motor policies. Each block in this diagram is described in a subsection of Sec. 4.1, and the psuedocode of this algorithm is shown in Algorithm 3.

trajectory data. The dynamics model is retrained with the additional data, the policy re-optimized, and the process repeated. All robot data is gathered from simulation (NVIDIA IsaacGym [109]), and after training, the policy is applied to real robots. Additional details and hyperparameters for each of the learning processes that follow are described in the appendix.

4.1.3 Initial model data acquisition

The designs D_{train} are randomly initialized at states perturbed from their nominal joint positions and velocities. Initially, the environments are set to be uniformly flat. Random actions are created by sampling 10 joint actions from a normal distribution, and fitting splines to create smooth joint commands over 100 time steps, similar to the methods used in Sec. 3.3.1. These actions are applied to obtain initial trajectories stored in a dataset of trajectories $\mathcal{T}_{\text{train}}$.

4.1.4 Model learning

The dataset $\mathcal{T}_{\text{train}}$ is used for supervised model learning, in which the model $\tilde{f}_\phi$ is trained using a multi-step probabilistic loss (3.5). $\tilde{f}_\phi$ is a GNN with body, leg, and wheel module node types. The body node takes as input the state of the body (world frame orientation and velocity). In contrast to Sec. 3.3.2, the method introduced by this chapter incorporates an additional local terrain observation processed through convolutional layers, flattened, and appended to the body node input. The leg and wheel nodes take in states from their respective modules, such as joint angles and joint velocities. Each node learns to output a change in state similarly to Sec. 3.3.2

and [113, 134, 176].

4.1.5 Policy optimization

The policy is optimized to minimize objective cost (maximize reward) with respect to the model. The policy optimization method developed in this chapter differs from that of Sec. 3.3. Where the previous chapter used the model to create a dataset of trajectories, which were then used to train a policy via behavioral cloning independently from the model. This work instead optimizes the policy network parameters end-to-end with the model. The loss function to minimize derived from (4.1) is

$$\mathcal{L}_{\text{RL}} = -\mathbb{E}_{e\sim\mathcal{E}}\left[\frac{1}{|D_{\text{train}}|}\sum_{d\in D_{\text{train}}}\mathbb{E}_{a\sim\pi}\sum_{t=1}^{T} r(\tilde{s}_t, a_t, d)\right] \tag{4.6}$$

When optimizing this loss, the approximate state evolves according to the model $\tilde{s}_{t+1} = \tilde{f}_\phi(\tilde{s}_t, a_t, e, d)$, where the model is held fixed during policy optimization. The actions are sampled according to the policy, $a_t \sim \pi_\theta(a_t|o_t, d)$, $o_t \sim O(\tilde{s}_t)$. The initial state $\tilde{s}_t = s_0$ is taken from the simulator. Over the time horizon length T, the policy is applied to the approximate state, a reward is computed, and the state is advanced according to the model. The total reward over T steps is accumulated in $\mathcal{L}_{\text{RL}}$, and gradients with respect to the policy parameters are used in gradient descent. In other words, after a fully differentiable T-step roll-out of forward passes using the model and policy GNNs, policy parameter gradients are computed by back-propagating through the sequence of states and actions. This process is repeated with random mini-batches of initial states.

The reward is computed for each state and action taken in the joint model-policy roll-out. The reward function r is designed to cause a variety of robot designs to locomote forward over rough terrain. The main term in the objective rewards the distance in the $+x$ direction. Additional terms with smaller relative weights penalize roll, pitch, yaw, deviation from $y = 0$, control effort, and distance from a nominal stance. The environment (terrain height) observations are taken by sampling the simulation map height at the approximated state's world frame position. Note that the objective does not contain gait parameters such as gait speed, foot phase, or contact patterns, unlike many of the related work discussed by Chap. 2. As a consequence, the algorithm allows robots to learn locomotive behaviors that vary depending on the design and on the environment sensed.

4.1.6 On-policy model data acquisition

After the policy training phase, the policy is used to gather additional trajectories in simulation. In early iterations, the model is likely inaccurate, such that the policy optimization can exploit model bias to produce a policy that will be low-cost under the model but could be high-cost in the simulation environment. To correct model bias near states created by the policy, we apply the policy to gather trajectories. The model is fine-tuned with this new data. At the next iteration of policy optimization, the model will be more accurate near the policy, causing a virtuous cycle in which the model improves near states visited by the policy, and the policy is optimized with respect to the refined model.

The policy is applied to the simulation environment deterministically, following $x_{t+1} = f(x_t, \pi_\theta(o_t, e_t))$. The resulting states and actions are added to $\mathcal{T}_{\text{train}}$. The policy can also be applied to the simulation environment with time-correlated noise. That is, the policy is applied to the model for T steps, $x_{t+1} = \tilde{f}_\phi(x_t, \pi_\theta(o_t, e_t))$, to obtain control actions $u_{1:T}$. These control actions are perturbed with time correlated noise similarly to the noise generated in Sec. 4.1.3, then applied to the simulation environment. Trajectories across multiple designs and environments at each iterations are added to $\mathcal{T}_{\text{train}}$. The model is then refit using supervised learning, and used for the next phase policy optimization.

4.1.7 Terrain curriculum

We created an adaptive curriculum on the terrain difficulty, inspired by recent related work [111, 132]. At the first iteration, the terrain is fully flat, presenting the easiest learning problem. The initial model and policy are trained on flat ground, and once all designs pass a threshold distance travelled, the maximum height of terrain features in all environments is incrementally increased. Then, each batch in the policy training and on-policy trajectories contains some samples from each terrain difficulty to prevent catastrophic forgetting.

4.1.8 Additional implementation details

We found that for our algorithm to succeed, a few additional training implementation details were helpful.

Firstly, where the policy in Ch. 3 was recurrent only within the GNN message-passing internal steps within a time step, the visual-motor policy is recurrent over time. The recurrent network hidden state inside each module policy node is set to zero at the start of a roll-out, and then is not reset between time steps. We found that a recurrent policy trained more reliably than a non-recurrent policy; a potential cause for this could be vanishing gradients over the time sequence in which the policy loss was computed. In order to train a recurrent policy to operate for more than T time steps (the policy optimization horizon), we apply truncated backpropogation through time [165]. Each time a batch of states are sampled as initial states for the policy rollout, half of those states come from the simulation of the previous policy iteration, and half of those states come from an "imagination" of applying the policy to the current model. The states that come from imagination are associated with a policy hidden recurrent vector, used as the initial hidden recurrent vector for those initial states. Using a policy that is recurrent over time, and not just recurrent over the internal propagation phase of each GNN forward pass, has an additional benefit. We were able to tune the number of internal propagation steps within the GNN at each time step to $N_{\text{int}} = 1$ (as described by Sec. 3.2), which means that the forward pass is slightly less computationally expensive, and has less complex gradient functions than it would if $N_{\text{int}} > 1$.

Secondly, we found that with a fixed learning rate, the policy optimization would sometimes diverge. We implemented a simple adaptive learning rate to stabilize policy optimization. At the start of policy optimization, we compute the net reward from applying the policy for T steps to a large set (e.g. 10x the batch size) of initial states. This "validation reward" acts similarly to a validation loss in a supervised learning problem. If the current validation reward becomes lower than that initial validation reward, we revert the policy parameters back to the last point when the validation reward was computed, lower the learning rate, and continue.

Thirdly, we found regulating the stochastic policy entropy (e.g. the variance of the normal distribution output by the policy) aids in balancing exploration and exploitation. The policy variance is assigned an upper and lower bound. Then, when the policy is applied to compute the RL loss, a small entropy bonus is added to prevent premature convergence to a poor local minima [66].

4.2 Generalization experiments

We first measure the modular visual-motor policy's capability to generalize along two domains: new designs and new environments.

4.2.1 Designs and environments tested

We created a set of training and test set of designs and environments.

In the following experiments, we limit the training designs D_{train} to three designs, each with four legs and two wheels, where the position of the wheels is left-right symmetric and either in the front, middle, or rear of the body. The test designs D_{test} consists of three designs, each with four legs and two wheels, but where the location of the wheels is left-right asymmetric.

We create training environments with two types of three-dimensional terrain. The first, which we label "stairs," contains regularly spaced ascending steps. The second, which we label "curbs" contains rectangular blocks with fixed width at a regular interval. The test environment consists of steps and flat regions staggered. See Fig. 4-2 for a visualization of these environments. The terrain difficulty curriculum includes multiple "levels" of difficulty, ranging from fully flat to higher obstacles, with the maximum step height increasing in 2 cm intervals. For the following experiments,

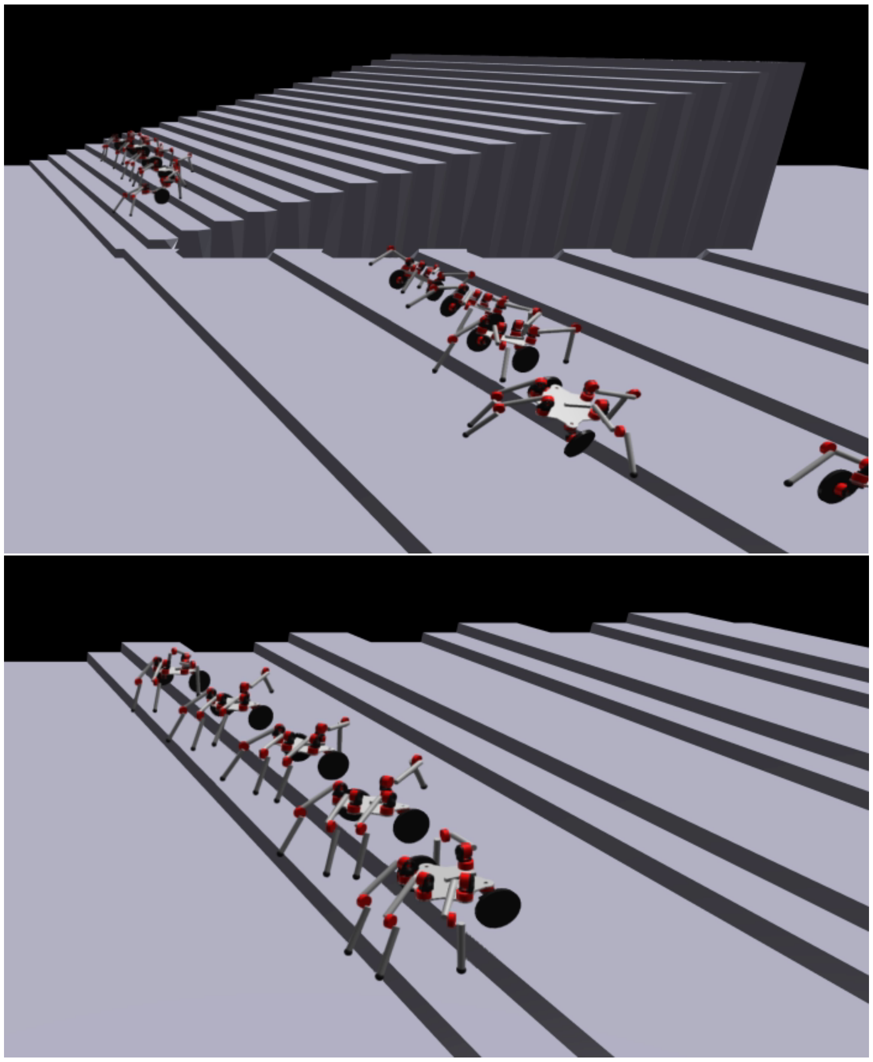


Figure 4-2: Our modular visual-motor policy is trained simultaneously using multiple designs and environments at the same time. Top: One of the designs trained to locomote through stairs and curbs. Bottom: Generalization to a new design and a new environment not seen during training. We show multiple instances of the same robot at the same time step, at states reached from different initial conditions.

we report the results of the same policy tested on a nearly-flat terrain (2 cm steps) and a more difficult terrain (10 cm steps).

Although the policy optimization objective function 4.1 includes multiple terms, we use the distance travelled forward as a metric for policy success, as the objective is dominated by this term and it is more easily interpretable than the full objective value. After training, for each design/environment combination, the policy was simulated from 10 perturbed initial conditions. The mean and standard deviation of the distance

Algorithm 3 Visual-motor MBRL for modular robots. Each step is conducted for multiple designs and environments.

```
 1: Collect dataset $\mathcal{T}_{\text{train}}$ from random action trajectories
 2: for $i = 1 \ldots N$ do
 3:     Model learning phase:
 4:     Train model $\tilde{f}_\phi$ from $\mathcal{T}_{\text{train}}$ with supervised learning
 5:     Policy optimization phase:
 6:     Initialize buffer $\mathcal{B}$ with random initial robot states and zero-valued policy
        hidden-state vectors, i.e., $\mathcal{B} \leftarrow \{s_{0,i}, h_0\}_{i=1}^{N_{\text{batch}}}$ where $h_0 = \vec{0}$
 7:     for $k = 1 \ldots K$ do
 8:         Sample a batch of initial states $s_0$ with hidden-state vectors from $\mathcal{B}$, and set
            the hidden state of the policy $\pi_\theta$
 9:         $R = 0$, $\tilde{s}_0 = s_0$
10:         Roll out policy with imagined dynamics
11:         for $t = 1 \ldots T$ do
12:             $a_t \sim \pi_\theta(a_t|O(\tilde{s}_t))$
13:             $\tilde{s}_{t+1} = \tilde{f}_\phi(\tilde{s}_t, a_t)$
14:             $R = R + r(\tilde{s}_t, a_t)$
15:         end for
16:         $\mathcal{L}_{\text{RL}} = -R$
17:         Gradient descent on policy parameters $d\mathcal{L}/d\theta$
18:         Overwrite part of $\mathcal{B}$ with intermediate states and policy hidden-state vectors,
            i.e., $\mathcal{B} \leftarrow \{s_{T/2,i}, h_{T/2}\}_{i=N_{\text{batch}}/2}^{N_{\text{batch}}}$
19:     end for
20:     Curriculum check:
21:     Increase environment difficulty for any designs that have met the performance
        threshold
22:     Data collection phase:
23:     Apply $\pi_\theta$ to collect data $\mathcal{T}_{new}$
24:     $\mathcal{T}_{train} \leftarrow \mathcal{T}_{train} \cup \mathcal{T}_{new}$
25: end for
```

travelled after 200 time steps (about 16.7 seconds) was recorded. Each policy was trained three separate times, and the results averaged for each cell. The model and policy GNN sizes and learning hyperparameters used in these experiments are listed in the Appendix.

4.2.2 Comparison to hand-crafted baseline

We developed a hand-crafted gait applicable to the various combinations of legs and wheels tested in this work. All legs are given an alternating tripod gait, with phase and position offsets assigned as if they were in a hexapod. All wheels are given differential drive/skid-steering commands, with position offsets and wheel speeds assigned as if they were on a car. Gait speeds and amplitudes were tuned to produce steps as fast and high as could be tracked by the joint velocity limits. This gait is not fully open-loop, as it reacts to steer back to face forwards based on the observed yaw angle, but it does not use other proprioceptive or exteroceptive observations. The baseline gait enables the tested designs to locomote effectively on flat ground and over small obstacles, but its performance degrades as the terrain features (steps, curbs) become larger. In the tables following, we denote the results from this gait as "tripod baseline."

4.2.3 Generalization to new designs and environments simultaneously

First we test the capability of the policy when applied to new designs, new environments, and both simultaneously. Here the policy is trained with all designs and environments jointly, then the average distance travelled by all designs recorded in Table 4.1. We found that the policy on average produced larger displacement than the baseline. In particular, the results indicate that the policy can generalize to new designs in new environments at the same time, and still produce behaviors better than our hand-crafted policy.

4.2.4 Generalization across domains

With a policy for multiple designs and environments in hand, we are able to formulate new hypotheses about the effect of data variation on policy training outcomes. We aim to build a better understanding of how robots with different designs but shared structure can learn from each other, and how robots operating in different environments can learn from each other. A common underlying assumption in modern machine learning is that learning from more a larger, more varied, or more diverse training set improves generalizability: the performance difference of a model when

	Avg. dist. traveled in 16.7 seconds (m)	
2 cm steps	Tripod baseline	Policy
Train designs and train env.	6.3	**7.0** ±0.5
Test designs and train env.	5.9	**7.0** ±0.4
Train designs and test env.	6.6	**7.1** ±0.3
Test designs and test env.	6.2	**7.3** ±0.3
10 cm steps	Tripod baseline	Policy
Train designs and train env.	2.3	**4.9** ±0.2
Test designs and train env.	3.5	**4.2** ±0.4
Train designs and test env.	2.5	**4.2** ±1.0
Test designs and test env.	3.6	**4.2** ±0.6

Table 4.1: Generalization to new designs, new environments, and both simultaneously. "Train" indicates that the design/environment was seen during training, "test" indicates that it was not. The policy mean and standard deviation are listed after the policy is trained from three times with random initial seeds.

evaluated on previously seen (training set) data and data it has never seen before (test set) [145]. For example, in supervised classification problems, adding more data results in better generalization and transfer to new inputs [29, 85, 133, 155, 156]. In reinforcement learning for locomotion or manipulation, adding more variation in environment dynamics leads to better generalization to new settings [156]. We observe that in a modular learning setting, the design and environment are two distinct domains on which data can vary.

The previous chapter showed learning a control policy from a variety of designs enables generalization to new designs. Similarly, related work demonstrated learning in multiple environments improves generalization to new environments [111]. A new question of interest posed by this work is *how diversity in one domain's training set effect generalization to the other domain's test set.* In other words, does learning from multiple designs improve generalization to new environments? And conversely does learning from different environments improve generalization to new designs? To our knowledge, this is the first investigation on how the variety of designs in training effects generalization to new environments, and vice versa.

We introduce the following related hypotheses:

- (H1) A policy will have higher performance when generalizing to new environments when it has seen multiple designs in its training set than it would if it has only seen a single design in its training set.

- (H2) A policy will have higher performance when generalizing to new designs when it has seen multiple environments in its training set than it would if it has only seen a single environment in its training set.

In other words, if these hypotheses were supported, it would indicate that training multiple designs jointly can improve generalization to new environments, and vice versa. We conducted experiments to test these hypotheses by training the policy jointly with multiple designs and environments, and with individual designs and environments.

4.2.5 The effect of design variety on environment generalization

We next test whether using multiple designs in training improves the policy's ability to generalize to new environments. Here, the policy is either trained with three designs jointly, or trained three times with an individual design each time. Then, the policies are applied to the test environment, and the results over the three designs averaged, and the results recorded in Table 4.2.

4.2.6 The effect of environment variety on design generalization

We next test whether using multiple designs in training improves the policy's ability to generalize to new environments. Here, the policy is either trained with two environments jointly, or trained two times with an individual environment each time. Then, the policies are applied to the test designs, and the results over the two environments averaged, and the results recorded in Table 4.3.

Avg. dist. traveled in 16.7 seconds (m)

	Tripod baseline	Designs ind. + Envs. ind.	Designs ind. Envs. joint	Design joint + Envs. joint	Joint learning delta (m)
Train envs. (2 cm)	6.3	7.2	7.6	7.0	-0.6
Test envs. (2 cm)	6.6	–	7.6	7.1	-0.5
Train envs. (10 cm)	2.3	6.4	6.2	4.9	-1.3
Test envs. (10 cm)	2.5	–	5.0	4.2	-0.8

Table 4.2: The effect of diversity in designs during training on generalization to new environments. In this table, all conditions use the training designs, and the average distance traveled in 16.7 seconds was measured. In the condition when designs and environments are trained individually, we do not test generalizability. A positive joint learning delta would indicate that using multiple designs in training helped improve policy performance, but a negative value indicates that using multiple designs was detrimental to policy performance.

4.2.7 Results

Though our algorithm can train a policy for multiple designs and environments, as the environments become more difficult, the policy performance decreased when additional designs and environment types were added to training. We observed what we denote the "joint learning delta:" when training with a more difficult objective (including either multiple designs or environments), the performance on the more difficult terrain decreases. On flat ground and low obstacles, the policy performance was not significantly effected by adding multiple designs or environments in training. But with larger obstacles (higher steps), the policy performance was worse when trained with multiple designs and environments than the policy was trained individually for each of those same designs or environments.

The joint learning delta most apparent from the upper right cell in Tables 4.2 and 4.3. This drawback can also be seen by comparing the results when training with both designs and environments individually, which allows the policy to specialize even further. Jointly training with multiple designs and environments will likely require larger neural network capacities, larger batch sizes, and more training time in order to reach the performance of the policy that is trained with a single design and

	Avg. dist. traveled in 16.7 seconds (m)				
	Tripod baseline	Designs ind. + Envs. ind.	Designs joint + Envs. ind.	Design joint + Envs. joint	Joint learning delta (m)
Train designs (2 cm)	6.3	7.2	7.0	7.0	0.0
Test designs (2 cm)	5.9	–	7.0	7.0	0.0
Train designs (10 cm)	2.3	6.4	5.1	4.9	-0.2
Test designs (10 cm)	3.5	–	4.6	4.2	-0.5

Table 4.3: The effect of diversity in environments during training on generalization to new designs. In this table, all conditions use the training environments, and the average distance traveled in 16.7 seconds was measured. In the condition when designs and environments are trained individually, we do not test generalizability. A positive joint learning delta would indicate that using multiple environments in training helped improve policy performance, but a negative value indicates that using multiple environments was detrimental to policy performance.

environment.

In this setting, we found that on average, *learning from multiple designs did not improve generalization to new environments, nor was the converse supported.* The results do not support Hypothesis 1: this can be seen in Table 4.2 from the "joint learning delta" column corresponding to the test environments. Nor do the results support Hypothesis 2: this can be seen in Table 4.3 from the "joint learning delta" column corresponding to the test designs.

4.3 Robot demonstrations

Simulations are doomed to succeed [23]. We next validate that our algorithm produces policies valid in the real world.

The robot chassis is comprised of a chassis with onboard power (batteries from GRIN technologies [60]), computation (an Intel NUC), sensing (a Realsense D435 and XSens IMU) and a small router. The actuators on the legs and wheels are X-series modules made by Hebi Robotics [70] and connected via ethernet cables. The onboard vision system uses VINS-Fusion [123] for odometry, and Elevation Mapping [49, 50] to produce terrain maps. The local terrain map in the form of a 21 by 21 grid of

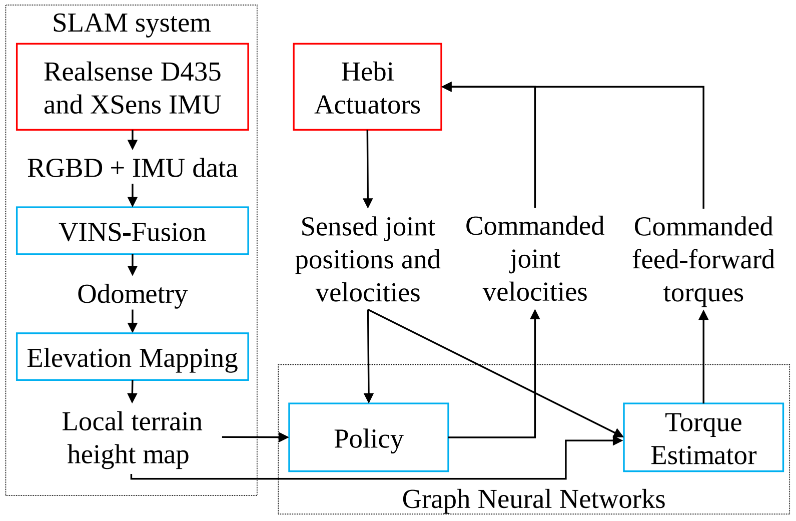


Figure 4-3: A diagram of the software interactions used in our hardware deployment. The SLAM system and policy are computed in real-time on an on-board computer (Intel NUC). In this diagram, the hardware components have red borders, software processes have blue borders, and data have no borders.

points aligned with the robot body frame are resampled from the global map, and sent as input to the policy.

The policy outputs target joint velocity commands, which are sent to the actuators, and tracked by low-level PID loops at a higher frequency on-board the actuator. The actuators perform more accurate tracking when provided with a feed-forward (FF) torque value τ, in particular when they are under heavy load. In addition to control outputs, we learn an additional torque estimation network (a GNN of similar structure to the policy GNN) that estimates the feed-forward torque needed for the actuator to track the desired velocity. The data for this output is obtained by tracking the torques experienced by the joints in simulation over multiple policy roll-outs.

Our policy was able to transfer from simulation to reality. Most notably, the policy was able to control a hexapod to climb outdoors human-scale stairs. The results of applying policies trained for a combination of curbs and stairs environments are shown in Figures 4-4 and 4-5.

In the stair-climbing demonstration, we found the limiting factor in reliability was the odometry system. The state estimate frequently diverged when the camera was obscured, which happens when the camera moved too close to the stairs, causing the map to be corrupted. Future implementations may apply different odometry systems or add additional cameras.

To test the reliability of the simulation-to-reality transfer, we conducted a repeated trials of the policy applied to a hexapod traversing a 19 cm-high curb. The robot was reset one meter from the curb, then the modular visual-motor policy applied for 20 s. We conducted five trials, and in each one, the robot was able to locomote over the curb, transitioning from a flat-ground walking motion to a climbing motion and back to flat ground. We also applied the baseline alternating tripod behavior, with the step size tuned to be high enough to step on to the curb. The alternating tripod baseline was only able to traverse the curb one out of five trials. This baseline indicates that this obstacle is difficult enough to require some behavior change away from open loop to succeed. Finally, we conducted an additional test in the same environment to determine whether the policy was using the vision system to influence its actions. In this test the policy, which was trained using local vision inputs, but at run-time was given a spoofed terrain map image consistent with walking on flat ground, effectively "blindfolding" the policy. Note this is different than policies trained by related work that do not use vision at all and learn to navigate through small terrain features proprioceptively, e.g., [94]. When the policy was not given terrain maps at run-time, it was able to locomote on flat ground, but was not able to traverse the curb in any of five trials.

4.4 Discussion

This work presents an algorithm to produce a visual-motor policy that can control multiple designs in multiple environments, and can generalize to new designs and environments. We have also shown how imitation learning can be used to transfer behaviors between designs and to accelerate training performance. Then, we formulated and tested hypotheses regarding generalizability when training jointly with multiple designs and/or environments, however the data did not support our hypothesis.

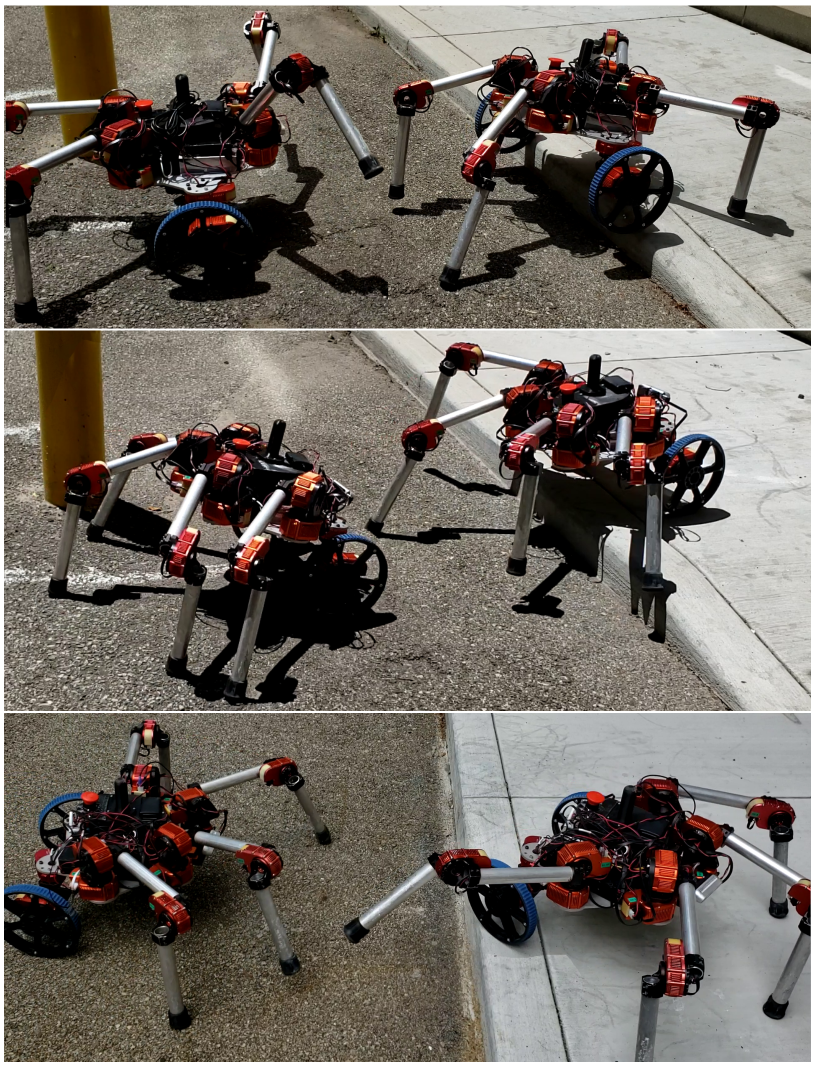


Figure 4-4: A timelapse of robots with four legs and two wheels climbing up a curb using the onboard vision system and modular visual-motor policy.

A policy trained with multiple designs and environments comes with drawbacks alongside its benefits. The policy is fit to a more complex objective, rather than specializing to a single design and environment. As a consequence, though it gains the ability to generalize, this benefit comes at a cost of decreased performance compared to a specialized policy. As the task becomes more difficult, allowing the policy to specialize to a single environment and design allowed it to perform better than when trained jointly.

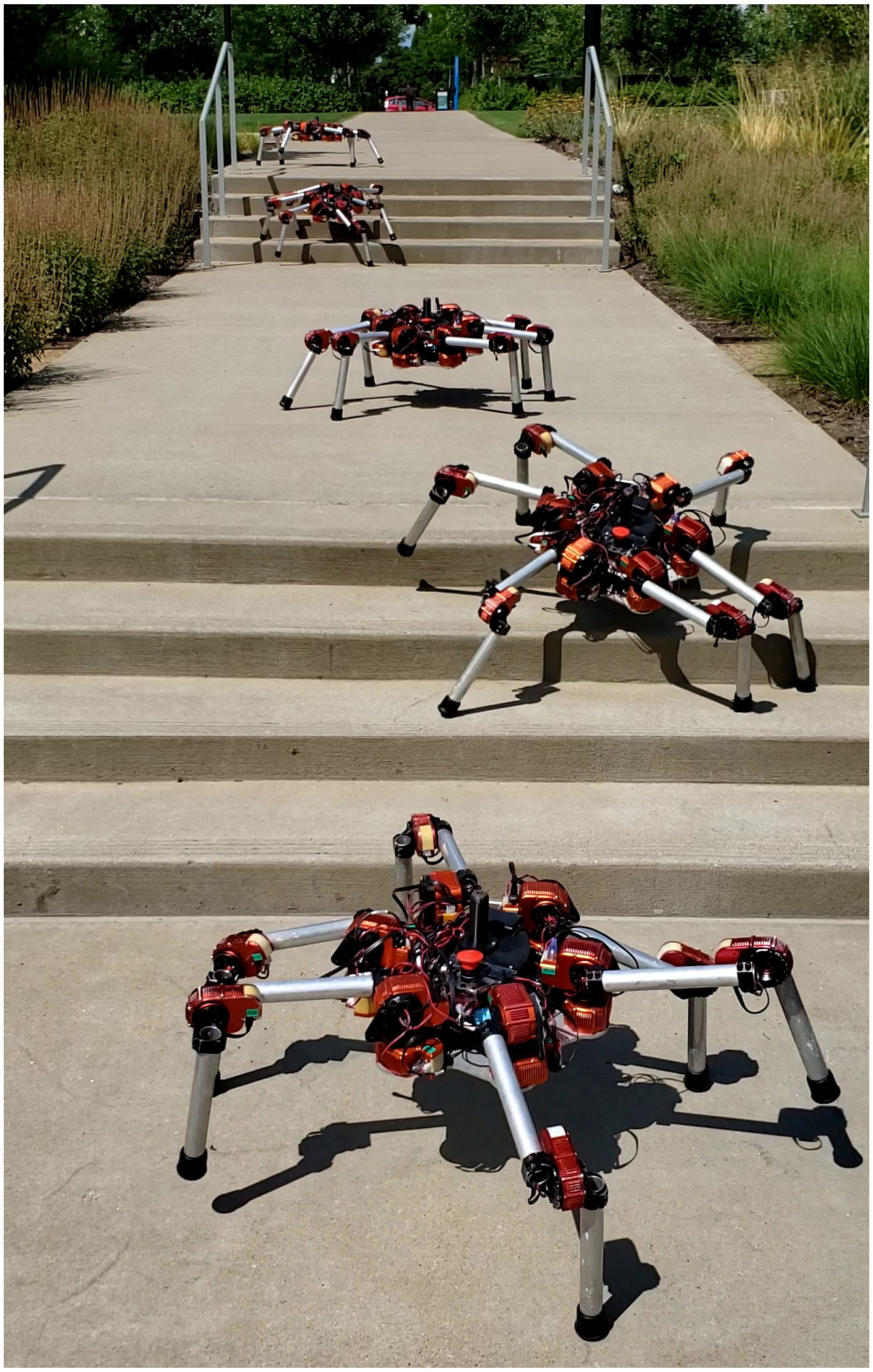


Figure 4-5: A time-lapse of a hexapod climbing up stairs using the onboard vision system and modular visual-motor policy. The steps in the stairs shown here are 14 cm high, comparable to the height of the robot body at its nominal resting stance.

4.4.1 Limitations

Our methods also come with a number of limitations.

Computational expense and training time.

Optimizing neural networks with millions of parameters requires significant compute time. Even as we continue to improve sample complexity by using more sophisticated learning representations and algorithms, at the same time aim for more complex tasks such as including more designs and environments. We used a custom GNN implementation, which does not batch well between different designs, so a better GNN package with improved batch, such as that of [4], or using optimized GNN packages, may improve training speed.

Behavior reward shaping.

The reward function plays a critical role in producing locomotion across three-dimensional terrain, and in creating policies that transfer to reality. The details of the reward weights used are listed in the Appendix, but required significant tuning to discover.

Hyperparameters.

In addition to the parameters involved in the reward, there are many hyperparameters in each of the phases of the algorithm. For example, the model learning phase has hyperparameters for GNN depth, width, activations, dropout, learning rate, batch size, number of iterations, and more, and the policy learning phase has even more. The time taken to train a single trial is only a portion of the total expense in development of novel deep learning applications, because tuning the network requires many trials before satisfactory results are achieved. Some hyperparameters are more sensitive than others– for example, the number of iterations can be increased, but increasing the learning rate can cause divergence. Similarly, if the time horizons in policy optimization are set too short, then locomotive behaviors with legs do not emerge, but if they are too long, then the approximate model trajectories become inaccurate. This is mitigated somewhat by the structure of algorithm in which learning happens in multiple phases rather than end-to-end; it is possible to isolate and tune each phase one at a time. But, each step requires expert knowledge, and can take hours of compute to test each hyperparameter combination setting.

Further, there are a large number of hyperparameters in tuning the physical robot behavior. Each joint has position, velocity, and effort PID loop gains, as well as low-pass filter values, deadzones, and a number of other parameters. These values have a significant impact on the real robot performance and change how well the policy transfers.

Limitations of MBRL.

Our method is susceptible to problems commonly associated with MBRL. Model bias, that is, the tendency for the policy to exploit inaccuracies in the model, does exist in our work. The algorithm mitigates model bias by iteratively gathering data from the simulation to train the model, but there is no guarantee that in general this will fully eliminate model bias. Further, our method does not have an explicit method for exploration. Including an entropy bonus in policy optimization allows it to discover behaviors sufficient for our tasks, but is unlikely to discover large changes in behavior. We mitigate this problem by using curriculum learning, such that the policy only ever has to discover small improvements in behavior to go from flat ground up to steep stairs. Future work will investigate techniques to explore while training.

Simulation transfer.

Our policy transfers from simulation to reality, but on more difficult environments, the transferred behaviors degrade compared to their simulated counterparts, though we have yet to quantify the sim-to-real gap for the variety of designs and environments used in this work. The success rate observed for the climbing behaviors is lower in simulation than reality. We observe that MBRL methods for "sim2real" are less well developed than are related MFRL methods. Techniques such as domain randomization [160] or domain adaptation [94] have been shown to be effective for sim2real. We included sensor noise in our observation function, and qualtitatively, this improved sim2real transfer, however, we did not yet quantify this effect.

A potentially viable alternative with MBRL would be to learn exclusively from real robot data [176], but for our application involving multiple designs and environments, the requirement to frequently change the design and environment would make data collection prohibitively expensive.

The dynamics of the real robot are different from the dynamics in simulation. The largest difference is in the dynamics of the actuators– the Hebi X-series modules are series-elastic actuated, with a rubber spring between the motor and output shaft. This elasticity is not modeled in the simulation, which treats the joints as pure torque sources.

Reactive policies.

Reactive policies, which act on recent observations and produce a single action at each time step, have a number of benefits, but also some limitations. Firstly, the policy must be specifically trained to produce single-step actions that result in long-horizon behaviors. This means that they are best coupled with a longer-term planner. Secondly, even when trained to convergence, the policy may not produce globally optimal actions at all time steps. One promising direction to combine the benefits of finite-horizon MPC and reactive policy learning such that the policy acts as a good initial seed to warm-start trajectory optimization. Such an approach has been successful for high-speed manipulation [83]

Baselines and comparisons.

This work does not compare against other learning methods. We do not claim that the policies developed in this work, when applied to an individual design and environment, would surpass those trained by methods such as PPO [132]. However, it has been our experience that adapting MFRL methods to new robots and tasks requires significant time and expertise. A direction for future work is developing comparisons to methods such as PPO [132], SMP [78], or hardware-conditioned MLPs [34]. Additional baselines, which would be expensive but informative to implement as well, would be a fully model-based (no RL) model-predictive control approach [20, 22].

The joint learning delta.

On flat ground, the previous chapter, and this chapter as well, found that a policy learning jointly with multiple designs could perform equally well as a policy trained with individual designs. Unfortunately, we did not find the same result on more difficult terrain. We found that policies trained on individual designs and environments performed better than those trained jointly on multiple designs and environments. We term this effect the "joint learning delta," caused by optimizing a more difficult/-complex objective over multiple tasks as opposed to optimizing multiple policies for multiple tasks. This effect may be mitigated in future work by increasing policy and model capacity while simultaneously increasing the amount of data, but would come at the cost of additional computation. Methods such as that of Fu et al. [52] could prove a future means of decreasing the joint learning delta as well.

Hardware limitations.

One of the downsides to modularity (mentioned in Chap. 1) is that modular robots are often heavier and less powerful than similarly sized non-modular counterparts. The speed of the robot, the size of environment features it can traverse, and the types of gaits it can use, are constrained by the power output of the actuator modules. While this is broadly true for all robots, it becomes apparent when comparing the locomotion speed of our robots to that of similarly-sized non-modular robots.

Chapter 5

Learning modular policies with imitation

Training a policy with deep reinforcement learning can be computationally expensive, especially when learning from multiple designs and environments. Robot learning algorithms, including those in our previous chapters, often assume that no prior knowledge is given and must discover behaviors from scratch [68, 71, 80, 151]. But, if demonstrations of optimal (or even just "good") trajectories are provided, that demonstration data can be leveraged in a combined reinforcement and imitation learning (IL) paradigm [121]. Imitation learning uses a dataset of expert demonstrations to learn a controller, where the experts are typically either humans [69, 130] or a model-predictive controller [115, 116]. However, in existing IL or or combination RL+IL methods, the demonstration data must be provided from the same robot to which the policy is trained. This chapter introduces a novel method of combining RL and IL to train a modular policy, where the demonstrations can come from robots with different designs than the robots being trained.

We extend the algorithm of Ch. 4 to make use of demonstrations from multiple modular designs, which bootstraps policy training for different modular designs. Specifically, we provide the learning algorithm with some prior demonstrations, in the form of a dataset of trajectories $\mathcal{T}_{\text{demo}}$ where each of the "demonstration designs" $D_{\text{demo}} \subset D_{\text{feas}}$ may have multiple trajectories in the dataset, and $|D_{\text{demo}}| << |D_{\text{feas}}|$

(e.g. $|D_{\text{demo}}| = 2$). Each demonstration trajectory consists of a sequence of states, actions, and observations. Each design $d \in D_{\text{demo}}$ has trajectories in the dataset, and this demonstration data is used to bootstrap learning for D_{train}. We show how this is still possible *even when the demonstrations are from designs that are not themselves being trained,* when $D_{\text{demo}} \cap D_{\text{train}} = \varnothing$.

This chapter introduces a novel combination imitation and reinforcement learning method that is able to incorporate demonstrations from modular robots with different designs than those being trained. Then, we conduct experiments showing how including demonstrations from different designs can accelerate learning.

5.1 Learning from modular demonstrations

The optimization problem (4.1) discovers a policy to maximize the average performance of multiple designs. It learns a behavior "from scratch" without prior knowledge that could be gained from policies known for other designs or from experts. We next turn to the question of how to incorporate such behavioral priors. The prior will be derived from a set of trajectories demonstrating behavior from designs D_{demo} that are allowed to, but not required to, be in D_{train}. Our approach combines RL and IL by adding a loss term to the policy optimization problem of Sec. 4.1, and additional steps in the training algorithm.

5.1.1 Problem formulation

A trajectory τ conssists of a sequence of states, actions, and observations for a single design. A set of L trajectories make up the demonstrations dataset for a single design index $i \in \{1 \dots |D_{\text{demo}}|\}$, and a set of demonstrations from multiple designs make up the full demonstration dataset $\mathcal{T}_{\text{demo}}$,

$$\begin{gathered} \tau = [(s_0, a_0, o_0), \dots (s_T, a_T, o_T)] \\ \mathcal{T}^i = [\tau_1^i \dots \tau_L^i], \quad \mathcal{T}_{\text{demo}} = [\mathcal{T}^1 \dots \mathcal{T}^{|D_{\text{demo}}|}] \end{gathered} \tag{5.1}$$

These demonstrations could be created via an external policy such as human teleoperation or open-loop hand-crafted behaviors.

Given these demonstrations, we derive a constraint requiring the policy to imitate the actions given the observations from the dataset. The likelihood of the policy producing individual single actions from the dataset is $\pi(a|o, \theta, d)$. The joint probability is

$$\pi(a_1|o_1, d_1) \cap \pi(a_2|o_2, d_1) \cap \ldots \pi(a_T|o_T, d_1) \cap \pi(a_1|o_1, d_2) \ldots$$

Assuming independence between samples in the dataset, we want to lower-bound this joint probability,

$$\Big[\prod_{d \in D_{\text{demo}}} \prod_{(o,a) \in \mathcal{T}_{\text{demo}}} \pi(a|o, \theta, d)\Big] > \kappa$$

where $\kappa \in \mathbb{R}^+$. Taking the logarithm of both sides maintains the inequality and converts the products to sums,

$$\Big[\sum_{d \in D_{\text{demo}}} \sum_{(o,a) \in \mathcal{T}_{\text{demo}}} \log(\pi(a|o, \theta, d))\Big] > \log(\kappa). \tag{5.2}$$

resulting in a constraint on the joint likelihood of the policy producing all of the actions from the dataset.

Converting this hard constraint into a soft constraint via a Lagrange multiplier λ leads to a loss function, optimized during the policy learning phase,

$$\mathcal{L}_{\text{IL}} = -\sum_{d \in D_{\text{demo}}} \sum_{(o,a) \in \mathcal{T}_{\text{demo}}} \log(\pi(a|o, \theta, d)) \tag{5.3}$$

$$\mathcal{L} = \mathcal{L}_{\text{RL}} + \lambda \mathcal{L}_{\text{IL}}. \tag{5.4}$$

A novel aspect of our RL+IL method is that *demonstration designs do not need to be the same as the training designs.* In other words, our method makes it possible to derive a behavioral prior from designs even when $D_{\text{demo}} \cap D_{\text{train}} = \varnothing$. As long as all designs in D_{demo} and D_{train} are constructed with combinations of modules, our method allows for robots with different designs to learn from one another.

5.1.2 Creating demonstration data

We create demonstration data for the lowest and highest degree-of-freedom designs in our feasible set: the four-wheel car (eight DoF) and the six-leg hexapod (eighteen DoF). We hand-craft behaviors based on expert knowledge for these designs. The hexapod demonstrations show alternating tripod gait trajectories, where the left and right side step sizes are modulated to steer the body yaw towards zero while walking forward. The car demonstrations show differential drive skid-steering, in which the left and right side wheel velocities are modulated to steer the body yaw towards zero while driving forward. Note these are the same behaviors used as a baseline in Chapter 4. For both the car and hexapod, we start the robot in 50 different randomly perturbed initial conditions, and create trajectories of 100 time steps showing the robot turning toward zero yaw and moving forward. These trajectories form the dataset $\mathcal{T}_{\text{demo}}$.

5.1.3 Learning from demonstration data

The demonstration data is incorporated into the MBRL algorithm to create an MBRL+IL algorithm. First, it is used to "pre-train" a policy, that is, to warm-start the policy before the algorithm in Sec. 4.1 begins. In policy pre-training, we employ behavioral cloning using the demonstration dataset. The IL loss (5.3) is used to perform supervised regression on the policy parameters, such that the policy produces similar actions to the dataset when the observations and designs in the dataset are input. The resulting policy is not effective globally– it does not know how to recover to the demonstrated gaits from arbitrary random positions. However, it serves to initialize the policy such that it does not need to discover behaviors from scratch. In this phase, policy learning only uses D_{demo}.

Then, the demonstrations $\mathcal{T}_{\text{demo}}$ are used in the loss (5.4). At each iteration in the policy optimization phase, a batch of data is sampled from $\mathcal{T}_{\text{demo}}$ and used to compute a behavioral cloning IL loss, and added to the RL loss. As a result, the policy is encouraged to reproduce the actions in the demonstrations, without being limited to those behaviors, and can still learn recovery behaviors.

5.1.4 Implementation details

The drawback to using demonstrations to bootstrap policy learning is that those demonstrations may not be optimal with respect to the objective, such that the IL loss (5.3) could act to limit the minimum RL loss (4.6) that could be achieved otherwise. To account for this possibility, we set λ in (5.4) to decay at each iteration. By decaying the IL term, we observe the policy initially imitates the demonstrations, then as the algorithm iterations between phases, the same gait style (e.g. alternating tripod-like gaits) persist even when as the behavior may diverge from the demonstrations.

Another scenario we encountered is a poor-quality local minima that can occur when the RL and IL loss conflict during early iterations. The likelihood of the policy imitating the demonstrations could be low, while still lowering the RL loss. To prevent this situation, we penalize the entropy of the actions output by the policy when given observations from the demonstration dataset, that is, an additional loss term of the form $\sum_{o \in \mathcal{T}_{\text{demo}}} H(\pi(a|o)$ where $H(\cdot)$ is the entropy function. We found this loss guided the policy towards matching the actions in the demonstration dataset, stabilizing the initial RL+IL iterations.

5.2 Experiments: RL + IL

We conducted experiments to determine the value of adding imitation learning to the MBRL algorithm. The demonstration trajectories can be derived from robots with either the same or different designs as the designs the policy is trained to control, as long as all designs involved are made up of combinations of the modules. We hypothesize that the inclusion of demonstration trajectories can improve policy convergence speed. We further hypothesize that the improvement in convergence speed will be more noticeable when the training designs are the same as the demonstration designs, but that there will still be an effect when the training and demonstration designs are disjoint sets, i.e., when the designs the policy learns to control are not shown by the demonstration dataset.

We test two cases: firstly, where demonstrations are from the training designs

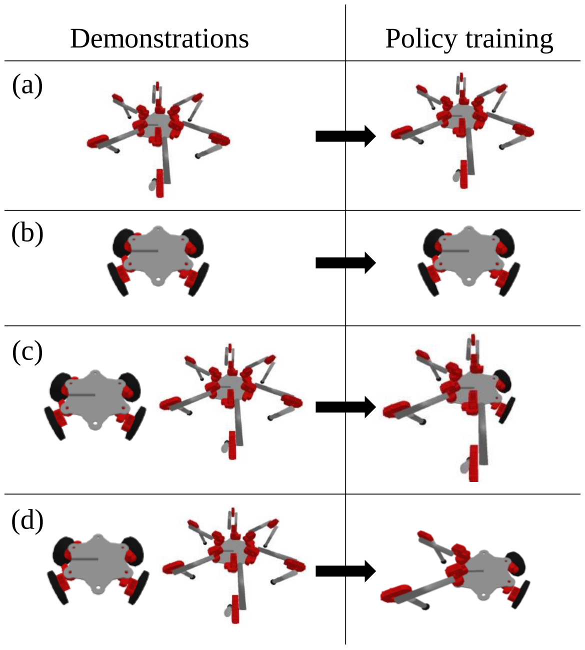


Figure 5-1: The designs used to create a demonstration dataset (left), and the designs used train a modular policy (right) with our RL+IL algorithm, in our experiments. The results of each condition (a-d) are shown in Fig. 5-2.

($D_{\text{demo}} = D_{\text{train}}$), and secondly, where demonstrations are from different designs ($D_{\text{demo}} \cap D_{\text{train}} = \varnothing$). To isolate the effect of the 3D terrain from the imitation learning, we apply these tests on flat terrain, although the method is applicable to the full visual-motor policy on 3D terrain. In each test, we run the algorithm three times, average the results, then plot the max, min, and mean of the distance travelled. The designs used in the demonstration dataset and in policy training are illustrated in Fig. 5-1.

In the first experiment, we set D_{train} to be only either the hexapod (six-leg) or car (four-wheel) design, and measure the effect of including demonstrations from those designs in training. The results of this test are shown in Figs. 5-2a and 5-2b. We found that including hexapod demonstrations improved the convergence speed of the algorithm, and that the car demonstrations had little effect on the car learning. The car is able to learn to locomote nearly optimally in a single iteration. This result suggests that when the learning task is less complex (lower dimensional, smoother), the demonstrations are not as useful, but do not impede the RL algorithm.

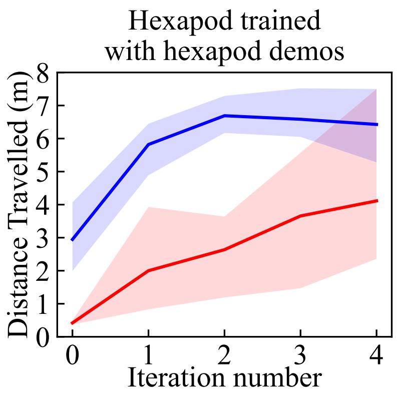
In the second experiment, we set D_{train} to be a design with four legs and two wheels, then a design with two legs and two wheels, and measure the effect of including both hexapod and car demonstrations in training. The results of this test are shown in Figs. 5-2c and 5-2d. The inclusion of demonstrations increases the speed of convergence and decreases the variance, even when the designs learning to locomote are different than the designs shown in the data.

5.2.1 Discussion

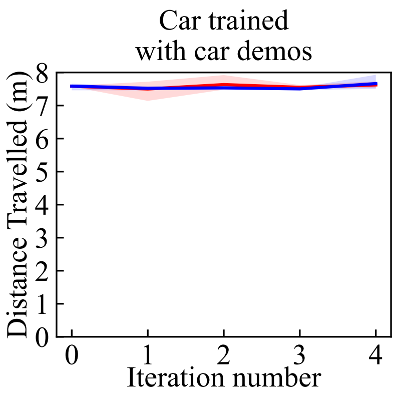
For robots to become more versatile and capable in the real world, they must be able to function in multiple environments, and they may benefit from learning from other robots that have different designs. Our results show that one policy can be used on multiple robots and environments, even outside of those seen during training. While we have focused on a narrow class of locomoting robots in this work, we believe our methods are general enough to be applied to different types of robots and settings. Our ongoing work will develop additional baseline comparisons, as well as testing other network architectures, designs, and environments.

Algorithm 4 MBRL with IL for modular robots. Each step is conducted for multiple designs and environments.

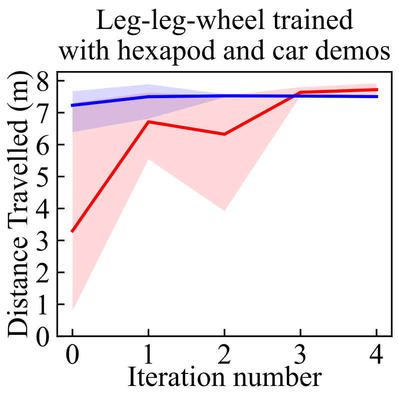
```
1: if Using prior demonstrations then
2:     Train $\pi_\theta$ with behavioral cloning from $\mathcal{T}_{\text{demo}}$
3: end if
4: Collect dataset $\mathcal{T}_{\text{train}}$ from random action trajectories
5: for $i = 1 \dots N$ do
6:     Model learning phase:
7:     Train model $\tilde{f}_\phi$ from $\mathcal{T}_{\text{train}}$ with supervised learning
8:     Policy optimization phase:
9:     Initialize buffer $\mathcal{B}$ with random initial robot states and zero-valued policy
       hidden-state vectors.
10:    for $k = 1 \dots K$ do
11:        Sample a batch of initial states $s_0$ with hidden-state vectors from $\mathcal{B}$, and set
           the hidden state of the policy $\pi_\theta$
12:        $R = 0$, $\tilde{s}_0 = s_0$
13:        Roll out policy with imagined dynamics
14:        for $t = 1 \dots T$ do
15:            $a_t \sim \pi_\theta(a_t | O(\tilde{s}_t))$
16:            $\tilde{s}_{t+1} = \tilde{f}_\phi(\tilde{s}_t, a_t)$
17:            $R = R + r(\tilde{s}_t, a_t)$
18:        end for
19:        $\mathcal{L}_{\text{RL}} = -R$
20:        if Using prior demonstrations then
21:            Sample a batch of $(o, a)$ from $\mathcal{T}_{demo}$
22:            Compute $\mathcal{L}_{\text{IL}}$ per (5.3), add to $\mathcal{L}_{\text{RL}}$
23:        end if
24:        Gradient descent on policy parameters $d\mathcal{L}/d\theta$
25:        if $k$%update rate $== 0$ then
26:            Create a batch of new random initial states, and apply the policy for
               $T_{\text{rand}} < T$ steps
27:            Add the resulting states and policy hidden-state vectors to $\mathcal{B}$
28:        end if
29:    end for
30:    Curriculum check:
31:    Increase environment difficulty for any designs that have met the performance
       threshold
32:    Data collection phase:
33:    Apply $\pi_\theta$ to collect data $\mathcal{T}_{new}$
34:    $\mathcal{T}_{train} \leftarrow \mathcal{T}_{train} \cup \mathcal{T}_{new}$
35: end for
```



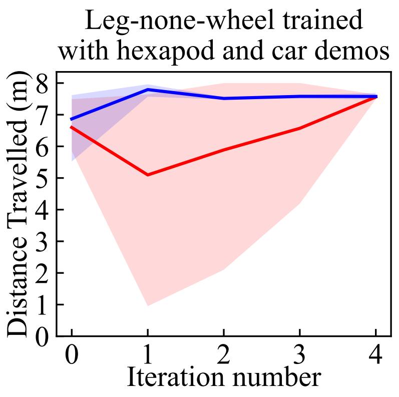
(a)



(b)



(c)



(d)

Figure 5-2: The results of the experiments described in Sec. 5.2, in which we compare the outcome of training with and without demonstration data (RL vs. RL+IL). In each plot, the blue curve corresponds to the average over three trials when training with RL+IL method, and the red curve to the RL method without IL. The shaded regions indicate the maximum and minimum value for the trials. (a) A hexapod trained using demonstrations of a hexapod walking with alternating tripod, (b) A car trained using demonstrations of a car driving with skid steering, (c) a robot with four legs and two wheels trained using demonstrations from both a hexapod and a car, and (d) a robot with two legs and two wheels trained using demonstrations from both a hexapod and a car.

Chapter 6

Automating manipulator design

Synthesizing the design of a modular robot for a given task involves a number of challenges, one being that the space of possible modular designs grows exponentially in the number of types of modules and ways they can be connected. When searching this exponentially large space, we have to evaluate whether each candidate robot can complete the task. This evaluation requires comparing the relative performance of each candidate. If the task requires complex behaviors, realistic simulation, or planning, then even evaluating a fraction of the possible designs at run-time would cause a delay in deployment, and is computationally intractable at scale. We address this intractability by learning a *design value function* (DVF), an estimate of how each module added to a design will impact the performance. After off-line training, designs can be obtained on-line through an inexpensive search using the DVF as a search heuristic.[1]

The main contribution of this chapter is an algorithm which uses deep reinforcement learning to create the DVF, enabling us to efficiently search the space of arrangements in the context of each robot's inherent capabilities for a given task. Specifically, the DVF is a state-action value function, conditioned on the task, that estimates the impact adding each module would have on robot performance. In this chapter, we limit the scope of the problem to synthesizing the arrangements of modular serial manipulators, for tasks in which the manipulator must reach a set of quasi-static

[1]This chapter is adapted from [168]

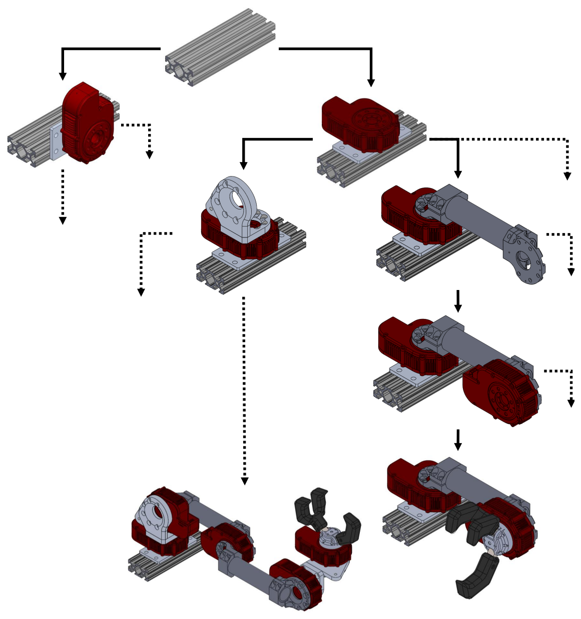


Figure 6-1: Our approach searches for modular manipulator designs by viewing the space of arrangements as a tree, where modules are sequentially added to the end of the robot. The arrangement at the root of the tree is a base mounting location. Solid arrows represent module additions, and dashed arrows indicate that the tree continues but is not shown. We use deep reinforcement learning to create a data-driven search heuristic which guides search on this tree. We apply our algorithm to modular components produced by Hebi Robotics [70].

workspace positions and orientations.

We build on prior modular design synthesis methods [44, 63] which incrementally construct and search a tree of different modular arrangements. Specifically, each node added as a child to a current leaf node represents adding a module to the distal end of the manipulator, as shown in Figure 6-1. We view the construction of this tree as a series of states (arrangements) and actions (adding modules), and treat assembly of an arrangement as a Markov Decision Process [149]. Under this formulation, we learn a state-action value function which approximates the benefit of adding each module type to an arrangement given the task. The DVF is a form of deep Q-network (DQN) which learns to approximates this value function using reinforcement learning [112]. The DVF is used within a search heuristic for a best-first graph search [18]. In the context of a branch-and-bound graph search, the DVF can be thought of as estimating the maximum performance (minimum cost) that a robot on a given branch of the search tree could have.

We compare our approach to two related methods which search for modular arrangements: a best-first search [63] and an evolutionary algorithm [82]. After training the DVF, our algorithm finds lower-cost solutions more efficiently than these related methods.

6.1 Deep Q-learning

We formulate the robot design problem as a finite Markov Decision Process, in which we construct a robot by adding one module at a time. We define a *complete* arrangement as one that ends with an end-effector module, and a *partial* arrangement as one that does not. At each time step t, the agent selects an action a_t that adds a module to a partial robot. The state s_t contains the arrangement, so the next state depends deterministically on only the previous state and the module added. This results in a new robot, s_{t+1}, and a 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 time t, $R_t = \sum_{t'=t}^{T} \gamma^{t-t'} r_{t'}$, with a discount factor $0 \leq \gamma \leq 1$. The state-action value function $Q_\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is then defined as the expected return given the action a_t is taken in state s_t following policy $\pi : \mathcal{S} \mapsto \mathcal{A}$. Our approach uses reinforcement learning to estimate the optimal state-action value function Q^*, which can be defined in terms of the Bellman equation,

$$Q^*(s_t, a_t) = \max_{\pi} \mathbb{E} \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \right]. \tag{6.1}$$

Tabular Q-learning [164], a temporal difference learning method [149], can be used to compute an estimate of the state-action value (“Q-value”) corresponding to every possible state-action pair. This approach becomes intractable for large state and action spaces, so deep Q-networks use a deep neural network as a function approximator $Q(s, a; \theta)$ with network parameters θ to approximate $Q^*(s, a)$ [112]. We train this network with experience replay [127] and a target network [158].

Our method also uses additions to the original DQN framework. Universal value function approximators (UVFA) are learned value functions conditioned on the task goal [139]. We use a UVFA to enable our DVF to apply to a range of goals, which separates our methods from those of related design RL methods for non-robot-systems such as [12, 43, 58, 135, 189]. Hindsight experience replay (HER) is a data augmentation technique employed for RL problems with sparse reward signals [10]. In HER, episodes are replayed with a different goal than the one used during the original episode.

6.2 Design value functions for manipulators

In contrast to recent work [62, 138] that used RL to solve an optimization problem to build a robot for each task, we use RL to learn a UVFA for a class of tasks [139]. Specifically we use a DQN as a UVFA to learn the expected state-action value of adding each module type to an arrangement given the goal of reaching a workspace target. The modules are chosen from a set of N_m types with indices $m \in 1, 2, ...N_m$. Each module could include any number of actuators and links, and may be able only to connect to some subset of other module types. The modular design synthesis problem is then to select a sequence of modules which form an arrangement A that can complete a given task.

In this work we limit the space of tasks to a set of N_T workspace targets which a serial manipulator should reach. A workspace target $T = [p, \hat{n}]$ consists of a position in space $p \in \mathbb{R}^3$ and tip axis orientation $\hat{n} \in \mathbb{R}^3, ||\hat{n}|| = 1$. This representation can include manipulation tasks including peg-in-hole-insertion, positioning a camera, or screw insertion.

Let $N_J(A)$ represent the number of actuated joints in a given arrangement A. The forward kinematics (FK) of A with joint angles $\vartheta \in \mathbb{R}^{N_J(A)}$

$$[p_{EE}, \hat{n}_{EE}] = \text{FK}(A, \vartheta), \tag{6.2}$$

outputs p_{EE}, the end-effector tip position, and $\hat{n}_{EE}$, the tip axis. To evaluate whether an arrangement can reach a target, we define the inverse kinematics (IK) of an arrangement as the joint angles that minimize the difference between the FK and a target,

$$\begin{aligned} \vartheta &= \mathrm{IK}(A, p, \hat{n}) \\ &= \underset{\vartheta}{\operatorname{argmin}} \, ||p - p_{EE}|| + (1 - \hat{n} \cdot \hat{n}_{EE}) \\ \text{s.t.} \quad & f(A, \vartheta) \leq 0 \end{aligned} \tag{6.3}$$

where f represents a set of constraints including self-collision avoidance, obstacle-collision avoidance, and joint limits. We use the interpenetration distance between colliding rigid bodies as the collision constraint metric. We solve IK numerically using gradient descent with multiple random initial seed restarts. In a slight abuse of notation, we will use $p_{EE}(A, p, \hat{n})$ and $\hat{n}_{EE}(A, p, \hat{n})$ to denote the forward kinematics output of the inverse kinematics solution for a given target. To evaluate whether an arrangement can reach a given target, we set tolerances ϵ_p and ϵ_n, and define a "reachability" function for the arrangement as

$$\mathrm{reach}(A, T) = \begin{cases} 1 & ||p - p_{EE}(A, p, \hat{n})|| \leq \epsilon_p \quad \text{and} \\ & 1 - \hat{n} \cdot \hat{n}_{EE}(A, p, \hat{n}) \leq \epsilon_n \\ 0 & \text{otherwise.} \end{cases} \tag{6.4}$$

Our goal is to find an arrangement of modules that is capable of reaching the targets. At the same time, we desire robots with fewer actuators (lower complexity) and lower mass. However, we must recognize that for arbitrary environments and module type sets, not every target may be reachable. Therefore, we pose this problem as a multi-objective optimization to maximize the number of targets reached while minimizing the complexity and mass of the robot, which gives us an objective function F,

$$F(A, T) = -w_J N_J(A) - w_M M(A) + \mathrm{reach}(A, T) \tag{6.5}$$

where we use $M(A)$ to represent the total mass in arrangement A, and w_J and w_M

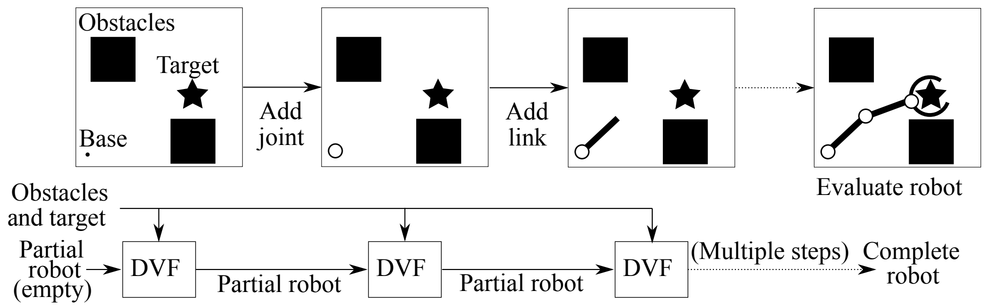


Figure 6-2: During training, the DVF is used repeatedly to evaluate the contribution each module type would have toward reaching a target. The arrangement is assembled sequentially (top) with modules selections made by the DVF (bottom)

are user-set weighting factor to trade off between the multiple objectives. We seek an arrangement that maximizes this function,

$$A^* = \underset{A}{\operatorname{argmax}} \sum_{i=1}^{N_T} F(A, T_i). \tag{6.6}$$

Next we will learn a neural network which approximates the benefit of adding each module to an arrangement to maximize (6.5) for a single target. Section 6.2.3 will describe how this function is used to maximize over multiple targets.

6.2.1 Deep Q-network for module selection

Our algorithm assembles a serial-chain manipulator one module at a time, as illustrated in Figure 6-2. We use the output of a trained DQN, which takes as input the partial design and the task, as the DVF. To use RL, we must first define the state, actions, and reward signals.

We encode the arrangement A 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 allowed in the arrangement N_{max}. At each time step an action selects a module type m from the set of N_m module types. Each episode is a series of steps where one module is added until either the arrangement is complete (an end-effector is added) or the maximum number of modules in an arrangement has been reached.

We append a single workspace target $T = [p, \hat{n}]$ to the state. This conditions

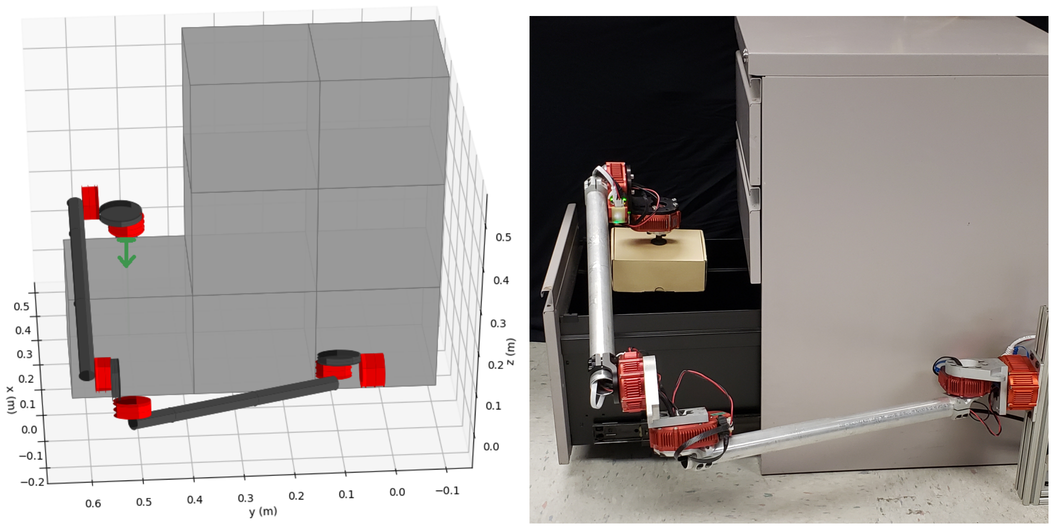


Figure 6-3: Left: An arrangement of modules (dark grey and red) with base located at the origin reaches a single workspace target position and tip axis (green point with arrow) without colliding with voxelized obstacles (grey cubes). Right: The physical modular robot matches the arrangement and environment.

the Q-values on the target, forming a UVFA that can apply to a range of targets [139]. We also condition the learned Q-value function on the locations of obstacles in the environment. To make a tractable parameterization of environment obstacles, we voxelize the space into a coarse "grid" and assign a binary occupied/unoccupied value to each voxel, so $O \in \{0,1\}^{(n_O \times n_O \times n_O)}$, where n_O is the number of voxels on each edge of the grid. The size of the voxels and the range of space over which they span were set by hand; we used $n_O = 5$ with voxel edge length 0.25 m; see Figure 6-3 for an illustration. The inputs to the DVF are the partial arrangement, the target, and the obstacle grid. Figure 6-4 depicts the structure of the neural network.

We use a reward signal such that the sum of rewards over an episode matches (6.5) because we aim to select an arrangement that maximizes that function in (6.6). The non-terminal rewards are penalties assigned for the mass and complexity of each module m added to the arrangement,

$$r(m) = -w_j N_J(m) - w_M M(m). \tag{6.7}$$

If the module added is an end-effector (EE), this is considered a terminal action,

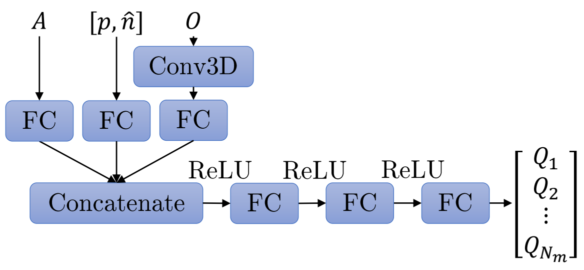


Figure 6-4: The neural network architecture we used for our DVF consists of fully connected (FC) layers with rectified linear unit (ReLU) activation, and a 3D convolution (Conv3D) over the grid of obstacles. The inputs to the DVF are the current arrangement A, target $T = [p, \hat{n}]$, and obstacle grid O. The outputs are the state-action values Q for each type of module.

and the terminal reward is returned. The reachability function (6.4) is evaluated and added to the reward. If the maximum number of modules is reached without any end-effector added, a penalty of -1 is returned,

$$r_{\text{terminal}} = \begin{cases} -1 & \text{length}(A') == N_{max} \\ & \text{and } m \text{ is not an EE} \\ \text{reach}(A', T) & m \text{ is an EE,} \end{cases} \tag{6.8}$$

where we define A' as the arrangement resulting from the addition of m to the existing arrangement A. The elements of the Q-value vector $Q \in \mathbb{R}$ output by a forward pass of the DVF represent the expected value of a module type m that could be added to the tip of the arrangement A given a target T and grid O,

$$\begin{aligned} Q(A, T, O, m) &= \mathbb{E}\big[r + \max_{m'} Q(A', T, O, m')\big] \\ &\approx \text{DVF}_m(A, T, O), \end{aligned} \tag{6.9}$$

where DVF_m is the m^{th} component of the output of the DVF, as shown in Fig. 6-4.

6.2.2 Training the DVF

The DVF is trained to approximate the Q-values of each module type for a given arrangement, target, and grid. At the start of each episode during training we randomize the target and grid. Each element of p and $\hat{n}$ is selected from a $[-1, 1]$ range, and $\hat{n}$ is normalized. When we randomize the target and environment occupancy, we ensure that any points that must be occupied by the robot (e.g. the base and target) are unoccupied.

During training we build up an arrangement by sequentially selecting modules. At each step in the episode, the network outputs Q-values for each module type. In our module set, each type of module can connect to only a subset of the other module types. We mask out invalid module connection actions, and only learn Q-values for valid actions. An episode ends when an EE module is chosen or the maximum number of allowable modules has been added. The episodes have a maximum length, enabling us to use a discount factor $\gamma = 1$. We use a Boltzmann exploration strategy [13], as there are multiple similar module choices with similar values that should be explored, such that we avoid exploiting a single robot arrangement for all tasks. We use curriculum learning [17] on the obstacle grid, mass penalty, and complexity penalty. We begin training with no obstacles or penalties, and periodically increase the maximum number of randomly selected obstacles and the penalty value during the early stages of training.

To learn from the sparse reward signal, we use HER [10]. Each time a complete arrangement is found which does not reach the target, the episode is replayed with the point that was reached set as the target. We introduce additional data augmentation by randomly sampling joint angles and occupancy grid for the robot found, calculating FK, removing any samples that are in collision, and replaying the episode with the pose reached by each sample's FK set as the target. We found this results in higher quality solutions to our full graph search procedure by training the network to better predict the potential value of lower mass/complexity arrangements. While training, we periodically test the DVF on a small set of randomly generated test points. The performance of the graph search procedure on these test sets is used as an evalution metric to decide when to end training.

6.2.3 Using the DVF to search for designs

To search for task-specific arrangements, we use the DVF module value approximator to guide a best-first search. The forward pass of the DVF outputs the Q-value for each module type conditioned on a single target and grid. This Q-value encodes the expected future value of the objective function F defined in (6.5).

Different tasks may involve reaching different numbers of targets; as per (6.6), we seek to maximize return over multiple targets. But, for a single neural network to operate on multiple points at once, the value function would need to be conditioned on all permutations of those points, and would be constrained to a fixed maximum number of points. It would be significantly more computationally expensive to train if each arrangement selection were to be conditioned on a set of targets than if it were conditioned on one target. To address this challenge, we create a search heuristic from the output of one forward pass for each target.

First we observe that at terminal actions, the state-action value summed over all targets matches the desired maximization in (6.6). That is, for actions that result in terminal states (when the selected action m is an end-effector),

$$\sum_{i=1}^{N_T} Q(A, T_i, O, m) = \sum_{i=1}^{N_T} F(A', T_i). \tag{6.10}$$

Even though this equation is not exact for non-terminal actions, we find that the summation over Q-values is a good search heuristic to maximize objective F. Therefore we form the search heuristic $h \in \mathbb{R}$ from a summation of forward passes of the DVF for each target,

$$h(A, T_1 ... T_{N_T}, O, m) = \sum_{i=1}^{N_T} \mathrm{DVF}_m(A, T_i, O). \tag{6.11}$$

This search heuristic prioritizes modules selected based on their potential to reach the targets with fewer additional modules.

Our DVF-best-first search algorithm is outlined in Algorithm 5. At each iteration, the arrangement with the highest heuristic value is popped from the open set. If it is

Algorithm 5 Manipulator arrangement search, a best-first search guided by the output of a DVF.

```
1: Input: A set of $N_T$ targets and an occupancy grid $O$
2: openset = [Empty arrangement]
3: **while** time < time limit **do**
4:     Pop node with highest $h$ value from openset
5:     Expand the node
6:     **if** node contains complete robot **then**
7:         Evaluate IK at all targets
8:         **if** all targets reached **then**
9:             Store return for the arrangement
10:        **end if**
11:    **else**
12:        Forward pass of DVF and sum output for each target as in (6.11)
13:        Add each child $A'$ to the openset with value $h$
14:    **end if**
15: **end while**
16: Return arrangement with highest return (lowest cost)
```

a complete robot, it is evaluated. Otherwise it is expanded, passed through the DVF to create new h values for its children, and those children are added to the open set.

The Q-value is the expected return from the current arrangement onward. We penalize the addition of modules, so the DVF outputs from arrangements with more modules are usually higher than the outputs from arrangements with fewer modules. As a result, the search tends to act more like a depth-first search than a breadth-first search. A neural network forward pass is computationally inexpensive, so computation of the DVF scales linearly with the number of targets, keeping computation for each node expansion low.

6.2.4 Comparisons to related work

We implemented two methods from prior work, a genetic and a best-first search, as bases of comparison. Here we describe these implementations and the experiments we ran.

Genetic algorithm

Each individual A in the population was represented with a gene $g \in [0,1)^{N_{max}}$. To convert each gene to an arrangement, each element was interpreted sequentially as the next valid module to attach. For example, if there are two possible children module types for the module at $j-1$, and element j of the gene is $0 \leq g_j < 0.5$, then the first of the two types would be selected, but if $0.5 \leq g_j < 1$ then the second of the two types would be selected. Each individual in the population was evaluated with a score combining their IK error, weighted complexity and mass, and whether they are complete. The population was resampled with elite selection, crossover, and mutation.

Best-first search algorithm

We implemented the algorithm of [63], in which the tree of possible designs is explored with a best-first search. At each step, partial robots are evaluated with a heuristic function based on an IK-like subproblem. The candidate with the lowest heuristic cost is expanded, and any complete robots are evaluated for the specified task. We removed velocity constraints from the IK and heuristic subproblem evaluations, which speeds up these functions which are evaluated many times.

Comparison tests

We conducted a comparison test between the different methods: a genetic algorithm, best-first search, and our DVF-best-first search. We used modular components produced by Hebi Robotics [70] with a set of 11 types of modules: three base mount orientations, one actuated joint, six different links/brackets, and one end-effector. We limit the maximum number of modules in an arrangement to $N_{max} = 16$, a sufficient length for complete robots with a maximum of seven actuated joints given these modules. During training and all tests, we set the objective weights $w_J = 0.025$, $w_M = 0.1$. In the comparison tests, we generated 50 sets of 10 random targets, each set with a randomized obstacle grid with up to 10 obstacles. For each method, we measured:

- the time until the first feasible arrangement (one which reaches all targets) was found for each set,
- the standard deviation of the time until the first feasible robot was found was found for each set,
- the penalty $w_J N_J(A) + w_M M(A)$ from the complexity and mass of the first feasible robot,
- the number of complete arrangements evaluated before a feasible robot was found,
- the feasible arrangement with the lowest cost found after five minutes, and
- the number of target sets for which no feasible arrangement was found after five minutes.

When no feasible arrangement was found for a given method and set within the time limit, that set was not included in the averages or times for that method. We selected these criteria because we are interested in rapid prototyping and field applications, where we may need to trade off between speed and solution quality. As such both the first arrangement found (fastest solution) and the solution found after a fixed amount of time are relevant. The IK evaluation of complete robots is the most computationally expensive step. We trained the DVF and conducted all tests on a desktop computer with Ubuntu 16.04, Intel i5 four-core processor at 3.5 GHz, and an NVIDIA GTX 1050 graphics card. We trained the DVF for 450,000 episodes (about 33 hours) before using it within our algorithm.

Searching with torque constraints

In addition to the DVF network above, we trained a network for a more difficult variant of the problem, with more module types and a constraint on the actuator torque limits. We added five more module types (four links and one rotary actuator),

Table 6.1: Results of the comparison tests described in Section 6.2.4 (lower values are better for all metrics).

	11 modules			16 modules, torque constraint	
Method	**Ours**	Best-first	Genetic	**Ours**	Genetic
Avg. runtime to first (min.)	**0.3**	3.0	0.6	**0.2**	31
Std. dev. runtime to first (min.)	**0.1**	1.0	0.7	**0.4**	1.7
Avg. num. complete robot evaluations to find first	**10**	29	138	**40**	311
Avg. cost for first found	**0.57**	0.59	0.62	**0.63**	0.64
Avg. best cost after five min.	**0.52**	0.58	0.53	**0.60**	0.63
Num. trials none found after five min.	**1/50**	16/50	1/50	**1/50**	28/50

for 16 total module types. One actuator module type had lower mass and lower maximum torque, and the other had higher mass and higher maximum torque. When evaluating the reachability function, if any actuator torque limit was exceeded, then a terminal reward of 0 was returned. As a basis of comparison, we modified the genetic algorithm to include a penalty on arrangements that overload the actuator torques. We were unable to compare this extension to the method of [63] as their method does not consider torques. The test set used in this test was the same as those described above. We trained this DVF for 700,000 episodes (about 57 hours).

6.3 Results

The results of the comparison tests are shown in Table 6.1. We found that our method produces the best results in all categories. For one of the tests, none of the three algorithms were able to find a feasible robot within five minutes.

The genetic algorithm finds costly feasible arrangements in few iterations by randomly sampling arrangements, and then refines those results over further iterations to less costly arrangements. Qualitatively we found it tends to do well when there are many feasible robots for the task, for example when there are few targets and few obstacles, because the initial sampling may include costly arrangements that complete the task. However, the genetic algorithm requires many complete robot planning evaluations. If the computational cost of evaluating planning for complete robots were to increase, we expect this method to correspondingly become more expensive.

The best-first search does not include obstacles in its search heuristic, so its performance tends to degrade in the presence of many obstacles. It evaluates robots in order of increasing complexity, but must solve an nonlinear program to evaluate each node. Due to this computationally expensive subproblem, this algorithm was not able to find solutions for a third of the test cases within the five minute time limit. In the cases where it did find a solution, it was not usually able to improve upon that solution within the remaining time.

We observed that our method acts depth-first initially, evaluating a complete robot after only a few DVF forward passes. The reward structure during training guides the search toward less costly arrangements. In contrast to the heuristic of [63], our heuristic considers obstacle locations. We found this improves average solution quality and run time over an ablated variant that did not condition the heuristic on obstacles.

In the variant with a torque constraint and additional module types, our method still searched the space of arrangements efficiently, and output feasible designs quickly, albeit after a longer training time. The higher-mass actuator module was frequently needed to create arrangements capable of extending to the farthest targets without exceeding the maximum torques, resulting in solutions with higher cost than in the previous experiments. Even with the larger set of modules and additional constraint, a feasible design was still consistently returned within one minute. In contrast, the genetic algorithm was unable to find a feasible arrangement within five minutes in the majority of the test cases.

In the most directly related work [63] the search suffers from the curse of dimensionality at runtime. When the branching factor (from number of types of modules available) increases, the number of heuristic function evaluations increases exponentially. In contrast, when more modules are added, we must train the DVF for additional time, but still use DVF forward passes to assign a heuristic to all children of the expanded node at once. Where our method is strongest, compared to related methods, is the low computation needed before finding a feasible arrangement, arising both from the computational efficiency with which the DVF is computed and in the lower number of complete robot evaluations. As the task becomes more complex, we expect that the number of complete robot motion planning evaluations will dominate the search time, resulting in decreased performance of related methods, but only increasing training time for our method.

6.4 Limitations

One limitation of our work is the need to retrain the neural network if the set of module types changes; future work will consider using a trained network to warm-start training with small differences in module set. Another limitation is that our formulation does not include costs on velocity/motion smoothness. In future work we will to move toward dynamic motion plans rather than quasi-static IK. Further, rather than rely on conventional motion planning algorithms for evaluation of each arrangement at the task, future work will involve learning control policies conditioned on the robot design, task, and environment [34] end-to-end with the module selection policy.

The module arrangement input representation in this work is a list of one-hot vectors, each vector representing a module in the sequence, and padded with zeros up to the maximum number of allowed modules in the arrangement. A limitation of this encoding is that it limits the arrangement to serial topologies. Similarly, we restricted the design to be composed of discrete selection of components. A more general, but more complex, case of robot designs composed of both continuous design parameters and discrete components is an area of ongoing research [166].

6.5 Discussion

In this chapter we presented an algorithm that uses a data-driven graph search heuristic to synthesize task-specific modular robot designs. We showed that our method returned lower-cost solutions more computationally efficiently than similar state-of-the-art methods. In the arrangement search, the “curse of dimensionality” appears from the high branching factor in the series of discrete module selection choices.

Search efficiency is needed to mitigate the computational burden of creating a motion plan for each candidate arrangement. Our method addresses these challenge by using a deep neural network forward pass to approximate the value of all options at once, moving the vast majority of the computation into off-line training. Although this chapter is focused on serial-chain fixed-base manipulators, a similar method could be applied to more complex body designs. In the next chapter, we turn to applying DVF to mobile robot designs.

Chapter 7

Automating modular mobile robot design

Conducting a combinatorial optimization process to create each new specialized design is computationally expensive, which becomes particularly important if the task changes frequently. The modular policies give us the ability to control, and therefore evaluate the performance of each design. With this performance criteria in hand, we can turn to optimizing mobile robot designs for different tasks.[1]

In this chapter, we modify the methods developed in the previous chapter to apply to mobile robot design. As in the previous chapter, we assume that the control policy is known *a priori*, but in contrast to the previous chapter which used simple quasi-static interpolation as the controller, here we use the learned policy from Chapter 3.

Our goal is to enable a user to progress from task specification, to modular robot design selection, to deployment within a short time frame. To do so, our method learns relationships among task, design, and performance far in advance of deployment, in the form of a Design Value Function (DVF). The DVF is used to identify the best design for a given task quickly at deployment. In this chapter, 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

[1]This chapter is adapted from [169]

new environments frequently. We use deep reinforcement learning to train the DVF, 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.

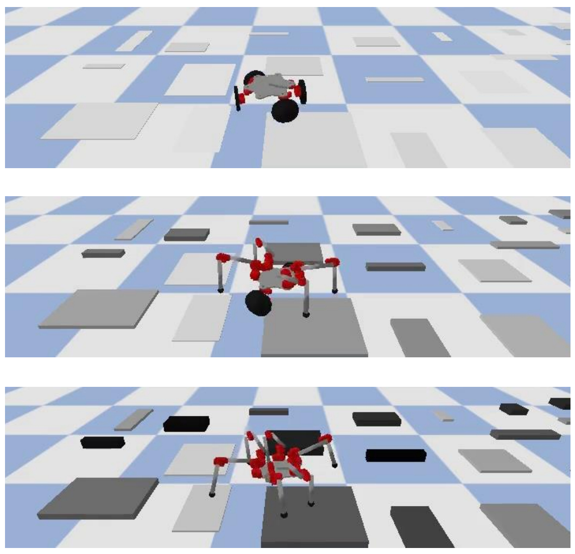
We recognize that the task/environment in reality can never exactly be replicated in simulation. Therefore, in this chapter, we require our algorithm to output multiple designs, and a ranking of their estimated performance, such that a user can physically test or choose between them.

Our long-term goal is to create a process that creates modular robot designs to complete a task in any specified environment. Chapter 6 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 7-2. 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 [149] which approximates the benefit of adding each module type given the task. We train a deep neural network to approximate this value function [112]. Completed designs are simulated, and their resulting performance is used to learn about the capabilities of each design in each environment.

We consider robots with various combinations of legs and wheels (the same as were shown in Fig. 1-4); 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.

7.1 Design value functions for mobile robots

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


(a)

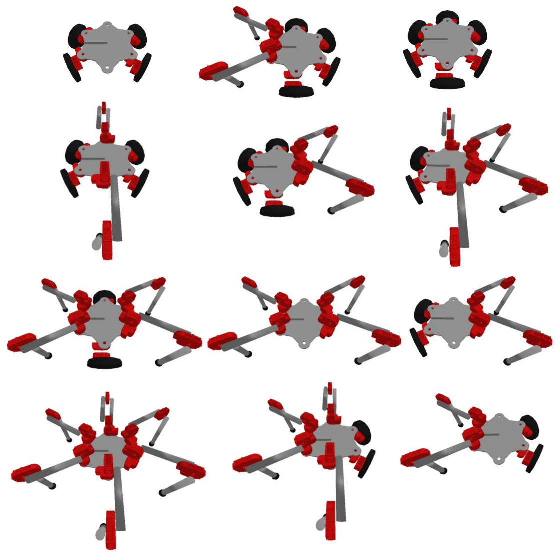
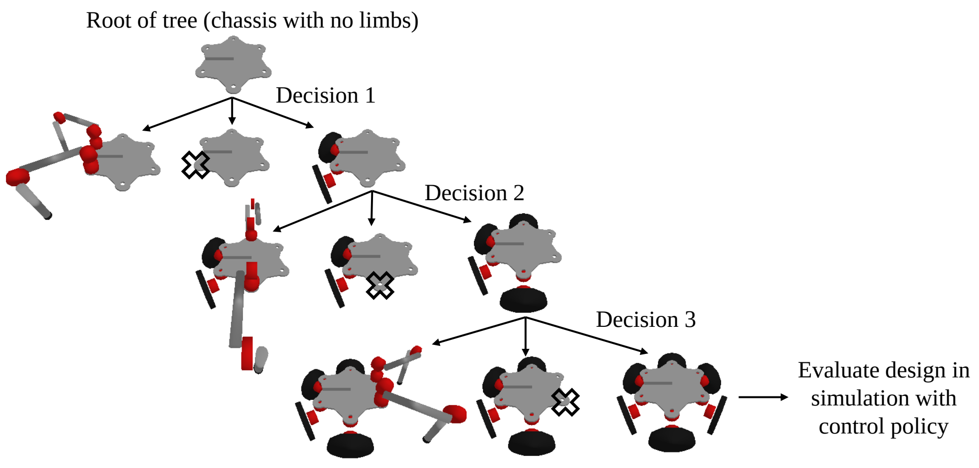

(b)

Figure 7-1: 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.

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. 7-2). 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



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

Figure 7-2: 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.

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,

$$Q^*(s_t, a_t) = \max_\pi \mathbb{E}\left[r_t + \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a')\right]. \tag{7.1}$$

Similarly to the method described in Chapter 6, the DVF is a deep neural network with network parameters θ that outputs $Q(s, a; \theta)$, trained to approximate $Q^*(s, a)$ [112]. We train this network with experience replay [127] and a target network [158]. We condition the value outputs on the task [139]: a key feature of the DVF is that it can apply to a range of tasks (in this case, environments to be traversed).

7.2 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 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,

$$\phi^* = \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim\mathbf{T}}\big[P(G_\phi(\tau), \tau)\big]. \tag{7.2}$$

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 [40] with robot models corresponding to physical hardware (Fig. 7-1) made from components produced by Hebi Robotics [70].

7.2.1 Module selection deep Q-network

Our algorithm assembles a mobile robot one module at a time, as illustrated in Figure 7-2, 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 flattened and appended to the current design, and passed through a series of fully connected layers with ReLU activation (Fig. 7-3).

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 (7.2) 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 Chapter 6. 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,

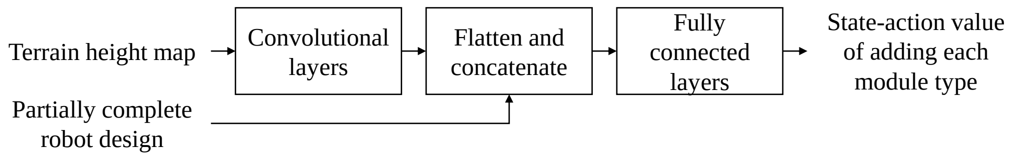


Figure 7-3: The neural network used as a DVF takes the terrain grid, passed through as series of convolutional layers (Conv2D), and the design encoding, then passed through a series of fully connected (FC) layers. The output of the network is interpreted as the state-action value of each module type that could be added to the partially complete robot design.

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. 7-1. 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.

7.2.2 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 DVF. The DVF 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 DVF, a module is added to the design. The states, actions, and rewards are stored in a replay memory buffer. We use Boltzmann exploration [13] 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.

7.2.3 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 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.

7.3 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 DVF.

The results of this experiment are summarized in Table 7.1. 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

Table 7.1: 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, in meters) 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 (distance traveled, in meters) in the three terrains.

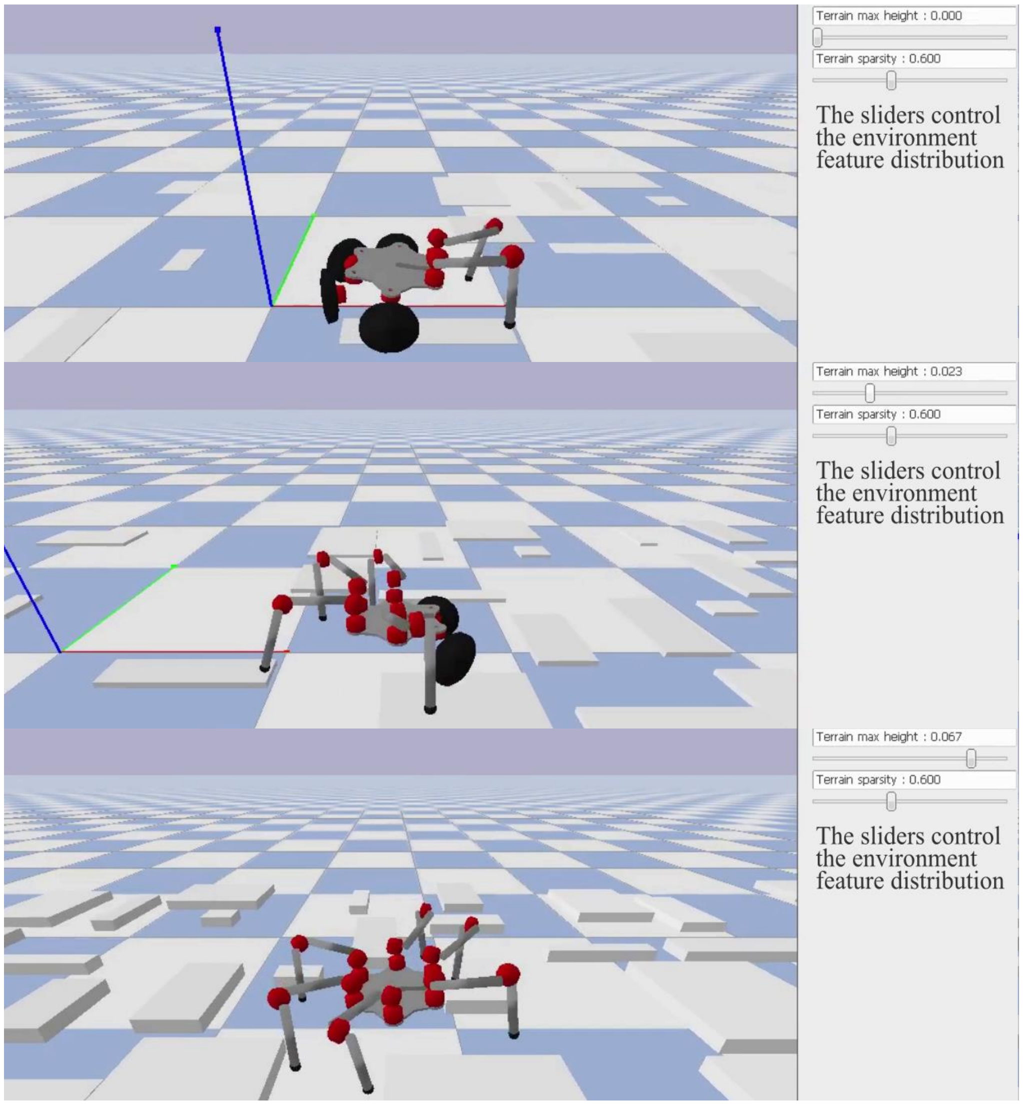
Low Terrain, 5/5 Match				Mid Terrain, 5/5 Match				High Terrain, 4/5 Match			
Estimated		Simulated		Estimated		Simulated		Estimated		Simulated	
d	P	d	P min-max	d	P	d	P min-max	d	P	d	P min-max
lwl	10.9	lww	10.7 - 11.5	lnw	4.6	lwl	7.5 - 9.5	lll	3.9	wll	4.6 - 5.2
wwl	10.3	wwl	11.0 - 11.1	lwl	4.6	lll	4.2 - 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

some terrains there are patches on which the robot may become stuck on some trials but may narrowly avoid on others.

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`, and still images from this video are shown in Fig. 7-4.

7.4 Discussion

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 control policy used here is that of Chap. 3 which 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.



Terrain max height : 0.000

Terrain sparsity : 0.600

The sliders control the environment feature distribution

Terrain max height : 0.023

Terrain sparsity : 0.600

The sliders control the environment feature distribution

Terrain max height : 0.067

Terrain sparsity : 0.600

The sliders control the environment feature distribution

Figure 7-4: Three still frames from the interactive automatic design selection GUI. As the user changes the environment, the design generator selects the robot it estimates will work best for that environment, the robot is loaded into the simulation, and the modular policy trained using the methods of Chap. 3 is applied to the robot.

Chapter 8

Discussion, Future work, and Conclusions

While one could craft a new policy from scratch for every possible modular design, such an approach is not scalable, especially given the large number of designs that can be generated from even a small set of modules. Instead, we created a modular policy framework where the policy structure is conditioned on the hardware arrangement, and trained a policy to control a variety of designs. In our policy architecture, information in the form of neural network parameters is shared not at the level of the robot but instead at the level of their modular components. The policy graph structure enables a single set of trained parameters to apply to many designs with different numbers of sensors and actuators. We showed how modular policies generalized to new robots and environments not seen during training.

Given the ability to control any design, we next turned to optimizing the design for the task. We introduced a modular design generator, learned in conjunction with the controller, that learns a value function estimating how each module would contribute toward completing the task. We used design value function to define a task-dependent distribution over designs, trained using deep reinforcement learning. After training, within the space of seconds, the generator output the designs deemed most likely to perform the task, i.e., reach waypoints in a given workspace, or locomote successfully in a given environment. Through these contributions, we showed that leveraging modularity in learning enables the creation and transfer of robot behaviors across tasks and designs.

8.1 Discussion

8.1.1 What drives the module engineering process?

Modularity in robotics can arise for multiple different of reasons. The system engineer's reason for using modularity effects the size, shapes, numbers, and types of modules created. Based on our experience, (both the authors' personal experience and based on the collective wisdom from the Biorobotics Laboratory), we create three categories of modular systems based on the methodologies used to create the modules.

Top-down modularity

We call one methodology of creating robot modules "top-down" modularity. In this form of modularity, first, a monolithic robot design is created. This monolithic design could be of any form; for instance, quadrupeds, wheeled bases, or even snake-like robots. Then, during iterations of the manual engineering design process, the engineers identify how some benefits of modularity (e.g., those listed in Sec. 1.3.1) could be applied, and convert components of the design into modules. For instance, if the actuators are the most failure prone elements, then the engineers may make the actuators modular. The actuators will be more easily interchanged, so that if one fails, the time to repair the robot can be reduced. Or, if the engineers would like to re-purpose the robot after the current project is complete, modular parts could be more easily reused for future projects, even if those projects do not require exactly the same robot.

The Biorobotics Lab has followed this route in some past projects. For example, the joints in a series-elastic-actuated snake-like robot were designed so that the length of the snake can be easily altered [128]. Then, those same actuator modules were re-used to create a series of legged robots [84, 167]. Other labs appear to have followed a similar procedure in engineering modules as well. As another example, NASA Jet Propulsion Laboratory created the components in their Robosimian robot originally for the 2015 DARPA Robotics Challenge [87]. But, since it was designed with modularity in its limbs and actuators, its components have since been re-purposed to form robots with different combinations of wheels, tracks, and arms [117, 126]. While this top-down approach to modularity is practical, it has some limitations. The modules are initially created to be used within one robot design. This means that their specifications are driven mainly by the needs of that one design and its associated tasks, potentially limiting the ability for those modules to be used to solve different tasks in the future.

Bottom-up modularity

We call a second category of modularity "bottom-up." In this form of modularity, the modules are designed with the intention of creating and reconfiguring into as many designs as possible. These usually result in a homogeneous set of generic shapes, such as cubes [180] or spheres [102, 146], which self-reconfigure into various designs. In our view, the mechanisms requirements that enable self-reconfiguration can limit the ability these modules have to form complex designs. The limited strength of the attachment mechanisms and the requirement to have multiple actuators and batteries in each module leads to designs that are weaker and heavier than designs made from manually-reconfigurable modules. More importantly, in bottom-up modularity, the tasks to be solved are not typically explicitly considered in the module engineering process. The ability to reconfigure into any arbitrary shape, while a worthwhile aspiration, does not necessarily help solve specific tasks.

Middle-out modularity

In top-down and bottom-up methods, the tasks to complete are considered only as secondary objectives in the creation of the modules. Those methodologies therefore run contrary to our objective in using modules; our goal is to re-combine the set of modules into different designs to solve a range of tasks. In practice, no single set of modules can be general enough to solve arbitrary tasks in arbitrary environments. But if instead we define a distribution of tasks, where we assume tasks will be encountered with frequency defined by that distribution, then it may be possible to select modules optimally with respect to those tasks. The modules could be designed such that for any task sampled from the distribution, the modules can be recombined into some robot design that can perform the task.

We call this view of modularity "middle-out," because the middle of the two aforementioned methodologies serves as the starting point: identifying the distribution of tasks to be solved. We can identify a few existing examples of middle-out modularity. One set of modules, which arose from the Biorobotics Lab members' experience with our modular snake and walking robots, are those produce by Hebi Robotics [70]. These modules were designed with a set of collaborative manipulation and locomotion tasks in mind, including the ability to make arms, legs, wheels, and gripping end-effectors. Another set of modules which appear to fit within the middle-out methodology are a line of modular manipulators produced by Schunk [143], which were made to solve fixed-base manipulation tasks, and have been used within research on automating modular manipulator design and control synthesis [7]. However, although these modules fit within the middle-out view, the process of specifying the composition of individual modules, or how many of each to make, is as far as we are aware primarily manual. There is, as of yet, no automated approach to deciding the contents of a set of modules given a distribution of tasks to be solved. A direction for future work that could take a step toward automating middle-out modularity is considered further in Sec. 8.2.2.

8.1.2 Connections to meta-learning

Robot modularity in this work enables the robot design to be adapted to the task at hand. We are motivated by situations where the robot needs to be deployed quickly, for example, where the task changes frequently, or where there is a cost to each minute the robot is delayed. As a result, we develop design optimization methods that search for new designs efficiently at run-time. The tasks used in Chapter 6 where manipulation tasks, where there was a distribution of obstacle placement and presence, and a distribution of end-effector goal locations. In Chapter 7 the tasks were represented by distribution of terrain obstacles to traverse.

We can draw a connection between our design search problems and meta-learning methods. Meta-learning is also known as “learning-to-learn,” “learning to optimize,” or “learning to search” [3, 5]. In a meta-learning paradigm, the machine learning model (e.g. deep neural network) gains experience over multiple learning episodes within a distribution of related tasks, and uses this experience to improve its future learning performance [75]. By learning over a distribution of tasks, the model can be used to adapt to new tasks quickly, either by directly identifying solutions, or by serving as a good initialization to be refined in a few steps. Our design generation methods are related to meta-learning methods because the design value functions are learned first for some randomly sampled tasks, then used within a search for new designs for new tasks: design value functions are learning to search. A direction for future work extending this connection is considered further in Sec. 8.2.3.

8.2 Future work

The work in this thesis lays groundwork for many additional research topics related to modular policy learning and design optimization.

8.2.1 Modular policies

Lifelong learning while adding modules and objectives

The methods of this thesis could be applied in a "lifelong learning" setting, in which the set of modules and tasks could be progressively expanded, and the same model, policy, and design value functions retrained to accommodate new tasks and modules. Modularity in learning may have a benefit in a lifelong learning setting: modular policy nodes and tasks can be added or subtracted without restarting training, and prior knowledge contained in the existing nodes may accelerate convergence. Specifically, we propose to test whether when adding additional module types, and using an existing modular policy as the initial seed, it may be computationally less expensive to warm start while adding additional GNN node types than it would be to retrain from scratch. Note that this section does not make any assumptions about how many *total* modules there are, but rather about how many *types* of modules there are in the system.

First, assume we have a modular policy with parameters θ has been trained to convergence, minimizing (3.1), with a total of M module types. In a modular policy, the parameters are divided up $\theta = [\theta_1, \dots \theta_M]$, where each θ_i is a vector of real numbers parameterizing the GNN functions used for all modules of type i. To add an additional module type, such that there would be a new total of $M + 1$ module types, we can add parameters to the policy to create a new set of parameters $\theta' = [\theta'_1, \dots \theta'_M, \theta'_{M+1}]$.

Consider two methods of initializing θ'. The first and simplest method is to randomly initialize all parameters in θ', and train the policy from scratch; the previous modular policy is erased and no prior knowledge is assumed. The second proposed method is to warm-start the new modular policy with the previous modular policy, by randomly initializing θ'_{M+1} but setting $\theta'_1 \leftarrow \theta_1, \dots \theta'_M \leftarrow \theta_M$. Similarly, if we would like to remove a module type at the same time, we can warm-start the remaining policy parameters by removing that module type's corresponding policy parameters. Here we will assume that the RL methods of Chapters 3 and 4 are applicable to training the policy. [1] Figure 8-1 illustrates these examples.

In this proposed method, parts of the previous policy, such as the behavior of the previous modules and the learned communication protocol (messages passed over graph edges), will still be applicable to the remaining modules. Re-training need only set the parameters of the new node type, and adjust the messages passed such that the new module's policy node is able to inter-operate with the existing set of modules; these adjustments may require fewer optimization steps than would training the full modular policy from a random initialization. We pose the following hypothesis: *When warm-starting the new policy parameters θ' with parts of the previous parameters θ, the number of iterations to convergence will be lower than the number of iterations to convergence when θ' is trained from random initialization.*

We could test this hypothesis via two experiments. In the first experiment, a module type would be added to the modules used in Chapters 3 and 4. The module added can be a four-DoF combination of a leg and wheel, consisting of a leg with a wheel in place of a foot. In the second experiment, a module type would be added to replace an existing module. The body module, which currently has six attachment ports, can be replaced with a smaller body with only four attachment ports and different overall dimensions. In each of these experiments, a new modular policy can be trained, both with and without warm-starting with previous policy parameters, and the number of iterations to convergence measured.

One could also alter the objective function (C in (3.1)) simultaneously with a change of modules, and warm-start training with previous policy parameters. Specifically, we propose to add a manipulator (e.g. a three-DoF arm with parallel-jaw end effector) as a module type that can be interchanged with the leg and wheel mod-

[1]When the previous policy is used to warm-start the new policy, it may be beneficial to freeze the parameters from the old policy for a few iterations, to allow the new modules to learn messages compatible with the previous modular policy functions, and prevent any initial catastrophic forgetting by the previous modules.

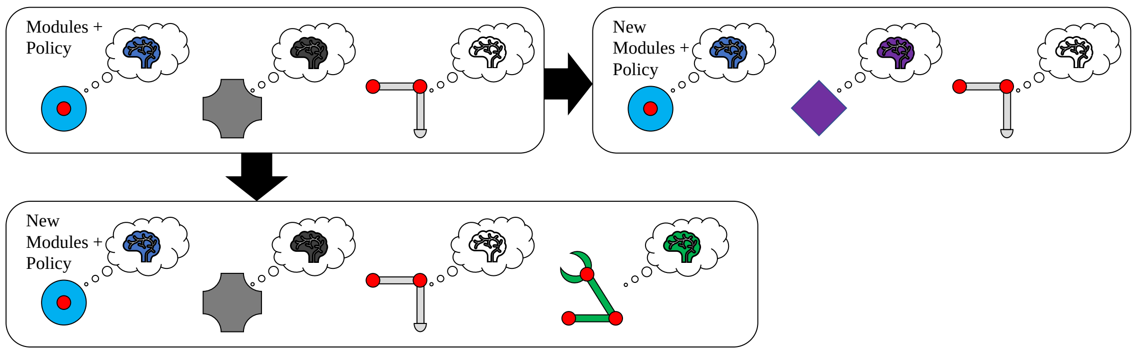


Figure 8-1: The modular policy in this thesis is trained for three module types (top). The policy could be used to warm-start training for either adding new modules (bottom), for instance adding an arm-type module. The policy could also be used to warm-start training when replacing a module type (right), for instance, changing the shape and/or number of ports on the body module.

ules. The locomotion objective can be augmented with a manipulation objective, for instance, to reach two locations in space for a "pick and place" task.

There may be drawbacks to adding more modules. For example, additional designs in the training set and more complex objective functions may require larger batch sizes, and though warm-starting may reduce the number of training iterations (gradient descent parameter optimization steps), it may come with increased computational cost of each iteration.

Learning architectures

In Chapter 3 we compared our modular policy architecture to two other policy architectures based on MLPs. However, new architectures are being discovered all the time, some of which could be used as alternatives to GNNs. For example, transformers [159] and other variants of graph networks [78] are promising candidates. Graph convolutional networks [185, 187] are promising for use as a design value function. Another possible addition to the learning architecture is to apply model ensembles [37] to capture epistemic uncertainty in the model predictions. Future work may compare the impacts of using these architectures for the models, policies, and value functions used throughout this thesis.

Developing additional modular policy training algorithms

The methods of Chapters 3 - 5 use model-based reinforcement learning. Our ongoing work aims to develop model-free methods with modular policies, leveraging recent parallelized simulation which can gain order-of-magnitude speed-ups MFRL policy learning [131]. Another alternative direction is to merge model-free and model-based learning to overcome problems of MBRL such as model bias, or to start with MBRL and then fine-tune the policy with MFRL [113].

Simulation to reality

In this thesis, we learn policies fully in simulation, then transfer them to reality. While we were able to apply our policies to real robots in this thesis, the transfer from simulation to reality was far from perfect– the robots did not behave as smoothly or travel as far in reality as they did in the simulation.

A promising direction for future work lies in using data from real robots to close the domain gap [176]. Another direction is to use real robot data to fine-tune a policy learned initially in simulation. A difficulty in using real robot data for modular robots is the need to repeatedly reconfigure the robot, which makes repeated experimentation expensive.

Another direction for future research is in adapting other simulation-to-reality approaches currently popular in model-free learning literature into situations with modular robots and model-based learning. For example, domain randomization [160] and domain adaptation [94] do not yet have direct well-established equivalents in model-based or modular learning.

Distributed learning and execution

Graph neural networks were originally introduced as a means to decentralize computation across a network of computers [137]. But in practice, we (and, to our knowledge, other related prior work on robot GNNs) conduct GNN training and inference on a single centralized computer. A potential avenue for future exploration is to decentralize the training and/or execution of the policy.

Each module in our work has an onboard micro-controller used for low-level motor control and inter-module communication. Given the low computational cost of neural network inference, these processors could be used to conduct the forward pass and message passing subroutine in a GNN as a distributed process. Distributed training may also be possible in the future. Distributed computation may have two benefits. Firstly, it would reduce the need for a centralized computer (either allowing a lighter-weight computer to be chosen, or freeing computation for other processes like SLAM). Secondly, it may make the system more robust to failure of individual modules.

Expanding model learning

The forward dynamics model approximation we learn takes in the current state and action, and outputs the estimated change in state from the current to the next time step. The model is trained with supervised learning. However, it is possible to add other predictions to the model output. For example, we experimented briefly with adding the contact state of each limb, or the distance of the center of mass location from the center of the support polygon, to the model output. Any quantity measured within the simulation could be recorded alongside the state data, and then used to train the model with additional outputs. While we did not find this approach necessary to create effective policies for our locomoting robots, the ability for the model to predict other quantities such as contact slip, or undesirable collisions, and to use those quantities in policy optimization, would allow one more easily penalizes those quantities within a differentiable reward function.

8.2.2 Design synthesis

Choosing sets of modules

In situations where the modules must be transported, for instance to a disaster response application, a large set of modules may be too bulky to transport or too costly to manufacture and maintain. Next we consider how we might choose the set of modules to minimize the total number of individual modules while still being able to construct designs that can perform different tasks from a distribution of tasks.

One of the underlying assumptions in this work is that the number of modules available is unconstrained. In other words, we assume that an arbitrary number of each type of module may be used in each robot design. This assumption may be warranted in cases where we have a sufficiently large supply of modules. For instance, with respect to the leg/wheel modules used through Chapter 7, any given robot can contain at most six legs and at most six wheels. As long as we have at least one body module, six leg modules, and six wheel modules, (a total of 13 modules) we can construct all feasible combinations of legs and wheels.

Future work may aim to select a "kit" of modules containing fewer total modules than those that would be needed to create all possible designs (e.g. for the leg-wheel example, fewer than 13 modules). The proposed methods would automatically select a finite kit of modules, where those modules can be used to construct multiple designs for multiple tasks. We will briefly review the unconstrained design optimization problem addressed in previous chapters of this thesis. Then, we will add constraints to the designs with respect to a kit, and propose a hierarchical problem to optimize the kit itself. Fig. 8-2 shows an example of how the kit selection effects the candidate and optimal designs.

The methods in this thesis optimized the parameters of ϕ of the generative model to maximize the expected robot performance criteria P over the task distribution, where elements from a space of tasks $\tau \in \mathcal{T}$ are sampled from task distribution $\mathbf{T}$,

$$\phi^* = \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{\tau \sim \mathbf{T}} \big[P(G_\phi(\tau), \tau) \big]. \tag{8.1}$$

For example, the performance P can represent the distance travelled by the design over the terrain in a fixed time span, and depends both on the design $d = G_\phi(\tau)$ and terrain. Our design value function (DVF) estimated the maximum expected performance achieved by adding each module to a partial design. The DVF served as a search heuristic to select designs, thereby acting as a generative model G.

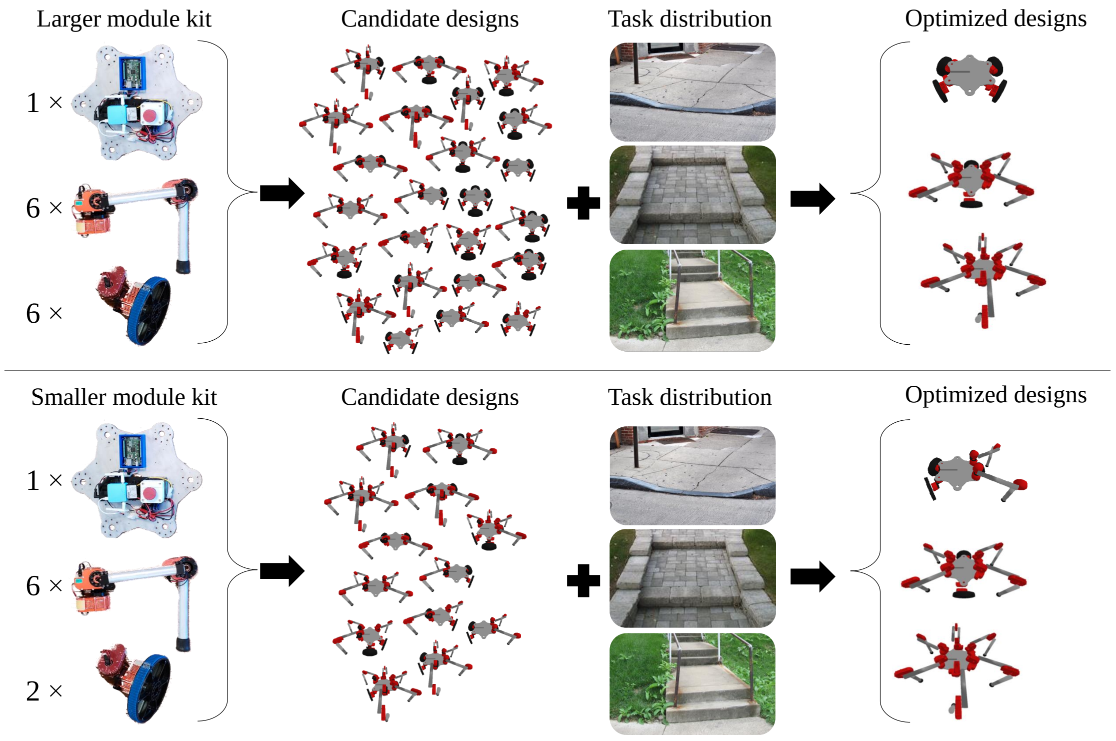


Figure 8-2: The top row shows a quantity of each module in a kit (left-most column), which can be made into many designs (second column from the left). Given tasks in the distribution (third from column the left), some of these will be optimal at a given task (right column). A kit with a large or even unlimited number of modules can be used to make a large number of designs, but not all of those designs would be optimal at any given task in the task distribution. The bottom row shows how a smaller quantity of those same modules can be made into a more limited set of designs, but nonetheless could be used to create multiple robots that can achieve the tasks from the task distribution.

The designs $d \in D$ are composed of M types of modules. Let count : $D \times \mathbb{Z} \rightarrow \mathbb{Z}$ be a function that returns the number of a given module type present within a design, i.e. for a design d and module type $i \in \{1 \ldots M\}$, if the design contains n instances of module type i within it, then $\text{count}(d, i) = n$. (8.1) does not constrain the number of each module type within a design, but since in practice we apply a maximum bound on the total number of modules N, it implicitly assumes that there is a constraint,

$$\sum_{i=1}^{M} \text{count}(G_\phi(\tau), i) \leq N \quad \forall \tau \in \mathcal{T}. \tag{8.2}$$

This constraint states that the generator can only output designs with at most N

total modules, but that there can be any number of each individual module type within the design, up to N total modules.

To constrain the number of each type of module used in a set of designs, we can parameterize a "kit" of modules $K \in \mathbb{Z}^M$ as a vector of integers representing the available number of each module type. Let K_i indicate the i-th element of this vector, corresponding to a module type $i \in \{1 \dots M\}$. For a given kit K, we can apply a constraint to (8.1),

$$\begin{aligned} \phi^* &= \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{\tau \sim \mathbf{T}}\big[P(G_\phi(\tau), \tau)\big] \\ \text{s.t.} \quad & \operatorname{count}(G_\phi(\tau), i) \leq K_i \quad \forall \tau \in \mathcal{T}, i \in \{1 \dots M\}. \end{aligned} \tag{8.3}$$

We will refer to the constraint in this problem as a *kit constraint.*

Creating a DVF that incorporates any fixed kit constraint is straightforward. While gathering data exploring different designs to train the DVF, one could treat any design that exceeds the kit constraint as an invalid design. The action leading to that invalid design is an invalid action. The Q-values output by the DVF would then reflect the limited number of modules. We will refer to such a DVF as a "constrained kit DVF," a DVF trained without a limit on the number of each module type as an "unconstrained kit DVF."

Creating a DVF that can be applied to various different kits is a slightly more challenging problem. The DVF trained on an unconstrained module set is still a valid search heuristic to search for designs under an the constraint of an arbitrary kit, but may be a poor search heuristic. One possible solution, then, is to train the DVF as per Chapter 3 or 4 with no constraint, and then apply a kit constraint during the tree search.

A more advanced method to create a DVF that can be applied to various different kits would be to condition the DVF on the kit contents. We will refer to such a DVF as a "kit-conditioned DVF." Let us define a distribution of kits $\mathbf{K}$ where kits can be sampled from the distribution, $K \sim \mathbf{K}$. One simple definition of kits would be a finite list of kits, each with uniform probability mass. Then we can pose an optimization problem to learn a generative model explicitly dependent on the kit,

$$\begin{aligned} \phi^* = \operatorname*{argmax}_{\phi} \mathbb{E}_{\tau\sim\mathbf{T},K\sim\mathbf{K}}\big[P(G_\phi(\tau|K),\tau)\big] \\ \text{s.t.} \quad \text{count}(G_\phi(\tau|K),i) \le K_i \quad \forall\tau\in\mathcal{T}, i\in\{1\dots M\}. \end{aligned} \tag{8.4}$$

We propose to address this altered problem statement via a modification to the DVF training procedure. First, the neural network representing the kit-conditioned DVF must be allowed an additional input, which would take in the kit. [2] Then, in addition to sampling tasks from the task distribution at each training episode, a kit would also be sampled from the distribution and fed into the kit-conditioned DVF. The DVF would then learn the dependence of the value of each module conditioned on how many of each module type is available.

With the notion of a kit constraint in hand, we can also ask an additional question: *how many modules of each type does one need to address the distribution of tasks?* Can we minimize the number of modules contained in the kit, without sacrificing the ability to build various designs that can perform various tasks? We can pose this as a minimization problem as well,

$$\begin{aligned} K^*,\phi^* = \operatorname*{argmax}_{K,\phi} \mathbb{E}_{\tau\sim\mathbf{T}}\big[P(G_\phi(\tau|K),\tau)\big] - \beta F(K) \\ \text{s.t.} \quad \text{count}(G_\phi(\tau|K),i) \le K_i \quad \forall\tau\in\mathcal{T}, i\in\{1\dots M\}. \end{aligned} \tag{8.5}$$

where $F : \mathbb{Z}^M \to \mathbb{R}^+$ is the cost (e.g. total mass or price) of a kit, and $\beta \in \mathbb{R}$ a relative objective weight.

(8.5) could be approached by applying a combinatorial optimization algorithm as an outer loop to the DVF training. First, we propose to train the kit-conditioned DVF to solve (8.4) over a uniform distribution of kits. Then, an algorithm such as an evolutionary algorithm, simulated annealing, or even model-free RL could be used to identify the optimal kit. It may also be possible to train the DVF and select a kit

[2] Note the representation of the kit as a list of integers may not be well-formed for neural network input. The kit vector may need to either be normalized, represented via one-hot vectors, or a different representation for the kit developed.

at the same time by iterating between kit optimization and DVF training steps. The kits sampled by the combinatorial optimization outer loop would effectively serve as a distribution of kits, instead of sampling from random kits.

Once the proposed methods are implemented to optimize the kit with respect to the task distribution, we propose to test new hypotheses about modular composition. To do so, we must define an additional metric not yet present in design synthesis research. Let us define the "kit utility" U as the function measuring the expected performance achieved by optimized designs over a distribution of tasks,

$$U(K, \mathbf{T}) = \max_{\phi} \mathbb{E}_{\tau \sim \mathbf{T}} \big[P(G_\phi(\tau | K), \tau) \big]. \tag{8.6}$$

The utility function assumes the existence of a generative model that can identify the optimal design for each task conditioned on the kit. We will define the unconstrained module set utility $U_{\max}$ as the utility of an unlimited number of modules, which will be equivalent to $U_{\max} = \max_K U(K, \mathbf{T})$. [3] If a kit K has high utility, then K can be made into designs each of which can perform some tasks optimally. Given this definition of utility, we can view the kit search (8.5) as a search for the kit that balances utility with the cost of the kit itself.

We may not need the wide variety of designs possible with an unconstrained kit in order to create different robots that complete different tasks– a kit's utility may still be high even if it is constrained. We propose the following hypothesis: *For a given task distribution $\mathbf{T}$, there exists a kit K with utility $U(K, \mathbf{T}) = U_{\max} - \delta$. δ increases as β increases, but the relationship will be non-linear, such that δ increases slowly for small values of β.* In other words, as we penalize adding modules to the kit, we will see a drop in the utility of the kit, but that drop will be small or even zero at first. This hypothesis would not be supported universally, as we can easily create counter examples: for instance, the case where every single task from the distribution requires

[3] We propose to use the above kit-conditioned DVF as the generative model in utility evaluation. It may be possible to use other methods as well, such as evolving a design for each task sampled. However, alternative methods will likely make computing the utility computationally expensive, since in practice, evaluating the utility will require sampling many tasks and optimizing a robot design for each.

a completely different design to perform that task well. However, for some systems, we may be able to reduce the number of modules while still constructing robots that can perform the tasks.

This hypothesis could be tested on the leg-wheel locomotion and fixed-base manipulation tasks defined by Chapters 6 and 7. Optimizing the contents of the kit and testing this hypothesis will lead to a better understanding of how many modules to build/purchase, or how to best pack modules for deployments.

Designing individual modules

Another underlying assumption in this work is set of modules is created *a priori* by robotics engineers. Future work may develop methods that automatically design the individual modules.

Conventionally, modules are manually designed, and robot module optimization has not been considered in existing literature. One direction extending this thesis is to optimize the structure of the individual modules to maximize their utility over a distribution of tasks. To do this, one could first train the modular policy and design generator with the most "atomic" set of design elements, i.e., joints, links, and brackets. These atomic modules can be combined to form larger interchangeable units used within multiple robots, which we term "composite" modules, i.e., arms, legs, or wheels. Fig. 8-3 illustrates this concept. In addition to a design generator, a "module generator" could be trained to estimate the optimal set of composite modules. The module generator would learn a value function estimating how each atomic element, if included within a composite module, can contribute to the module kit's utility across the task distribution.

Learning distributions of designs

In Chap. 7, we investigated using a design value function to output not only the estimate of the optimal design, but by sampling stochastically from the value function, to estimate additional near-optimal designs. However, this method is not guaranteed to produce an accurate ranking of the top design options, and the objective in the

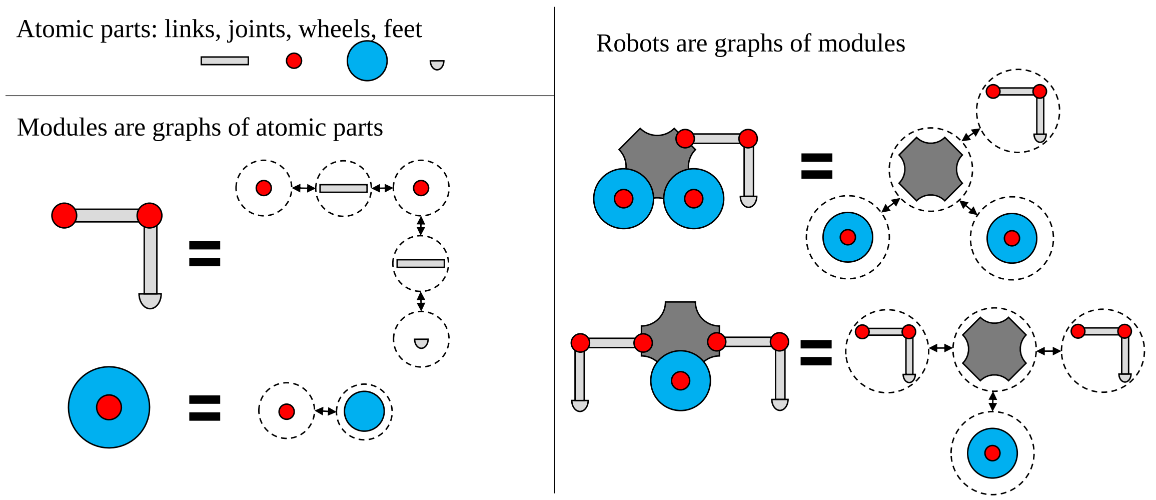

Atomic parts: links, joints, wheels, feet
Modules are graphs of atomic parts
Robots are graphs of modules

Figure 8-3: Small design components, which do not necessarily have the computational infrastructure on-board to act as intelligent modules, can be considered “atomic parts” (top left). Those atomic parts can be assembled in a graph to form “composite modules” (bottom left). Robots (right) can be viewed as graphs of composite modules. This abstraction, combined with the design search methods in this thesis, could be used by future work to develop methods that automatically design the individual modules for a distribution of tasks.

RL (Q-learning) methods is not designed to produce solutions other than a single optimum.

An alternative method to design synthesis is to formulate objectives and algorithms that produce multi-modal solutions, e.g. producing the top-k solutions, or the multiple solutions on a Pareto front of multiple objectives. To do so, we have begun to develop methods in recent publications [76, 77] which merge methods from evolutionary algorithms and generative adversarial networks, and provide an alternate branch of design synthesis methods from our RL-based methods.

Including both continuous and discrete design components

Our past work [166] presented a gradient-descent-based method for optimizing link lengths and the number of joints in manipulators. One possible direction would use combined discrete-continuous reinforcement learning methods like those of Neunert et al. [114], and adapt design generators to output both discrete componetns (joints, links) and continuous variables associated with them (link lengths, nominal offsets).

This is a challenging problem in part because each module may have a different number and dimension of continuous parameters, which may require a hierarchical design value function architecture. Another potential avenue is in combining discrete component selection using RL with continuous parameter optimization with gradient-descent.

8.2.3 Co-optimization

In this thesis we first created a control policy for many different designs, then applied that control policy to evaluate candidate designs for different tasks. Another direction for future work is in how to co-optimize the designs and control, e.g. a single training process that could result in both the design value function (DVF) and modular policies. Fig. 8-4 illustrates this concept. Design and control co-optimization has been investigated for some robot classes [62, 108, 138, 163], but not specifically for modular robots, not yet using articulated realistic robot models, and not incorporating a distribution of tasks.

Single-task co-optimization

Prior work on joint design and control co-optimization (e.g. [62, 108, 138, 163]) address the problem of co-adapting the design and policy, with a problem formulation similar to

$$d^*, \pi^* = \underset{d,\pi}{\operatorname{argmax}}\, P(\pi, d, \tau) \tag{8.7}$$

where we have left the details out of the optimization problem by leaving out the definitions of states, actions, dynamics, rewards, and the policy representation, and have only left the design d, policy π, task τ, and performance $P \in \mathcal{R}$ (sum of rewards over an episode). The output from solving this type of problem is a single design and policy that performs a task. In our view, the largest limitation of this type of method is that if the task changes, a new optimization problem must be posed and solved. We propose to depart from prior work by incorporating the ideas introduced in this thesis: considering a distribution of tasks in the optimization problem, then using a design value function to learn a mapping from task to design jointly with a modular policy.

Task-distribution co-optimization

Including a distribution of tasks $\mathbf{T}$, the optimization problem becomes,

$$\phi^*, \pi^* = \underset{\phi,\pi}{\operatorname{argmax}} \, \mathbb{E}_{\tau \sim \mathbf{T}} P(\pi, G_\phi(\tau), \tau) \tag{8.8}$$

where the generative model $G : \mathcal{T} \to D$ with parameters ϕ identifies the optimal design for an input task. This problem no longer searches for a single design– similar to the methods introduced in Chapter 6 and 7, we propose to learn a generative model that can output designs for different tasks. Note that in the edge case where the task distribution includes only a single task, then the design generator need only ever output one design, and the problem reduces to (8.7). However, in general there may be multiple different designs that are optimal with respect to performing different tasks. In this optimization problem, the modular policy must learn to control multiple designs, but only as many as are needed to perform the tasks in the distribution. Therefore, the policy will be allowed to specialize more than it would when trained with a larger set of designs, as the policy would not need to apply to designs that are not optimal for any particular task $\tau \sim \mathbf{T}$. Where this thesis thus far has assumed that the modular policy is trained with a hand-selected set of designs, the proposed method would allow the training set to be chosen dynamically using the DVF.

To solve 8.8 we propose an alternating maximization approach using steps from modular policy and DVF learning methods. Algorithm 6 describes the proposed method in pseudocode. We propose an algorithm leveraging the learning methods used in this thesis, however, it may also be possible to apply different policy learning methods (e.g. [78]) or different design generative models (e.g. [77]) as well with the same algorithm structure.

The proposed algorithm is initialized with a randomly-seeded DVF and modular policy. To begin training the policy and DVF, first, a random set of many designs and tasks are sampled, and the policy training algorithm applied. The performance (net return over an episode of the design/task/policy combination) is collected, and used to train the DVF.

The DVF would then be used to select a new set of designs. A new set of tasks will be randomly sampled, and the DVF applied to search for the estimated optimal design for each of these tasks. This set of designs and tasks would serve as the training set for the next iteration of policy training. The alternating maximization of policy and DVF training will proceed until either a computation/time limit is reached, or convergence is reached. Convergence can be determined by the point at which the estimates made by the DVF consistently match the true simulated performance of the different design/task combinations sampled. Then, at run-time when new tasks are sampled online, the DVF can be used to efficiently output a design, and the modular policy immediately applied to control that design.

The convergence speed and stability of the proposed algorithm may be limited by the fact that the DVF is effectively "chasing a moving target," because the performance of the various design/task combinations will change as the policy is trained. A proposed variation on this algorithm would address this problem by adding an additional learned function operating jointly with the DVF. We propose to add a *performance increase estimator* (PIE). The PIE consists of a learned function structured and trained similarly to the DVF: a deep Q-network that outputs an estimated decision tree branch value for each module added to a design. However, where the DVF estimates the maximum performance of a design/task combination, the PIE will estimate how much the performance of that design *would increase* given K iterations of policy training. The PIE could then be used in conjunction with the DVF to select the designs used during policy training. The DVF selects designs based on how well those designs would perform under the current (potentially unconverged) policy, but the PIE selects designs based on how much we expect the performance of the policy to improve with respect to those designs.

To further motivate the proposed PIE, consider the case where the design set consists only of two designs: a four-wheel car and six-leg hexapod, which share the same modular policy, and where the task distribution consists only of two terrains to be traversed: one flat and one with stairs. We found in our modular policy training in Ch. 3 that the car reaches maximum performance (driving forward at maximum wheels speed) within many fewer iterations than the hexapod, but that the hexapod can achieve a higher performance on stairs after additional training iterations. When the policy is not yet converged, any data collected from applying these two designs to the two tasks would indicate that the car has a higher performance on both tasks, because the hexapod has not yet learned to walk. As a consequence, if the DVF is trained using this data, the DVF would estimate that the car is optimal for both tasks, and would learn not to select it. However, the performance of the car would not improve over iterations beyond the first, because it will not be able to climb steep stair no matter how many iterations the policy is trained. The car will have a low performance improvement, which could be learned by the PIE. In contrast, the hexapod performance will increase with each policy training iteration. The hexapod will have a high performance improvement initially, and the PIE outputs should converge to zero for all designs over multiple iterations as the policy converges.

We propose to collect train the PIE to estimate how much the policy improved at each iteration (note that change in performance between iterations is a form of meta-data). Then, a combination of the DVF and PIE would be used to select designs to use as the next iteration of policy, for instance, selecting most of the batch with the PIE in early iterations and most of the batch with the DVF in later iterations. The proposed PIE is a form of active learning for exploration in the design space, related to the "curiosity" functions learned by [6, 25]. We hypothesize that the DVF will more accurately identify the optimal designs, and the policy will be higher-performing with respect to those designs, when the dynamically-changing nature of the policy performance is taken into account during co-optimization. This hypothesis could be test with respect to the mobile robot system described by Ch. 7.

An additional variation of the proposed algorithm would conduct the policy and DVF training and data collection processes asynchronously in multiple threads. While this parallelization may not reduce the overall computation budget, it could decrease

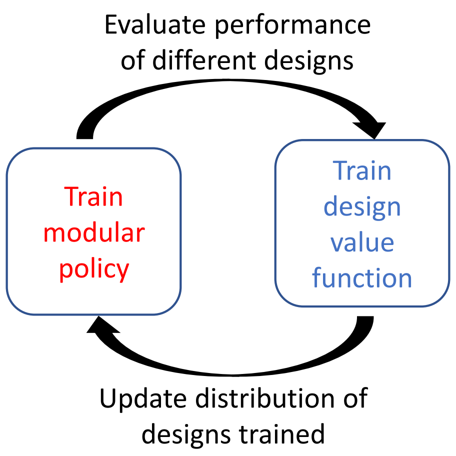

Evaluate performance of different designs
Train modular policy
Train design value function
Update distribution of designs trained

Figure 8-4: A future extension of this work could integrate modular policies with design value functions in a new formulation of design and control co-optimization.

wall clock training time.

Co-optimization as meta-learning

If design and control co-optimization (8.8) is posed as a combination of learning a modular policy and learning a function that searches for designs, the problem can be viewed as a form of *meta-learning*, i.e. "learning to learn" or "learning to optimize" [3, 5], in which one learned function learns to compose another [27]. A "meta-policy" is a policy operating on the structure of other learned function components, which assembles a policy that can perform a task [27].

In the proposed work, G will serve as the meta-policy that recombines the design/policy for different tasks, and guides the designs on which the policy parameters are trained. This would make the proposed algorithm a form of meta-learning, but with a distinct difference to prior work: the modules being assembled are not purely software abstractions, but are tied also to hardware. We propose to call this *embodied meta-learning:* the generator learns to assemble modules, where each module has both a software component (a part of the policy) and hardware. The design generator learns to choose both the physical structure and the policy structure.

Algorithm 6 Proposed (future work) algorithm for joint co-optimization of DVF and modular policies.

```
Require: Task distribution $\mathbf{T}$, design space $D$. Hyperparameters include batch size
    $N$, policy training iterations $K$, batch ratio reduction $\beta \in [0, 1]$. Performance
    function $P$, e.g., sum of returns over an episode.
 1: Initialize design value function (DVF), performance increase estimator (PIE) and
    modular policy.
 2: $\alpha = 1/2$ (Relative DVF/PIE batch size factor)
 3: while Not converged: do
 4:     Sample tasks $\{\tau\}_{i=1}^{N}$, $\tau_i \sim \mathbf{T}$.
 5:     if In first iteration then
 6:         Sample a set of random designs $\{d\}_{i=1}^{N}$ from $D$
 7:     else
 8:         Apply DVF to search for designs $\{d\}_{i=\text{round}(\alpha N)}^{N}$ estimated to be optimal at
            the tasks.
 9:         Apply PIE to search for designs $\{d\}_{i=1}^{\text{round}(\alpha N)}$ estimated to have a high increase
            in performance during policy training.
10:     end if
11:     Apply modular policy to $\{d\}_{i=1}^{N}$ to gather performance $P_0$ before training
12:     Train modular policy with design set $\{d\}_{i=1}^{N}$, e.g. by applying $K$ iterations of
        Alg. 3, or other policy optimization method.
13:     Apply modular policy to $\{d\}_{i=1}^{N}$ to gather performance $P$ after training.
14:     Apply modular policy to additional designs randomly sampled from $D$ to gather
        performance $P$.
15:     Train DVF with newly gathered design/task/performance data.
16:     Train PIE with data $\Delta P = P - P_0$, change of performance from training.
17:     $\alpha \leftarrow \alpha/\beta$
18: end while
19: For new tasks, apply DVF to select robot designs and modular policy to control
    those robots.
```

8.2.4 Integrated design and control synthesis demonstration

The methods in this work could be integrated into a single demonstration. A user could scan in a 3D environment using a handheld or aerial robot with a RGBD sensor, and construct an environment map, for instance, using the work of Zhou et al. [188]. The user could designate a start location and desired direction of motion for a robot, and input that specification to the design generator. The output design could rapidly assembled, then deployed using the modular policy. See Fig. 8-5 for an example.

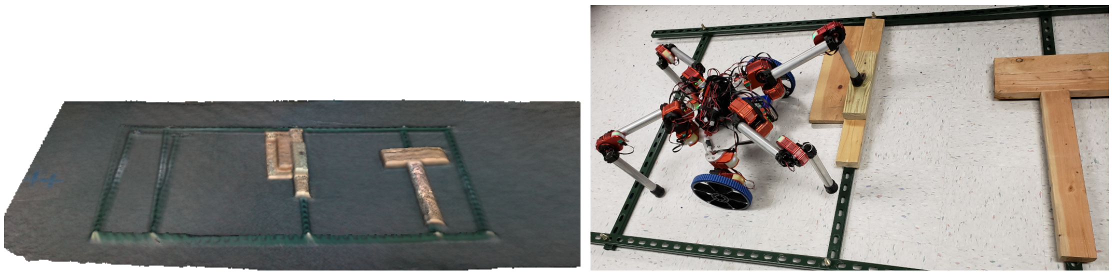


Figure 8-5: The terrain could be scanned in with depth sensors (left), then using our method, the best robot for the terrain could be selected and deployed with the push of a button (right).

8.2.5 Applicable throughout the thesis

Additional modules, tasks, and domains

This thesis, with the exception of Chap. 6, was limited to legs, wheels, and a body as the modules. We have investigated training a policy with additional modules. For instance, a modular policy was trained for combination leg and wheel and used in our work [77]. Future work may seek to create graphs with more modules in them, which will in turn create GNNs that require greater computation to train.

This thesis focuses on some fundamental tasks in manipulation (pick and place) and locomotion (travel as fast as possible forwards over the terrain). Future work may seek to expand the range of tasks, for instance by combining manipulation and locomotion (travel to a point and pick up an object), or to create more complex tasks or constraints on those tasks. While this thesis considers modular robots as the application domain for our work on design and control automation, these ideas may be applicable to domains outside of robotics, for instance, in chemical engineering (e.g. molecule generation) or energy systems (e.g. designing and controlling a set of wind turbines [72]).

Metrics

In testing control and design synthesis, we often found we lacked appropriate metrics by which to measure designs, tasks, and generalization. For example, there is no uniform definition of the difficulty of an environment. The difficulty relates both the to task and design, e.g. the complexity of the robot, the maximum torque/power capabilities of the robot, its sensing capabilities, and its controller capacity. Similarly, when a design or controller is transferred to a new scenario, some forms of generalization appear more difficult than others. Take for example the transfer of a policy trained with a combination of legs and wheels. Transfer to a new design with only wheels tended to result in good performance, seemingly because wheels are "easier" to control. But, transfer to a design with only legs is "harder," if only because we found our policies did generalize to new robots with more legs, but not as well as it did to wheels. One potential future direction is to predict how well a policy or design generator will generalize to a new task, or estimate how much additional training would be needed to reach competency at a new task.

Scaling laws

We observed that when adding more modules and designs to the modular policies and design value function training processes, the time taken to train and number of samples needed to reach satisfactory performance increased, but not always linearly with the number of modules or designs. Future work could further investigate how sample-efficiency is affected by increased number of designs and modules, in particular, when the designs added to the training set may be very similar (only different by one module, perhaps) to those already in the training set, such that the data they produce is somehow "similar."

Applications

We envision our methods could be used in applications where there is a finite mass or monetary budget for parts, and a need to apply the same modules to applications that change frequently, but where the set of module types remains fixed, for instance, in space, low-volume manufacturing, or military applications.

One potential future application of this work is in self-reconfigurable robots. Prior work on self-reconfiguration often assumes that the initial and final designs are provided by a user. Our work could be used to automatically select the final design, enabling a higher degree of autonomy in future self-reconfigurable robots. In this thesis, we use manually reconfigurable modules which do not have self-reconfiguration mechanisms, but our methods would be suitable for self-reconfigurable robots as well.

Another potential application of modular robots is urban search and rescue or disaster response. In such a deployment, one does not know in advance what type of environment one will encounter. One possibility may be to transport a set of robots, as contingencies for each type of environment. The total cost and mass of robot components can be reduced by reusing a small general set of modules to build a robot with some combination of legs and wheels specially adapted to the environment. Testing an integrated modular automated control and design system in a mock disaster site could inform which parts of the pipelines are weakest, or reveal new problems to be solved.

8.3 Conclusions

This thesis developed tools that augment the conventional robot engineering process. 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, because as modular hardware become less expensive to produce, automated design tools like ours could help lower the barrier to robot prototyping. Further, as modular robot hardware components become more commonplace, we believe the need for scalable methods to simulate, prototype, and evaluate the potential of the many possible designs will grow as well. Modular robots and synthesis algorithms could have wide reaching impacts given the variety of potential applications for modular systems in areas such as manufacturing, defense, or space exploration. Though we used a limited set of modules, and created robots operating in a handful of environments, this work represents a stepping stone towards the vision of rapidly-deployable task-specific robots.

In our long-term vision of a broader approach to modularity, modules can not only be the individual joints, links, legs, or wheels on an individual robot, but can also be parts in a full system of inter-connected robots working together.

Modular policies form a framework by which robots with different designs can pool knowledge learned when they share some parts in common. The robots could combine reinforcement and imitation learning to learn from one another, even when they do not have identical designs. Modular policies also form a basis for robots to adapt to new or upgraded components without learning from scratch.

Design value functions could serve as a basis for system-level adaptive hardware compositions. A fleet of robots with various different designs could share a pool of modules, autonomously decide when to reassemble to adapt to new tasks, decide how to make the best use of new modules added to the system, and grow towards ever-changing user needs. Design automation could also serve within a more sustainable re-use structure for robot components. If all parts in a robot system are modular, then when some modules fail or become obsolete, the system could decide how to reconfigure to best incorporate remaining modules into other robots, or decide which modules to shift between robots to fill gaps. These concepts could be applicable to future domains like extra-planetary exploration, in which a collection of modules could be launched without exact knowledge of the environment in which those parts will land. The system could observe the environment, decide how to self-assemble into a collection of robots best suited for exploration or construction tasks, and then continuously share knowledge while learning to improve their collective control policies.

Our methods aim to make modules, both in physical hardware and in their control, into the general building blocks with which robots can specialize to a task. Similarly to the way in which computer components have standardized and become more modular over the past few decades, we believe standardization and modularity in robot components will be a critical step towards a future where robots are ubiquitous in daily life.

Appendix A

Appendix

A.1 Modular policy learning

A.1.1 Hyperparameters and cost functions for policy learning

All neural networks in this work were implemented using PyTorch [118]. The many hyperparameters listed below were tuned by hand. Tuning was conducted on each process independently (i.e. tuning the model learning first on its own, then tuning the trajectory optimization on its own, etc.).

A.1.2 Simulator parameters

The simulator used throughout this work had the following settings:

- Simulator: Pybullet [40]
- Simulator time step: 1/240 seconds
- Time steps per control action: 20, resulting in an effective time step for learned model and controller of 20/240 seconds.

A.1.3 Approximate dynamics model learning

The dynamics model learning process used the following settings:

- Length of random rollouts: 100 steps
- Number of random rollouts per design: 300 rollouts per actuated joint on the designs (between 12 and 18 joints).
- Batch size per design forward pass: 500
- GNN (internal state, message, hidden layer) size: (100, 50, 350)

- GNN (input, message, update, output) function hidden layers: (0, 1, 0,0)
- GNN Update function LSTM hidden size: 50
- GNN training steps initial: 10000
- GNN training steps after each set of TrajOpt complete: 1000
- Learning rate: 1×10^{-3}, decays by half every 2500 steps
- GNN weight decay (weight norm penalty): 10^{-4}
- Number of designs used per step: 6, increasing by one every 2000 steps

A.1.4 Trajectory optimization

The trajectory optimization used the following cost weights:

- Horizon length $H = 20$
- Execute the first $n_{ex} = 10$ steps of each optimized control sequence
- Cost on norm control signal: 0.01
- Slew rate cost: 7
- Cost $w_z||z - z_d||^2$ for z-position (height) of body: $w_z = 5$, $z_d = 0.23$
- Cost $w_r||r||^2$ for roll of body: $w_r = 30$
- Cost $w_p||p||^2$ for pitch of body: $w_p = 20$
- Cost $w_x||x - x_d||^2 + w_y||y - y_d||^2$ for time-varying x- and y-position of body, following desired goal position over time: $w_x = w_y = 110$, $x_d = x_0 + v_{x,d}t$ $y_d = y_0 + v_{y,d}t$, given body position (x_0, y_0) at the start of the MPC replanning step
- Cost $w_\gamma||\gamma - \gamma_d||^2$ for time-varying yaw of body, following desired goal yaw over time: $w_\gamma = 25$, $\gamma_d = \gamma_0 + \omega_{\gamma,d}t$, given body yaw γ_0 at the start of the MPC replanning step
- Penalty for joint angle of open-loop gait style for first joint on wheel modules (set to zero joint angle for all time): 0.4
- Penalty for joint angle of open-loop gait style on legs (set to zero joint angle for all time for first joint, sine wave with amplitude 0.6 rad and period 1.25 s for second and third joint): 6
- Maximum goal velocity in (x,y) directions, based on maximum wheel rotation rate: 0.7 m/s
- Maximum goal velocity in yaw directions, based on maximum wheel rotation rate and body radius: 2.4 rad/s

A.1.5 Control policy learning

The control policy learning process used the following settings:

- Length of expert TrajOpt rollouts: 40 steps
- Number of expert rollouts per design: 750, or 1000 at final iteration
- Batch size per design forward pass: 500
- GNN (internal state, message, hidden layer) size: (100, 50, 250)
- GNN [input, message, update, output] function hidden layers: (0, 1, 2, 0)
- GNN Update function LSTM hidden size: 50
- GNN training steps: 8000
- Learning rate: 3×10^{-3}, decays by half every 2000 steps
- GNN weight decay (weight norm penalty): 10^{-4}
- Feed-forward torque loss weighting: 0.25

A.1.6 Velocity matching metric

The velocity metric V is calculated based on the desired x, y, and yaw changes over n_{ex} time steps, using the weights from Sec. A.1.5 as follows,

$$\begin{aligned} e_1 &= w_x(\Delta x_{des} - \Delta x)^2 + \\ &\quad w_y(\Delta y_{des} - \Delta y)^2 + \\ &\quad w_\gamma(\Delta \gamma_{des} - \Delta \gamma)^2 \\ e_2 &= w_x(\Delta x_{des})^2 + w_y(\Delta y_{des})^2 + w_\gamma(\Delta \gamma_{des})^2 \\ V &= (e_2 - e_1)/e_2. \end{aligned} \tag{A.1}$$

A.1.7 MLP baseline Hyperparameters

The MLP baselines were set with a comparable depth and capacity as the GNNs. The same hidden layer sizes were used for both the "shared trunk" and "hardware-conditioned" baselines. The shared trunk network has a separate input and output layer for each design, which transforms its inputs and outputs to the hidden layer dimension. The hardware-conditioned network uses the same input and output layers for all designs, but pads the inputs with zeros as needed to reach the input layer size.

- Model network, 6 hidden layers with 300 ReLU units
- Policy network, 6 hidden layers with 350 ReLU units

A.2 Modular visual-motor policy learning

A.2.1 Software packages

All neural networks are implemented in Pytorch [118]. All stochastic gradient descent steps use the Adam optimizer [90]. We used the NVIDIA IsaacGym simulator, which allows for batch computation with GPU acceleration [109].

A.2.2 Model learning details and hyperparameters

In the first iteration, the model is trained with trajectories from random actions. In subsequent iterations, the model is trained with trajectories from applying the most recent iteration of the policy. The trajectory data from random actions and from previous iterations of the policy are kept in the dataset, but the sampling procedure during stochastic gradient descent is manually biased to sample the newer data more often than uniformly random. As a result, even with relatively few new trajectories relative to the total number of trajectories, the model will devote more of its capacity to the fitting the newer data while preventing catastrophic forgetting. In each batch of sampled states, we force one quarter of the samples to be taken from the newest iteration's data, even when that data makes up less than a quarter of the full dataset.

The model is a message-passing GNN with the following hyperparameters, tuned by hand:

- (Input, update, message, output) function number of hidden layers: [0, 1, 2, 1]
- (Input, update, message, output) function hidden layer size: 400
- (Internal state, messages out, LSTM hidden state) sizes: [250, 100, 75]
- Dropout with fraction 0.05 applied after each hidden layer in the input and update function
- Batchnorm after each convolutional layer in the input function

The first model learning iteration uses 7500 steps, starting with a multi-step loss sequence length of 2 and a learning rate of $4e-3$, and increasing the sequence length and decreasing the learning rate progressively over the course of training up to a sequence length of 10. The batch size was set to 1000 in the first iteration to allow space on the GPU for the full dataset to be stored.

A.2.3 Policy optimization details and hyperparameters

The policy is a message-passing GNN with the following hyperparameters, tuned by hand:

- (Input, update, message, output) function number of hidden layers: [0, 1, 1, 1]
- (Input, update, message, output) function hidden layer size: 400

- (Internal state, messages out, LSTM hidden state) sizes: [250, 100, 75]
- Batchnorm after each convolutional layer in the input function

The policy is optimized with 250 steps in the first iteration, 200 on the second, and 150 on subsequent iterations. The learning rate starts at $6e-4$ during the first iteration, $5e-4$ in the second, $3e-4$ in the third, and decays every 100 steps. The policy time horizon $T = 30$, and the batch size 350.

We also emulate hardware latency caused by the time taken to compute the policy actions and send those commands to the actuators. The policy is given the previous instead of the current state.

A.2.4 Reward weights

The reward function was composed of the following terms and weights:

- Mean squared body roll, weight (running cost): 1.0
- Mean squared body pitch: 1.35
- Mean squared body yaw, to point the body forward: 2.5
- Mean squared yaw velocity, to prevent high yaw rate: 0.01
- Absolute value of joint angle deviation from center,
  - First leg joint: 0.075
  - Second leg joint: 0.05
  - Third leg joint: 0.1
  - First wheel joint: 0.075
  - Second (continuous) wheel joint: 0
- Penalty when the leg folds in (second joint <0 and third joint>0), absolute value of total bend: 4.0
- Control action sum squared: 0.0005
- Slew rate (change between subsequent actions): 0.0275
- Height of the body above the ground: 30.0 for each meter above 0.18 m
- RL entropy bonus: 0.03
- IL entropy penalty: 1.5
- x-direction position change after T steps (terminal cost): 100
- y-direction position change after T steps: -10

A.2.5 Other settings

The simulation was set to a time step of 1/60 s, with each action repeated 5 times for an effective time step of 5/60 s. The random-action data generation phase created 10,000 trajectories of length 100, split up among the designs in the training set proportional to the number of actuators in each design. The policy was applied 1200 times with the first application method and 600 with the second at each iteration, split up among the designs in the training set proportional to the number of actuators in each design. Each trajectory was length 150 time steps for the first method, and 75 for the second. The simulation was parallelized using the GPU acceleration from IsaacGym.

The full loop was run for 20 iterations in each experiment, then we select the best iteration, as some fluctuation in performance may occur after the maximum terrain difficulty level is reached. Including the initial data generation phase, this took approximately 12 hours for the three-design-two-environment experiments using one GTX 1070 with 8 Gb memory. The time taken decreases nearly linearly with the number of designs. This is mainly due to our implementation using separate forward passes, with separate mini-batches, for each design, the gradients of which are averaged at each training step. The primary bottlenecks are in the policy optimization step, which can gain significant speed improvements when using newer graphics cards with larger memory allowing for larger batch sizes.

A.3 Modular policy learning with imitation

A.3.1 Imitation learning details and hyperparameters

The initial policy was learned with imitation via behavioral cloning, with a maximum sequence length for recurrent supervised learning of 10. This phase trained for 14000 steps with a batch size of 500 and learning rate of 2e-3, decayed periodically.

During policy optimization, the prior weight λ was set to 0.25 in the first iteration, then decayed in subsequent iterations. The batch size for the IL loss was 200.

Bibliography

[1] Blind bipedal stair traversal via sim-to-real reinforcement learning. In *Robotics: Science and Systems*, 07 2021.

[2] Anurag Ajay, Maria Bauza, Jiajun Wu, Nima Fazeli, Joshua B Tenenbaum, Alberto Rodriguez, and Leslie P Kaelbling. Combining physical simulators and object-based networks for control. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3217–3223. IEEE, 2019.

[3] Ferran Alet, Tomás Lozano-Pérez, and Leslie P Kaelbling. Modular meta-learning. In *Conference on Robot Learning*, pages 856–868. PMLR, 2018.

[4] Ferran Alet, Erica Weng, Tomás Lozano-Pérez, and Leslie P Kaelbling. Neural relational inference with fast modular meta-learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[5] Ferran Alet, Erica Weng, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Neural relational inference with fast modular meta-learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 11827–11838. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/b294504229c668e750dfcc4ea9617f0a-Paper.pdf`.

[6] Ferran Alet, Martin F Schneider, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Meta-learning curiosity algorithms. In *International Conference on Learning Representations (ICLR)*, 2020.

[7] M Althoff, A Giusti, SB Liu, and A Pereira. Effortless creation of safe robots from modules through self-programming and self-verification. *Science Robotics*, 4(31):eaaw1924, 2019.

[8] Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable mpc for end-to-end planning and control. In *Proceedings of the 32nd Int. Conf. on Neural Information Processing Systems*, pages 8299–8310, 2018.

[9] Brandon Amos, Samuel Stanton, Denis Yarats, and Andrew Gordon Wilson. On the model-based stochastic value gradient for continuous reinforcement learning. In *Learning for Dynamics and Control*, pages 6–20. PMLR, 2021.

[10] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[11] Jacob Arkin, Daehyung Park, Subhro Roy, Matthew R Walter, Nicholas Roy, Thomas M Howard, and Rohan Paul. Multimodal estimation and communication of latent semantic knowledge for robust execution of robot instructions. *The International Journal of Robotics Research*, 39(10-11):1279–1304, 2020.

[12] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[13] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial intelligence*, 72(1-2):81–138, 1995.

[14] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray kavukcuoglu. Interaction networks for learning about objects, relations and physics. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 4509–4517, 2016.

[15] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[16] Beckhoff Robotics. [Online] `https://www.beckhoff.com/en-us/products/motion/atro-automation-technology-for-robotics/`, 2022.

[17] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual Int. Conf. on machine learning*, pages 41–48. ACM, 2009.

[18] Mohak Bhardwaj, Sanjiban Choudhury, and Sebastian Scherer. Learning heuristic search via imitation. In *Conference on Robot Learning*, pages 271–280, 2017.

[19] ZM Bi and Wen-Jun Zhang. Concurrent optimal design of modular robotic configuration. *Journal of Robotic systems*, 18(2):77–87, 2001.

[20] Marko Bjelonic, Prajish K Sankar, C Dario Bellicoso, Heike Vallery, and Marco Hutter. Rolling in the deep–hybrid locomotion for wheeled-legged robots using online trajectory optimization. *IEEE Robotics and Automation Letters*, 5(2): 3626–3633, 2020.

[21] G. Bledt and S. Kim. Extracting legged locomotion heuristics with regularized predictive control. In *2020 IEEE Int. Conf. on robotics and Automation (ICRA)*, pages 406–412. IEEE, 2020.

[22] Gerardo Bledt. *Regularized predictive control framework for robust dynamic legged locomotion*. PhD thesis, Massachusetts Institute of Technology, 2020.

[23] Rodney A Brooks and Maja J Mataric. Real robots, real learning problems. In *Robot learning*, pages 193–213. Springer, 1993.

[24] Alberto Brunete, Avinash Ranganath, Sergio Segovia, Javier Perez de Frutos, Miguel Hernando, and Ernesto Gambao. Current trends in reconfigurable modular robots design. *International Journal of Advanced Robotic Systems*, 14(3): 1729881417710457, 2017.

[25] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *ICLR*, 2019.

[26] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, and Nicolas Mansard. The pinocchio c++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *2019 IEEE/SICE International Symposium on System Integration (SII)*, pages 614–619. IEEE, 2019.

[27] Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L Griffiths. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations*, 2018.

[28] Yevgen Chebotar, Mrinal Kalakrishnan, Ali Yahya, Adrian Li, Stefan Schaal, and Sergey Levine. Path integral guided policy search. In *Int. Conf. on robotics and automation (ICRA)*, pages 3381–3388. IEEE, 2017.

[29] Edward Chen, Howie Choset, and John Galeotti. Uncertainty-based adaptive data augmentation for ultrasound imaging anatomical variations. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 438–442. IEEE, 2021.

[30] I-Ming Chen. *Theory and applications of modular reconfigurable robotic systems*. PhD thesis, California Institute of Technology, 1994.

[31] I. Ming Chen. On optimal configuration of modular reconfigurable robots. In *Proceedings of the 4th International Conference on Control, Automation, Robotics, and Vision*, 1996.

[32] I-Ming Chen and Joel W Burdick. Determining task optimal modular robot assembly configurations. In *proceedings of 1995 IEEE International Conference on Robotics and Automation*, volume 1, pages 132–137. IEEE, 1995.

[33] I-Ming Chen, Guilin Yang, and Song Huat Yeo. Automatic modeling for modular reconfigurable robotic systems: Theory and practice. In *Industrial Robotics: Theory, Modelling and Control*. IntechOpen, 2006.

[34] Tao Chen, Adithyavairavan Murali, and Abhinav Gupta. Hardware conditioned policies for multi- transfer learning. In *Advances in Neural Information Processing Systems 31*, pages 9333–9344, 2018.

[35] Nick Cheney, Josh Bongard, Vytas SunSpiral, and Hod Lipson. Scalable co-optimization of morphology and control in embodied machines. *Journal of The Royal Society Interface*, 15(143):20170937, 2018.

[36] Sachin Chitta. *Dynamics and control of a class of modular locomotion systems*. PhD thesis, University of Pennsylvania, 2005.

[37] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[38] Ignasi Clavera, Jonas Rothfuss, John Schulman, Yasuhiro Fujita, Tamim Asfour, and Pieter Abbeel. Model-based reinforcement learning via meta-policy optimization. In *Conference on Robot Learning*, pages 617–629. PMLR, 2018.

[39] R Cohen, MG Lipton, MQ Dai, and B Benhabib. Conceptual design of a modular robot. *Journal of Mechanical Design*, 1992.

[40] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2019.

[41] Michael Crawshaw. Multi-task learning with deep neural networks: A survey. *arXiv preprint arXiv:2009.09796*, 2020.

[42] Jonathan Daudelin, Gangyuan Jing, Tarik Tosun, Mark Yim, Hadas Kress-Gazit, and Mark Campbell. An integrated system for perception-driven autonomy with modular robots. *Science Robotics*, 3(23), 2018.

[43] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

[44] Ruta Desai, Ye Yuan, and Stelian Coros. Computational abstractions for interactive design of robotic devices. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1196–1203. IEEE, 2017.

[45] Ruta Desai, Margarita Safonova, Katharina Muelling, and Stelian Coros. Automatic design of task-specific robotic arms. *arXiv preprint arXiv:1806.07419*, 2018.

[46] Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 2169–2176. IEEE, 2017.

[47] Yiming Ding, Carlos Florensa, Pieter Abbeel, and Mariano Phielipp. Goal-conditioned imitation learning. *Advances in neural information processing systems*, 32, 2019.

[48] Nick Eckenstein and Mark Yim. Modular reconfigurable robotic systems: Lattice automata. In *Robots and Lattice Automata*, pages 47–75. Springer, 2015.

[49] Péter Fankhauser, Michael Bloesch, Christian Gehring, Marco Hutter, and Roland Siegwart. Robot-centric elevation mapping with uncertainty estimates. In *International Conference on Climbing and Walking Robots (CLAWAR)*, 2014.

[50] Péter Fankhauser, Michael Bloesch, and Marco Hutter. Probabilistic terrain mapping for mobile robots with uncertain localization. *IEEE Robotics and Automation Letters (RA-L)*, 3(4):3019–3026, 2018. doi: 10.1109/LRA.2018.2849506.

[51] Shane Farritor, Steven Dubowsky, Nathan Rutman, and Jeffrey Cole. A systems-level modular design approach to field robotics. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 2890–2895. IEEE, 1996.

[52] Zipeng Fu, Ashish Kumar, Jitendra Malik, and Deepak Pathak. Minimizing energy consumption leads to the emergence of gaits in legged robots. 2021.

[53] Moritz Geilinger, Roi Poranne, Ruta Desai, Bernhard Thomaszewski, and Stelian Coros. Skaterbots: Optimization-based design and motion synthesis for creatures with legs and wheels. *ACM Transactions on Graphics (TOG)*, 37 (4):160, 2018.

[54] Markus Giftthaler. *Towards a Unified Framework of Efficient Algorithms for Numerical Optimal Control*. PhD thesis, ETH Zurich, 2018.

[55] Kyle Gilpin and Daniela Rus. Modular robot systems. *IEEE robotics & automation magazine*, 17(3):38–55, 2010.

[56] Kevin G Gim and Joohyung Kim. Snapbot v2: a reconfigurable legged robot with a camera for self configuration recognition. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4026–4031. IEEE, 2020.

[57] Andrea Giusti and Matthias Althoff. On-the-fly control design of modular robot manipulators. *IEEE Transactions on Control Systems Technology*, 26(4):1484–1491, 2017.

[58] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

[59] Felix Grimminger, Avadesh Meduri, Majid Khadiv, Julian Viereck, Manuel Wüthrich, Maximilien Naveau, Vincent Berenz, Steve Heim, Felix Widmaier, Thomas Flayols, et al. An open torque-controlled modular robot architecture for legged locomotion research. *IEEE Robotics and Automation Letters*, 5(2): 3650–3657, 2020.

[60] Grin Technologies. [Online] `https://ebikes.ca`, 2022.

[61] Agrim Gupta, Silvio Savarese, Surya Ganguli, and Li Fei-Fei. Embodied intelligence via learning and evolution. *arXiv preprint arXiv:2102.02202*, 2021.

[62] David Ha. Reinforcement learning for improving agent design. *Artificial Life*, 25(4):352–365, 2019.

[63] Sehoon Ha, Stelian Coros, Alexander Alspach, James M Bern, Joohyung Kim, and Katsu Yamane. Computational design of robotic devices from high-level motion specifications. *IEEE Transactions on Robotics*, 34(5):1240–1251, 2018.

[64] Sehoon Ha, Joohyung Kim, and Katsu Yamane. Automated deep reinforcement learning environment for hardware of a modular legged robot. In *2018 15th International Conference on Ubiquitous Robots (UR)*, pages 348–354. IEEE, 2018.

[65] Sehoon Ha, Peng Xu, Zhenyu Tan, Sergey Levine, and Jie Tan. Learning to walk in the real world with minimal human effort. *arXiv preprint arXiv:2002.08550*, 2020.

[66] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[67] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2019.

[68] Roland Hafner, Tim Hertweck, Philipp Kloeppner, Michael Bloesch, Michael Neunert, Markus Wulfmeier, Saran Tunyasuvunakool, Nicolas Heess, and Martin Riedmiller. Towards general and autonomous learning of core skills: A case study in locomotion. In *Conference on Robot Learning*, pages 1084–1099. PMLR, 2021.

[69] Jeffrey Hawke, Richard Shen, Corina Gurau, Siddharth Sharma, Daniele Reda, Nikolay Nikolov, Przemysław Mazur, Sean Micklethwaite, Nicolas Griffiths, Amar Shah, et al. Urban driving with conditional imitation learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 251–257. IEEE, 2020.

[70] Hebi Robotics. [Online] `www.hebirobotics.com`, 2022.

[71] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[72] Rockey Hester. Wind farm yaw optimization and real-time control using graph neural networks. Master's thesis, Carnegie Mellon University, 2021.

[73] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[74] Gregory S Hornby, Hod Lipson, and Jordan B Pollack. Generative representations for the automated design of modular physical robots. *IEEE transactions on Robotics and Automation*, 19(4):703–719, 2003.

[75] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

[76] Jeff Hu, Howard Coffin, Julian Whitman, Matt Travers, and Howie Choset. Large-scale heterogeneous multi-robot coverage via domain decomposition and generative allocation. In *International Workshop on the Algorithmic Foundations of Robotics.*, 2022.

[77] Jeff Hu, Julian Whitman, Matt Travers, and Howie Choset. Modular robot design optimization with generative adversarial networks. In *Proceedings of the International Conference on Robotics and Automation.* IEEE, 2022.

[78] Wenlong Huang, Igor Mordatch, and Deepak Pathak. One policy to control them all: Shared modular policies for agent-agnostic control. In *ICML*, 2020.

[79] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[80] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40 (4-5):698–721, 2021.

[81] Esra Icer and Matthias Althoff. Cost-optimal composition synthesis for modular robots. In *2016 IEEE Conference on Control Applications (CCA)*, pages 1408–1413. IEEE, 2016.

[82] Esra Icer, Heba A Hassan, Khaled El-Ayat, and Matthias Althoff. Evolutionary cost-optimal composition synthesis of modular robots considering a given task. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3562–3568. IEEE, 2017.

[83] Jeffrey Ichnowski, Yahav Avigal, Vishal Satish, and Ken Goldberg. Deep learning can accelerate grasp-optimized motion planning. *Science Robotics*, 5(48): eabd7710, 2020.

[84] Simon Kalouche, David Rollinson, and Howie Choset. Modularity for maximum mobility and manipulation: Control of a reconfigurable legged robot with series-elastic actuators. In *2015 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 1–8. IEEE, 2015.

[85] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[86] Petros Karamanakos, Eyke Liegmann, Tobias Geyer, and Ralph Kennel. Model predictive control of power electronic systems: Methods, results, and challenges. *IEEE Open Journal of Industry Applications*, 1:95–114, 2020.

[87] Sisir Karumanchi, Kyle Edelberg, Ian Baldwin, Jeremy Nash, Jason Reid, Charles Bergh, John Leichty, Kalind Carpenter, Matthew Shekels, Matthew Gildner, et al. Team robosimian: semi-autonomous mobile manipulation at the 2015 darpa robotics challenge finals. *Journal of Field Robotics*, 34(2):305–332, 2017.

[88] Elia Kaufmann, Antonio Loquercio, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Deep drone acrobatics. In *Robotics: Science and Systems*, July 2020.

[89] Joohyung Kim, Alexander Alspach, and Katsu Yamane. Snapbot: a reconfigurable legged robot. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5861–5867. IEEE, 2017.

[90] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Int. Conf. on machine learning*, 2015.

[91] J Zico Kolter and Gaurav Manek. Learning stable deep dynamics models. In *Advances in Neural Information Processing Systems*, pages 11128–11136, 2019.

[92] Michael S Konstantinov. Servicing of industrial robots-the modular concept. In *Performance Evaluation of Programmable Robots and Manipulators: Report of a Workshop Held at Annapolis, Maryland, October 23-25, 1975*, volume 459, page 123. US Department of Commerce, National Bureau of Standards, 1976.

[93] Keith Kotay, Daniela Rus, Marsette Vona, and Craig McGray. The self-reconfiguring robotic molecule. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, volume 1, pages 424–431. IEEE, 1998.

[94] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. 2021.

[95] Nathan Lambert, Brandon Amos, Omry Yadan, and Roberto Calandra. Objective mismatch in model-based reinforcement learning. *arXiv preprint arXiv:2002.04523*, 2020.

[96] Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

[97] Jay H Lee. Model predictive control and dynamic programming. In *2011 11th International Conference on Control, Automation and Systems*, pages 1807–1809. IEEE, 2011.

[98] Chris Leger. *Darwin2K: An evolutionary approach to automated design for robotics*, volume 574. Springer Science & Business Media, 2012.

[99] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

[100] Sergey Levine and Vladlen Koltun. Guided policy search. In *Int. Conf. on Machine Learning*, pages 1–9, 2013.

[101] Weibing Li, Robert C Richardson, and Jongrae Kim. A tri-state prismatic modular robotic system. *Mechatronics*, 64:102287, 2019.

[102] Guanqi Liang, Haobo Luo, Ming Li, Huihuan Qian, and Tin Lun Lam. Freebot: A freeform modular self-reconfigurable robot with arbitrary connection point-design and implementation. In *IEEE/RSJ Int. Conf. Intell. Robots Syst., Las Vegas, USA*, 2020.

[103] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*, pages 270–282. PMLR, 2018.

[104] Hod Lipson and Jordan B Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978, 2000.

[105] Chao Liu, Sencheng Yu, and Mark Yim. Motion planning for variable topology truss modular robot. In *Proceedings of Robotics: Science and Systems*, 2020.

[106] Chao Liu, Qian Lin, Hyun Kim, and Mark Yim. Smores-ep, a modular robot with parallel self-assembly. *arXiv preprint arXiv:2104.00800*, 2021.

[107] Stefan B Liu and Matthias Althoff. Optimizing performance in automation through modular robots. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4044–4050. IEEE, 2020.

[108] Kevin Sebastian Luck, Heni Ben Amor, and Roberto Calandra. Data-efficient co-adaptation of morphology and behaviour with deep reinforcement learning. In *Conference on Robot Learning*, pages 854–869. PMLR, 2020.

[109] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[110] Daniel Marbach and Auke Jan Ijspeert. Co-evolution of configuration and control for homogenous modular robots. In *Proceedings of the eighth conference on intelligent autonomous systems (IAS8)*, number CONF, pages 712–719. IOS Press, 2004.

[111] Takahiro Miki, Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, 7(62):eabk2822, 2022.

[112] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[113] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE Int. Conf. on robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[114] Michael Neunert, Abbas Abdolmaleki, Markus Wulfmeier, Thomas Lampe, Tobias Springenberg, Roland Hafner, Francesco Romano, Jonas Buchli, Nicolas Heess, and Martin Riedmiller. Continuous-discrete reinforcement learning for hybrid control in robotics. In *Conference on Robot Learning*, pages 735–751. PMLR, 2020.

[115] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntak Lee, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Agile autonomous driving using end-to-end deep imitation learning. In *Robotics: Science and Systems*, 2018.

[116] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos A Theodorou, and Byron Boots. Imitation learning for agile autonomous driving. *The International Journal of Robotics Research*, 39(2-3): 286–302, 2020.

[117] Aaron Parness. Science objectives and rover design for a limbed comet rover mission concept. In *2017 IEEE Aerospace Conference*, pages 1–7. IEEE, 2017.

[118] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[119] Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A. Efros. Learning to control self- assembling morphologies: A study of generalization via modularity. In *arXiv preprint arXiv:1902.05546*, 2019.

[120] Chandana Paul, Francisco J Valero-Cuevas, and Hod Lipson. Design and control of tensegrity robots for locomotion. *IEEE Transactions on Robotics*, 22(5):944–957, 2006.

[121] Xue Bin Peng, Erwin Coumans, Tingnan Zhang, Tsang-Wei Edward Lee, Jie Tan, and Sergey Levine. Learning agile robotic locomotion skills by imitating animals. In *Robotics: Science and Systems*, 07 2020. doi: 10.15607/RSS.2020.XVI.064.

[122] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[123] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34 (4):1004–1020, 2018.

[124] Aravind Rajeswaran, Igor Mordatch, and Vikash Kumar. A game theoretic framework for model based reinforcement learning. In *International Conference on Machine Learning*, pages 7953–7963. PMLR, 2020.

[125] James B Rawlings. Tutorial overview of model predictive control. *IEEE control systems magazine*, 20(3):38–52, 2000.

[126] William Reid, Blair Emanuel, Brendan Chamberlain-Simon, Sisir Karumanchi, and Gareth Meirion-Griffith. Mobility mode evaluation of a wheel-on-limb rover on glacial ice analogous to europa terrain. In *2020 IEEE Aerospace Conference*, pages 1–9. IEEE, 2020.

[127] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[128] David Rollinson, Yigit Bilgen, Ben Brown, Florian Enner, Steven Ford, Curtis Layton, Justine Rembisz, Mike Schwerin, Andrew Willig, Pras Velagapudi, et al. Design and architecture of a series elastic snake robot. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4630–4636. IEEE, 2014.

[129] John W Romanishin, Kyle Gilpin, Sebastian Claici, and Daniela Rus. 3d m-blocks: Self-reconfiguring robots capable of locomotion via pivoting in three dimensions. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1925–1932. IEEE, 2015.

[130] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[131] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *5th Annual Conference on Robot Learning*, 2021.

[132] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[133] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[134] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

[135] Benjamin Sanchez-Lengeling and Alán Aspuru-Guzik. Inverse molecular design using machine learning: Generative models for matter engineering. *Science*, 361 (6400):360–365, 2018.

[136] Guillaume Sartoretti, William Paivine, Yunfei Shi, Yue Wu, and Howie Choset. Distributed learning of decentralized control policies for articulated mobile robots. *IEEE Transactions on Robotics*, 35(5):1109,1122, 2019-10. ISSN 1552-3098.

[137] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[138] Charles Schaff, David Yunis, Ayan Chakrabarti, and Matthew R Walter. Jointly learning to construct and control agents using deep reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*, pages 9798–9805. IEEE, 2019.

[139] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1312–1320, 2015.

[140] Donald Schmitz, Pradeep Khosla, and Takeo Kanade. The cmu reconfigurable modular manipulator system. 1988.

[141] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[142] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[143] Schunk. Lightweight arm for changing application locations. `https://schunk.com/de_en/press/press-service/press-releases/article/743-lightweight-arm-for-changing-application-locations`, 2021.

[144] Jungwon Seo, Jamie Paik, and Mark Yim. Modular reconfigurable robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:63–88, 2019.

[145] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[146] Alexander Sproewitz, Aude Billard, Pierre Dillenbourg, and Auke Jan Ijspeert. Roombots-mechanical design of self-reconfiguring modular robots for adaptive furniture. In *2009 IEEE international conference on robotics and automation*, pages 4259–4264. IEEE, 2009.

[147] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[148] Kasper Stoy, David Brandt, David J Christensen, and David Brandt. *Self-reconfigurable robots: an introduction.* MIT press Cambridge, 2010.

[149] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[150] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[151] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018. doi: 10.15607/RSS.2018.XIV.010.

[152] Ning Tan, Abdullah Aamir Hayat, Mohan Rajesh Elara, and Kristin L Wood. A framework for taxonomy and evaluation of self-reconfigurable robotic systems. *IEEE Access*, 8:13969–13986, 2020.

[153] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.

[154] Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. Robots that use language. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:25–55, 2020.

[155] Ryan Therrien and Scott Doyle. Role of training data variability on classifier performance and generalizability. In *Medical Imaging 2018: Digital Pathology*, volume 10581, page 1058109. International Society for Optics and Photonics, 2018.

[156] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[157] Universal Robotics. [Online] `https://www.universal-robots.com/plus/products/`, 2022.

[158] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[159] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[160] Sean J Wang and Aaron M Johnson. Domain adaptation using system invariant dynamics models. In *Learning for Dynamics and Control*, pages 1130–1141. PMLR, 2021.

[161] Sean J Wang, Samuel Triest, Wenshan Wang, Sebastian Scherer, and Aaron Johnson. Rough terrain navigation using divergence constrained model-based reinforcement learning. In *Conference on Robot Learning*, pages 224–233. PMLR, 2022.

[162] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *Int. Conf. on Learning Representations*, 2018.

[163] Tingwu Wang, Yuhao Zhou, Sanja Fidler, and Jimmy Ba. Neural graph evolution: Automatic robot design. In *Int. Conf. on Learning Representations*, 2019.

[164] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.

[165] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[166] Julian Whitman and Howie Choset. Task-specific manipulator design and trajectory synthesis. *IEEE Robotics and Automation Letters*, 4(2):301–308, 2018.

[167] Julian Whitman, Shuang Su, Stelian Coros, Alex Ansari, and Howie Choset. Generating gaits for simultaneous locomotion and manipulation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2723–2729. IEEE, 2017.

[168] Julian Whitman, Raunaq Bhirangi, Matthew J Travers, and Howie Choset. Modular robot design synthesis with deep reinforcement learning. In *AAAI*, pages 10418–10425, 2020.

[169] Julian Whitman, Matthew Travers, and Howie Choset. Modular mobile robot design selection with deep reinforcement learning. In *Machine Learning for Engineering Modeling, Simulation, and Design Workshop, at Neural Information Processing Systems*, 2020.

[170] Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. Information-theoretic model predictive control: Theory and applications to autonomous driving. *IEEE Transactions on Robotics*, 34(6): 1603–1622, 2018.

[171] Alexander W Winkler, C Dario Bellicoso, Marco Hutter, and Jonas Buchli. Gait and trajectory optimization for legged systems through phase-based end-effector parameterization. *IEEE Robotics and Automation Letters*, 3(3):1560–1567, 2018.

[172] Kevin C Wolfe, Matthew S Moses, Michael DM Kutzer, and Gregory S Chirikjian. M 3 express: a low-cost independently-mobile reconfigurable modular robot. In *2012 IEEE International Conference on Robotics and Automation*, pages 2704–2710. IEEE, 2012.

[173] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[174] Zhaoming Xie, Patrick Clary, Jeremy Dao, Pedro Morais, Jonanthan Hurst, and Michiel Panne. Learning locomotion skills for cassie: Iterative design and sim-to-real. In *Conf. on Robot Learning*, pages 317–329, 2020.

[175] Guilin Yang and I-Ming Chen. Task-based optimization of modular robot configurations: minimized degree-of-freedom approach. *Mechanism and machine theory*, 35(4):517–540, 2000.

[176] Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, and Vikas Sindhwani. Data efficient reinforcement learning for legged robots. In *Conf. on Robot Learning*, pages 1–10. PMLR, 2020.

[177] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics Automation Magazine*, 14(1):43–52, 2007.

[178] Mark Yim. *Locomotion with unit-modular reconfigurable robot.* PhD thesis, Stanford University, 1994.

[179] Mark Yim, Paul J White, Michael Park, and Jimmy Sastra. Modular self-reconfigurable robots.

[180] Mark Yim, David G Duff, and Kimon Roufas. Modular reconfigurable robots, an approach to urban search and rescue. In *the Proceedings of the 1st International Workshop on Human-friendly Welfare Robotics Systems, Taejon, Korea*, 2000.

[181] Mark Yim, David G Duff, and Kimon D Roufas. Polybot: a modular reconfigurable robot. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 1, pages 514–520. IEEE, 2000.

[182] Wenhao Yu, Deepali Jain, Alejandro Escontrela, Atil Iscen, Peng Xu, Erwin Coumans, Sehoon Ha, Jie Tan, and Tingnan Zhang. Visual-locomotion: Learning to walk on complex terrains with vision. In *5th Annual Conference on Robot Learning*, 2021.

[183] Baohe Zhang, Raghu Rajan, Luis Pineda, Nathan Lambert, André Biedenkapp, Kurtland Chua, Frank Hutter, and Roberto Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 4015–4023. PMLR, 2021.

[184] Marvin Zhang, Xinyang Geng, Jonathan Bruce, Ken Caluwaerts, Massimo Vespignani, Vytas SunSpiral, Pieter Abbeel, and Sergey Levine. Deep reinforcement learning for tensegrity locomotion. In *2017 IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 634–641. IEEE, 2017.

[185] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1): 1–23, 2019.

[186] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Facial landmark detection by deep multi-task learning. In *European conference on computer vision*, pages 94–108. Springer, 2014.

[187] Allan Zhao, Jie Xu, Mina Konaković-Luković, Josephine Hughes, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. Robogrammar: graph grammar for terrain-optimized robot design. *ACM Transactions on Graphics (TOG)*, 39(6): 1–16, 2020.

[188] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

[189] Zhenpeng Zhou, Steven Kearnes, Li Li, Richard N Zare, and Patrick Riley. Optimization of molecules via deep reinforcement learning. *Scientific reports*, 9 (1):1–10, 2019.