Discovering Adaptable Symbolic Algorithms from Scratch

Stephen Kelly[1,4], Daniel S. Park[1], Xingyou Song[1,2], Mitchell McIntire[3], Pranav Nashikkar[3],
Ritam Guha[5], Wolfgang Banzhaf[5], Kalyanmoy Deb[5], Vishnu Naresh Boddeti[5], Jie Tan[1,2], Esteban Real[1,2]
[1]Google Research, [2] Google DeepMind, [3]Google, [4]McMaster University, [5]Michigan State University

COIN Report Number 2023010

***Abstract*— Autonomous robots deployed in the real world will need control policies that rapidly adapt to environmental changes. To this end, we propose AutoRobotics-Zero (ARZ), a method based on AutoML-Zero that discovers zero-shot adaptable policies from scratch. In contrast to neural network adaption policies, where only model parameters are optimized, ARZ can build control algorithms with the full expressive power of a linear register machine. We evolve modular policies that tune their model parameters *and* alter their inference algorithm on-the-fly to adapt to sudden environmental changes. We demonstrate our method on a realistic simulated quadruped robot, for which we evolve safe control policies that avoid falling when individual limbs suddenly break. This is a challenging task in which two popular neural network baselines fail. Finally, we conduct a detailed analysis of our method on a novel and challenging non-stationary control task dubbed Cataclysmic Cartpole. Results confirm our findings that ARZ is significantly more robust to sudden environmental changes and can build simple, interpretable control policies.**

I. Introduction

Robots deployed in the real world will inevitably face many environmental changes. For example, robots' internal conditions, such as battery levels and physical wear-and-tear, and external conditions, such as new terrain or obstacles, imply that the system's dynamics are non-stationary. In these situations, a static controller that always maps the same state to the same action is rarely optimal. Robots must be capable of continuously adapting their control policy in response to the changing environment. To achieve this capability, they must recognize a change in the environment without an external cue, purely by observing how actions change the system state over time, and update their control in response. Recurrent deep neural networks are a popular policy representation to support fast adaptation. However, they are often (1) monolithic, which leads to the *distraction dilemma* when attempting to learn policies that are robust to multiple dissimilar environmental physics [1], [2]; (2) overparameterized, which can lead to poor generalization and long inference time; and (3) difficult to interpret. Ideally, we would like to find a policy that can express multiple modes of behavior while still being simple and interpretable.

We propose AutoRobotics-Zero (ARZ), a new framework based on AutoML-Zero (AMLZ) [3] to specifically support the evolution of dynamic, self-modifying control policies in a realistic quadruped robot adaptation task. We represent these policies as *programs* instead of neural networks and demonstrate how the adaptable policy and its initial parameters can be evolved from scratch using only basic mathematical operations as building blocks. Evolution can discover control programs that use their sensory-motor experience to fine-tune their policy parameters or alter their control logic on-the-fly while interacting with the environment. This enables the adaptive behaviors necessary to maintain near-optimal performance under changing environmental conditions. Unlike the original AMLZ, we go beyond toy tasks by tackling the simulator for the actual Laikago robot [4]. To facilitate this, we shifted away from the supervised learning paradigm of AMLZ. We show that evolved programs can adapt during

```
# wX: vector memory at address X.
def f(x, v, i):
  w0 = copy(v)
  w0[i] = 0
  w1 = abs(v)
  w1[0] = -0.858343 * norm(w2)
  w2 = w0 * w0
  return log(x), w1

# sX: scalar memory at address X.
# vX: vector memory at address X.
# obs, action: observation and action vectors.
def GetAction(obs, action):
  if s13 < s15: s5 = -0.920261 * s15
  if s15 < s12: s8, v14, i13 = 0, min(v8, sqrt(min(0, v3))), -1
  if s1 < s7: s7, action = f(s12, v0, i8)
  action = heaviside(v12)
  if s13 < s2: s15, v3 = f(s10, v7, i2)
  if s2 < s0: s11, v9, i13 = 0, 0, -1
  s7 = arcsin(s15)
  if s1 < s13: s3 = -0.920261 * s13
  s12 = dot(v3, obs)
  s1, s3, s15 = maximum(s3, s5), cos(s3), 0.947679 * s2
  if s2 < s8: s5, v13, i5 = 0, min(v3, sqrt(min(0, v13))), -1
  if s6 < s0: s15, v9, i11 = 0, 0, -1
  if s2 < s3: s2, v7 = f3(s8, v12, i1)
  if s1 < s6: s13, v14, i3 = 0, min(v8, sqrt(min(0, v0))), -1
  if s13 < s2: s7 = -0.920261 * s2
  if s0 < s1: s3 = -0.920261 * s1
  if s7 < s1: s8, action = f(s5, v15, i3)
  if s0 < s13: s5, v7 = f(s15, v7, i15)
  s2 = s10 + s3
  if s7 < s12: s11, v13 = f(s9, v15, i5)
  if s4 < s11: s0, v9, i13 = 0, 0, -1
  s10, action[i5] = sqrt(s7), s6
  if s7 < s9: s15 = 0
  if s14 < s11: s3 = -0.920261 * s11
  if s8 < s5: s10, v15, i1 = 0, min(v13, sqrt(min(0, v0))), -1
  return action
```

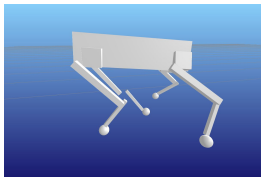


Fig. 1: Automatically discovered Python code representing an adaptable policy for a realistic quadruped robot simulator (top–right inset). This evolved policy outperforms MLP and LSTM baselines when a random leg is suddenly broken at a random time. (Lines in red will be discussed in the text).

Full version: https://arxiv.org/abs/2307.16890
Videos: https://youtu.be/sEFP1Hay4nE
Correspondence: spkelly@mcmaster.ca

their lifetime without explicitly receiving any supervised input (such as a reward signal). Furthermore, while AMLZ relied on the hand-crafted application of three discovered functions, we allow the number of functions used in the evolved programs to be determined by the evolutionary process itself. To do this, we use conditional automatically defined functions (CADFs) and demonstrate their impact. With this approach, we find that evolved adaptable policies are significantly simpler than state-of-the-art solutions from the literature because evolutionary search begins with minimal programs and incrementally adds complexity through interaction with the task domain. Their behavior is highly interpretable as a result.

In the quadruped robot, ARZ is able to evolve adaptable policies that maintain forward locomotion and avoid falling, even when all motors on a randomly selected leg fail to generate any torque, effectively turning the leg into a passive double pendulum. In contrast, despite comprehensive hyperparameter tuning and being trained with state-of-the-art reinforcement learning methods, MLP and LSTM baselines are unable to learn robust behaviors under such challenging conditions.

While the quadruped is a realistic complex task, simulating the real robot is time-consuming. Due to the lack of efficient yet challenging benchmarks for adaptive control, we created a toy adaptation task dubbed *Cataclysmic Cartpole* and repeated our analysis on this task with similar findings. In both cases, we provide a detailed analysis of evolved control programs to explain how they work, something notoriously difficult with *black box* neural network representations.

In summary, this paper develops an evolutionary method for the automated discovery of adaptable robotic policies from scratch. We applied the method to two tasks in which adaptation is critical, Quadruped Leg-Breaking and Cataclysmic Cartpole. On each task, the resulting policies:

- surpass carefully-trained MLP and LSTM baselines;
- are represented as interpretable, symbolic programs; and
- use fewer parameters and operations than the baselines.

These points are demonstrated for each task in Section V.

II. Related Work

Early demonstrations of Genetic Programming (GP) established its power to evolve optimal nonlinear control policies from scratch that were also simple and interpretable [5]. More recently, GP has been used to distill the behavior of complex neural network policies developed with Deep Reinforcement Learning into interpretable and explainable programs without sacrificing control quality [6]. In this work, we extend these methods to evolve programs that can change their behavior in response to a changing environment.

We demonstrate how to automatically discover a controller that can *context switch* between distinct behavior modes when it encounters diverse tasks, thus avoiding trade-offs associated with generalization across diverse environmental physics. If we can anticipate the nature of the environmental change a robot is likely to encounter, we can simulate environments *similar* to the expected changes and focus on building multitask control policies [2], [7]. In this case, some form of domain randomization [8] is typically employed to expose candidate policies to a breadth of task dynamics. However, policies trained with domain randomization often trade optimality in any particular environment dynamics for generality across a breadth of dynamics. This is the problem we aim to address with ARZ. Unlike previous studies in learning quadruped locomotion in the presence of non-stationary morphologies (e.g., [9]), we are specifically interested in how controllers can be automatically built from scratch without requiring any prior task decomposition or curriculum learning. This alleviates some burden on robotics engineers and reduces researcher bias toward known machine learning algorithms, opening the possibility for a complex adaptive system to discover something new.

In addition to anticipated non-stationary dynamics, another important class of adaptation tasks in robotics is sim-to-real transfer [11], where the robot needs to adapt policies trained in simulation to unanticipated characteristics of the real-world. Successful approaches to learn adaptive policies can be categorized by three broad areas of innovation: (1) New adaptation operators that allow policies to quickly tune their model parameters within a small number of interactions [10], [11], [12], [13]; (2) Modular policy structures that separate the policy from the adaptation algorithm and/or world model, allowing *both* to be learned [14], [15], [16], [17]; and (3) Hierarchical methods that allow a diverse set of complete or partial behaviors to be dynamically switched in and out of use at run-time, adapting by selecting the best strategy for the current environmental situation [9], [2], [18]. These algorithmic models of behavioral plasticity, modular structure, and hierarchical representations reflect the fundamental properties of meta-learning. In nature, these properties emerged through adaptation at two timescales (evolution and lifetime learning) [19]. ARZ makes these two time scales explicit by implementing an evolutionary search loop that acts on a "genome" of code, and an evaluation that steps through an episode which is analogous to the "lifetime" of the robot.

III. Methods

A. Algorithm Representation

As in the original AutoML-Zero [3], policies are represented as linear register machines that act on virtual memory [20]. In this work, we support four types of memory: scalar, vector, matrix, and index (*e.g.* `s1`, `v1`, `m1`, `i1`). Scalar, vector, and matrix memory are floating-point, while index memory stores integers. Algorithms are composed of two core functions: `StartEpisode()` and `GetAction()`. `StartEpisode()` runs once at the start of each episode of interaction with the environment. Its sole purpose is to initialize the contents of virtual memory with evolved constants. The content of these memories at any point in time can be characterized as the control program's state. Our goal is to discover algorithms that can adapt by tuning their memory state or altering their control code on-the-fly while interacting with their environment. This adaptation, as well as the algorithm's decision-making policy, are implemented

by the `GetAction()` function, in which each instruction executes a single operation (*e.g.*`s0=s7*s1` or `s3=v1[i2]`). We define a large library of operations (Table S2) and place no bounds on the complexity of programs. Evolutionary search is employed to discover what sequence of operations and associated memory addresses appear in the `GetAction()` function.

Conditional Automatically Defined Functions: In addition to `StartEpisode()` and `GetAction()`, up to 6 Conditionally-invoked Automatically Defined Functions [21] (CADFs) may be generated in an algorithm. Each CADF represents an additional function block, itself automatically discovered, which is callable from `GetAction()`. Since each CADF is conditionally invoked, the sequence of CADFs executed at each timestep throughout an episode is dynamic. This property is advantageous for multi-task learning and adaptation because programs that can switch control code in and out of the execution path on-the-fly are able to dynamically integrate general, re-useable code for related tasks *and* specialized code for disjoint tasks. We demonstrate in Section IV how this improves performance for the quadruped task. Each CADF receives 4 scalars, 2 vectors, and 2 indices as input, and execution of the function is conditional on a `<` comparison of the first 2 scalars (a configuration chosen for simplicity). The set of operations available is identical to `GetAction()` except that CADFs may not call each other to avoid infinite recursion. Each CADF uses its own local memory of the same size and dimensionality as the main memory used by `Setup()` and `GetAction()`. Their memory is initialized to zero at the start of each episode and is persistent across timesteps, allowing functions to integrate variables over time. Post-execution, the CADF returns the single most recently written index, scalar, and vector from its local memory.

The policy-environment interface and evaluation procedure are illustrated in Fig. 2. Sections V-A and V-B provide examples of evolved programs in this representation for the quadruped robot and Cataclysmic Cartpole task, respectively.

B. Evolutionary Search

Two evolutionary algorithms are employed in this work: Multi-objective search with the Nondominated Sorting genetic algorithm II (NSGA-II) [22] and single-objective search with Regularized evolution (RegEvo) [23], [3]. Both search algorithms iteratively update a population of candidate control programs using an algorithmic model of the Darwinian principle of natural selection. The generic steps for evolutionary search are:

1) Initialize a population of random control programs.
2) Evaluate each program in the task (Fig. 2).
3) Select promising programs using a task-specific fitness metric (See Fig. 2 caption).
4) Modify selected individuals through crossover and then mutation (Fig. S1).
5) Insert new programs into the population, replacing some proportion of existing individuals.
6) Go to step 2.

```
# StartEpisode = initialization code.
# GetAction = control algorithm.
# Sim = simulation environment.
# episodes = number of evaluation episodes.
# sX/vX/mX/iX: scalar/vector/matrix/index memory
# at address X.
def EvaluateFitness(StartEpisode, GetAction):
  sum_reward = 0
  for e in episodes:
    reward = 0
    steps = 0
    # Initialize sX/vX/mX with evolved parameters.
    # iX is initialized to zero.
    StartEpisode()
    # Set environment initial conditions.
    state = Sim.Reset()
    while (!Sim.Terminal()):
      # Copy state to memory, will be accessible
      # to GetAction.
      v1 = state
      # Execute action-prediction instructions.
      GetAction(state)
      if Sim.NumAction() > 1:
        action = v4
      else:
        action = s3
      state = Sim.Update(action)
      reward += Reward(state, action)
      steps += 1
    sum_reward += reward
    sum_steps += steps
  return sum_reward/episodes, sum_steps/episodes
```

Fig. 2: Evaluation process for an evolved control algorithm. The single-objective evolutionary search uses the mean episodic reward as the algorithm's *fitness*, while the multi-objective search optimizes two fitness metrics: mean reward (first return value) and mean steps per episode (second return value).

For the purposes of this study, the most significant difference between NSGA-II and RegEvo is their selection method. NSGA-II identifies promising individuals using multiple fitness metrics (e.g., forward motion *and* stability) while RegEvo selects based on a single metric (forward motion). Both search methods simultaneously evolve: (1) Initial algorithm parameters (*i.e.* initial values in floating-point memory `sX`, `vX`, `mX`), which are set by `StartEpisode()`; and (2) Program content of the `GetAction()` function and CADFs.

1) Multi-Objective Search: In the Quadruped robot tasks, the goal is to build a controller that continuously walks at a desired pace in the presence of motor malfunctions. It is critical that real-world robots avoid damage associated with falling, and the simplest way for a robot to achieve this is by standing relatively still and not attempting to move forward after it detects damage. As such, this domain is well suited to multi-objective search because walking in the presence of unpredictable dynamics while maintaining stability are conflicting objectives that must be optimized simultaneously. In this work, we show how NSGA-II maintains a diverse population of control algorithms covering a spectrum of trade-offs between forward motion and stability. From this diverse population of partial solutions, or *building blocks*, evolutionary search operators (mutation and cross-over) can build policies that are competent in both objectives. NSGA-II objective functions and constraints for the quadruped robot

task are discussed in Section IV.

2) Single-Objective Search: The Cataclysmic Cartpole task provides a challenging adaptation benchmark environment without the safety constraints and simulation overhead of the real-world robotics task. To further simplify our study of adaptation and reduce experiment time in this task, we adopt the RegEvo search algorithm and optimize it for fast experimentation. Unlike NSGA-II, asynchronous parallel workers in RegEvo also perform selection, which eliminates the bottleneck of waiting for the entire population to be evaluated prior to ranking, selecting, and modifying individuals.

Crossover and Mutation Operators: We use a simple crossover operator that swaps a randomly selected CADF between two parent algorithms. Since all CADFs have the same argument list and return value format, no signature matching is required to select crossover points. If either parent algorithm contains no CADFs, one randomly selected parent is returned. Post-crossover, the child program is subject to stochastic mutation, which adds, removes, or modifies code using operators listed in Table S1.

C. Algorithm Configurations and Baselines

Temporal memory is the primary mental system that allows an organism to change, learn, or adapt during its lifetime. In order to predict the best action for a given situation in a dynamic environment, the policy must be able to compare the current situation with past situations and actions. This is because generating an appropriate action depends on the current state *and* a prediction of how the environment is changing. Our evolved algorithms are able to adapt partly because they are **stateful**: the contents of their memory (`sX`, `vX`, `mX`, and `iX`) are persistent across timesteps of an episode.

We compare ARZ against stateless and stateful baselines. These policy architectures consist, respectively, of multilayer perceptrons (**MLP**) and long short-term memory (**LSTM**) networks whose parameters to be optimized are purely continuous. Therefore, we use Augmented Random Search (ARS) [24], which is a state-of-the-art continuous optimizer and has been shown to be particularly effective in learning robot locomotion tasks [12], [25]. In comparison, Proximal Policy Optimization [26] underperformed significantly; we omit the results and leave investigation for future work. All methods were allowed to train until convergence with details in Supplement S1-A.

IV. Non-Stationary Task Domains

We consider two different environments: a realistic simulator for a quadruped robot and the novel Cataclysmic Cartpole. In both cases, policies must handle changes in the environment's transition function that would normally impede their proper function. These changes might be sudden or gradual, and no sensor input is provided to indicate when a change is occurring or how the environment is changing.

A. Quadruped Robot

We use the Tiny Differentiable Simulator [27] to simulate the Unitree Laikago robot [4]. It is a quadruped robot with 3 actuated degrees of freedom per leg. Thus the action space has 12-dimensional real values corresponding to desired motor angles. A Proportional-Derivative controller is used to track these desired angles. The observation space includes 37 real values describing the angle and velocity for each joint as well as the position, orientation, and velocity of the robot body. Each episode begins with the robot in a stable upright position and continues for a maximum of 1000 timesteps (10 seconds). Each action suggested by the policy is repeated for 10 consecutive steps.

The goal of the non-stationary quadruped task is to move forward (x-axis) at 1.0 meters/second. Adaptation must handle sudden *leg-breaking* in which all joints on a single, randomly selected leg suddenly become passive at a random time within each episode. The leg effectively becomes a double pendulum for the remainder of the episode. The episode will terminate early if the robot falls and this results in less return. We design the following reward function:

$$r(t) = 1.0 \quad 2 \quad |v(t) \quad v| \quad ||a(t) \quad a(t \quad 1)||_2; \quad (1)$$

where the first term 1.0 is the survival bonus, v is the target forward velocity of 1 m/s, $v(t)$ is the robot's current forward velocity, and $a(t)$ and $a(t \quad 1)$ are the policy's current and previous action vectors. This reward function is *shaped* to encourage the robot to walk at a constant speed for as long as possible while alleviating motor stress by minimizing the change in the joint acceleration. In the context of multi-objective search, maximizing the mean of Equation 1 over a maximum of 1000 timesteps is Objective 1. To discourage behaviors that deviate too much along the y-axis, we terminate an episode if the robot's y-axis location exceeds $\quad 3.0$ meters. Objective 2 is simply the number of timesteps the robot was able to survive without falling or reaching this y-axis threshold. Importantly, we are not interested in policies that simply stand still. Thus, if Objective 2 is greater than 400 *and* Objective 1 is less than 50, both fitnesses are set to 0. As shown in Fig. S2, these fitness constraints eliminate policies that would otherwise persist in the population without contributing to progress on the forward motion objective.

B. Cataclysmic Cartpole Environment

To study the nature of adaptation in more detail, we introduce a new, highly challenging but computationally simple domain called Cataclysmic Cartpole in which multiple aspects of the classic Cartpole ([28]) physics are made dynamic. Adaptation must handle the following non-stationary properties:

Track Angle: The track tilts to a random angle at a random time. Because the robot's frame of reference for the pole angle ( ) is relative to the cart, it must figure out the new direction of gravity and *desired* value of to maintain balance, and respond quickly enough to keep the pole balanced. The track angle is variable in [-15, 15] degrees. This simulates a change in the external environment.

Force: A force multiplier f is applied to the policy's action such that its actuator strength may increase or

decrease over time. The policy's effective action is f action, where f changes over time within the range [0.5, 2]. This simulates a drop in actuator strength due to a low battery, for example.

Damping: A damping factor D simulates variable joint friction by modifying joint torque as $_D = Dq_f$, where q_f is the joint velocity (see eqns. 2.81, 2.83 in [29]). This simulates joint wear and tear. D changes over time in the range [0.0, 0.15].

Each type of change is controlled by a single parameter. We investigate two schedules for how these parameters might change during an episode, illustrated in Fig. S4.

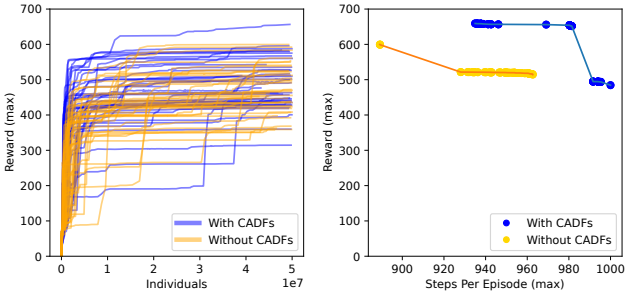
V. Results

A. Quadruped Leg-Breaking

1) Comparison with Baselines: ARZ—with the inclusion of CADFs—is the only method that produced a viable control policy in the leg-breaking task. This problem is exceedingly difficult: finding a policy that maintains smooth locomotion and is robust to leg breaking requires 20 evolution experiment repetitions (Fitness > 600 in Fig. 3a). In Fig. 3a, training fitness between 500 and 600 typically indicates either (1) a viable forward gait behavior that is only robust to 3/4 legs breaking or (2) a policy robust to *any* leg breaking but which operates at a high frequency not viable for a real robot, with its reward being significantly penalized by fitness shaping as a result. Within the single best repeat, the NSGA-II search algorithm produces a variety of policies with performance trade-offs between smooth forward locomotion (reward objective) and stability (steps objective), Fig. 3b. From this final set of individuals, we select a single policy to compare with the single best policy from each baseline. Due to practical wall-clock time limits, we were only able to train both ARS+MLP and ARS+LSTM policies up to 10^6 trials in total, but found that under this sample limit, even the best ARS policy only achieved a reward of 360, much lower than the 570 found by the best ARZ policy, suggesting that ARZ can even be more sample efficient than standard neural network baselines.

Fig. 4 confirms that ARZ is the only method capable of building a controller that is robust to multiple different legs breaking mid-episode. We plot post-training test results for one champion ARZ policy in comparison with the single-best controller discovered by ARS+MLP and ARS+LSTM. ARZ's adaption quality (as measured by mean reward) is superior to baselines in the case of each individual leg, and its performance on the stationary task (See "None" in Fig. 4) is significantly better than any other method. Interestingly, Fig. 4 indicates that the MLP also learned a policy that is robust to the specific case of the back-right leg breaking. Unlike ARZ, it is unable to generalize this adaptation to any other leg. Finally, while the LSTM policy performed better than the MLP on the stationary task, it fails to adapt to any of the leg-breaking scenarios.

Visualizing trajectories for a sample of 5 test episodes from Fig. 4 confirms that the ARZ policy is the only controller



(a) Evolution progress (b) Best Pareto fronts

Fig. 3: CADFs speed up evolution on average and produced the best final result. (a) shows ARZ search data recorded over 20 independent repeats with and without the use of CADFs. The horizontal axis for (a) shows the total number of individual programs evaluated, while the vertical axis shows mean return (Equation 1) over 32 episodes for the single best individual discovered so far. (b) shows Pareto fronts for the single repeats with max reward from each experiment. Each point in (b) represents the bi-objective fitness of one control program.

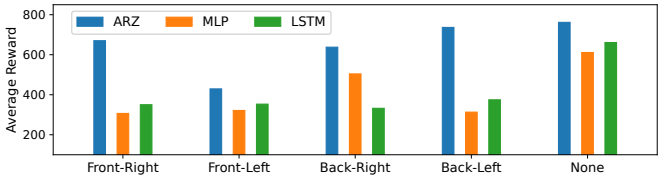
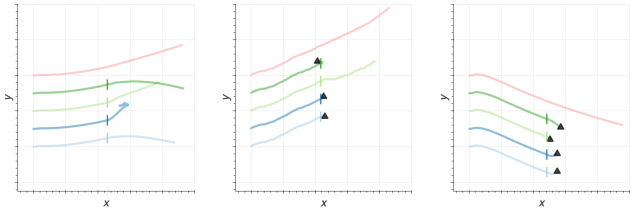


Fig. 4: ARZ discovers the only policy that can adapt to any leg breaking. The plot shows test results for the single best policy from ARZ and ARS baselines (MLP and LSTM) in the *mid-episode leg-breaking* task. For each leg, bars show mean reward over 100 episodes in which that leg is broken at a randomly selected timestep. A reward < 400 in any column indicates the majority of test episodes for that leg ended with a fall.

that can avoid falling in all scenarios, although in the case of the front-left leg breaking, it has trouble maintaining forward motion, Fig. 5. This is reflected in its relatively weak test reward for the front-left leg (See Fig. 4). The MLP policy manages to keep walking with a broken back-right leg but falls in all other dynamic tasks. The LSTM, finally, is only able to avoid falling in the stationary task in which all legs are reliable.



(a) ARZ (b) MLP (c) LSTM

Fig. 5: ARZ discovers the only policy that consistently avoids falling. Plot shows sample trajectories in each leg-breaking task. The vertical bar indicates the change point (step 500). ▲ indicates that the robot fell over. Each plot shows 4 test episodes in which a unique leg breaks. From top to bottom, the affected legs are: None, Back-Left, Back-Right, Front-Left, Front-Right.

2) On Simplicity and Interpretability: The policy for the Quadruped Leg-Breaking task discovered by evolutionary search is presented in Fig. 1. This algorithm uses 608 parameters and can be expressed in less than 40 lines of code,

executing at most 2080 floating point operations (FLOPs) per step. This should be contrasted with the number of parameters and FLOPs expended in the baseline MLP/LSTM models, which use more than 2.5k/9k parameters and 5k/18k FLOPs per step, respectively. A detailed account of how these numbers were obtained can be found in Section S4. We note that each function possesses its own variables and memory, which persists throughout the run. The initialization value for the variables are tuned for the `GetAction` function, thus counted as parameters, while they are all set to zero for `f`.

Here we provide an initial analysis of the ARZ policy, leaving a full analysis and interpretation of the algorithm to future work. The key feature of the algorithm is that it discretizes the input into four states, and the action of the quadruped is completely determined by its internal state and the discrete label. The temporal transitions of the discretized states show a stable periodic motion when the leg is not broken, and the leg-breaking introduces a clear disruption in this pattern, as shown in Fig. 6. This being a stateful algorithm with multiple variables accumulating and preserving variables from previous steps, we conjecture that the temporal pattern of the discrete states serves as a signal for the adaptive behavior of the quadruped.

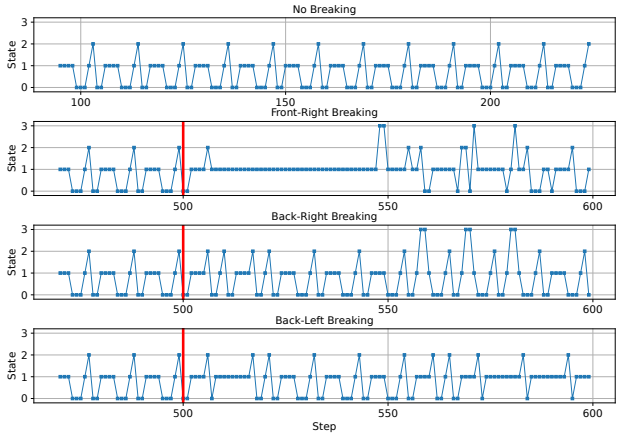


Fig. 6: State trajectories of various leg-breaking patterns. The leg-breaking event is marked by a vertical red line. Note that different leg breaking patterns result in different state trajectories. We conjecture that these trajectories serve as signals that trigger the adaptive response in the algorithm.

We now expand upon how the continuous input signal is discretized in the ARZ algorithm presented in Fig. 1. We first observe that the only way the incoming observation vector interacts with the rest of the algorithm is by forming scalar `s12`, by taking an inner-product with a dynamical vector `v3` (the second of the three red-colored lines of code). The scalar `s12` affects the action only through the two `if` statements colored in red. Thus the effect of the input observation on the action is entirely determined by the relative position of the scalar `s12` with respect to the two decision boundaries set by the scalars `s15` and `s7`. In other words, the external input of the observation to the system is effectively discretized into four states: 0 (`s12 s15, s7`), 1 (`s15, s7 < s12`), 2 (`s7 < s12 s15`) or 3 (`s15 < s12 s7`).

Thus external changes in the environment, such as leg breaking, can be accurately detected by the change in the pattern of the state trajectory, because the variables `s7` and `s15` defining the decision boundary of the states form a stable periodic function in time. We demonstrate this in Fig. 7, where we plot the values of the three scalars `s12`, `s15` and `s7` for front-leg breaking, whose occurrence is marked by the vertical red line. Despite the marked change of behavior of the input `s12` after leg-breaking, we see that the behavior of the two scalars `s7` and `s15` are only marginally affected. Intriguingly, the behavior of the scalar registers `s7` and `s15` resemble that of central pattern generators in biological circuits responsible for generating rhythmic movements [30].

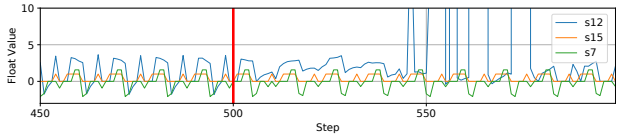
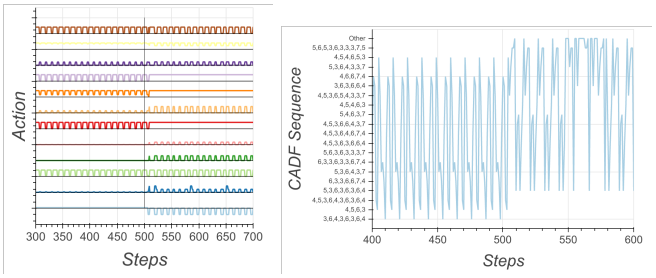


Fig. 7: The scalar values `s12`, `s15` and `s7` of the quadruped during front-leg breaking. Note the consistent periodic behavior of the scalars `s15` and `s7` despite leg breaking, marked by the vertical red line. The same periodicity is observed for all leg-breaking scenarios analyzed.

The policy's ability to quickly identify and adapt to multiple unique failure conditions is clear in Fig. 8a, which plots the controller's actions one second before and after a leg breaks. We see a clear, instantaneous change in behavior when a leg fails. This policy is able to identify when a change has occurred and rapidly adapt. Fig. 8b shows the particular sequence of CADFs executed at each timestep before and after the change, indicating that CADFs do play a role in the policy's ability to rapidly adjust its behavior. Indeed, only evolutionary runs that included CADFs were able to discover a policy robust to any leg breaking.



(a) Actions (b) CADF call sequences

Fig. 8: ARZ policy behavior changes when Front-Left leg breaks mid-episode (step 500), as shown by the dynamics of the actions and the program control flow due to CADFs.

B. *Cataclysmic Cartpole*

Introducing a novel benchmark adaptation task is an informative addition to results in the realistic quadruped simulator because we can empirically adjust the nature of the benchmark dynamics until they are significant enough to create an *adaptation gap*: when stateless policies (i.e., MLP generalists) fail to perform well because they cannot adapt their control policy in the non-stationary environment (See Section S2

for details.). Having confirmed that Cataclysmic Cartpole requires adaptation, we only examine stateful policies in this task.

1) Comparison with Baselines: In Cataclysmic Cartpole, we confirm that ARZ produces superior control relative to the (stateful) ARS+LSTM baseline in tasks with a sudden, dramatic change. Fig. 9 and 10 show testing that was done after the search is complete. A fitness score of 800 indicates the policy managed to balance the pole for 800 timesteps, surviving up to the last point in an episode with any active dynamics (See Fig. S4). "Stationary" is the standard Cartpole task while "Force", "Damping", and "Track Angle" refer to Cartpole with sudden or continuous change in these parameters *only* (See Section IV-B). "All" is the case where all change parameters are potentially changing simultaneously. Legends indicate the policy type and corresponding task type used during evolution. First, note that strong adaptable policies do not emerge from ARZ or ARS+LSTM evolved in the stationary task alone (See ARZ [Stationary] and LSTM [Stationary]), implying that proficiency in the stationary task does not directly transfer to any non-stationary configuration. However, when exposed to non-stationary properties during the search, ARZ and ARS+LSTM discover policies that adapt to all sudden and continuous non-stationary tasks. ARZ is significantly more proficient in the sudden change tasks (Fig. 10), achieving near perfect scores of 1000 in all tasks. In continuous change, the single best LSTM policy achieves the best multitasking performance with a stronger score than ARZ on the Track Angle problem, and it is at least as proficient as ARZ on all other tasks. However, unlike the LSTM network, ARZ policies are uniquely interpretable.

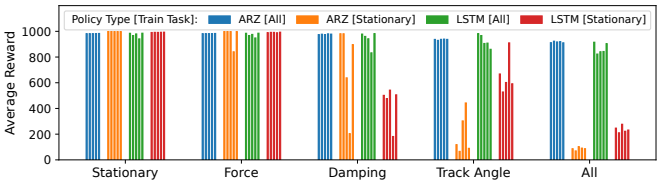


Fig. 9: Post-evolution test results in the Cataclysmic Cartpole *continuous-change* task. Legend indicates policy type and search task. `[All]` marks policies exposed to all tasks during evolution. ARZ and LSTM both solve this adaptation task, and no direct transfer from stationary tasks to dynamic tasks is observed. The best 5 policies from each experiment are shown.

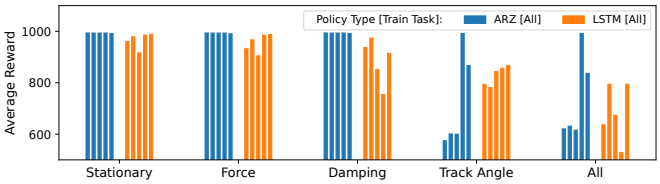


Fig. 10: Post-evolution test results in the Cataclysmic Cartpole *sudden-change* task. `[All]` marks policies exposed to all tasks during evolution. ARZ discovers the only policy that adapts to all sudden-change Cataclysmic Cartpole tasks. The best 5 policies from each experiment are shown.

2) On Simplicity and Interpretability: Here we decompose an ARZ policy to provide a detailed explanation of how it integrates state observations over time to compute optimal actions in a changing environment. An example of an algorithm discovered in the ARZ [All] setting of Fig. 9 is presented in Fig. 11. Note that CADFs were not required to solve this task and have thus been omitted from the search space in order to simplify program analysis. What we find are three accumulators that collect the history of observation and action values from which the current action can be inferred.

```
# sX: scalar memory at address X.
# obs: vector [x, theta, x_dot, theta_dot].
# a, b, c: fixed scalar parameters.
# V, W: 4-dimensional vector parameters.
def GetAction(obs, action):
  s0 = a * s2 + action
  s1 = s0 + s1 + b * action + dot(V, obs)
  s2 = s0 + c * s1
  action = s0 + dot(obs, W)
  return action
```

Fig. 11: Sample stateful action function evolved on the task where all parameters are subject to continuous change (ARZ [All] in Fig. 9). Code shown in Python.

This algorithm uses 11 variables and executes 25 FLOPs per step. Meanwhile, the MLP and LSTM counterparts use more than 1k and 4.5k parameters, expending more than 2k and 9k FLOPs per step, respectively. More details for this computation are presented section S4.

There are two useful ways to view this algorithm. First, by organizing the values of `s0`, `s1`, and `s2` at step n into a vector Z_n, which can be interpreted as a vector in latent space of $d = 3$ dimensions, we find that the algorithm can be expressed in the form: $s_{n+1} = \text{concat}(\text{obs}_{n+1};\ \text{act}_n)$; $Z_{n+1} = \widetilde{U}\ Z_n + \widetilde{P}\ s_{n+1}$; $\text{act}_{n+1} = \widetilde{A}^T\ Z_{n+1} + \widetilde{W}^T\ s_{n+1}$, with the projection matrix $\widetilde{P}$ that projects the state vector to the latent space, and a $d \quad d$ evolution matrix $\widetilde{U}$. This is a linear recurrent neural network with internal state Z_n. The second way to view the algorithm is to interpret it as a generalization of a proportional–integral–derivative (PID) controller. This can be done by first explicitly solving the recurrent equations presented above and taking the continuous limit. Introducing a single five-dimensional state vector $s(t) = [x(t);\ \ (t);\ \underline{x}(t);\ \_(t);\ \text{act}(t)]$, and d-dimensional vectors u, v, and w, a five-dimensional vector p and a constant term c, the algorithm in the continuous time limit can be written in the form: $\text{act}(t) = c + w^T\ \ U^t\ \ u + p^T\ \ s(t) + v^T\ \ \int_0^t d\ \ U^t\ \ P\ \ s(\ )$ where P and U are the continuous-time versions of $\widetilde{P}$ and $\widetilde{U}$. In our particular discovered algorithm (Fig. 11), d happens to be 3. Notice that the integration measure now has a time-dependent weight factor in the integrand versus the conventional PID controller. Further derivations, discussions, and interpretations regarding this algorithm are presented in the supplementary material.

VI. Conclusion and Discussion

We have shown that using ARZ to search simultaneously in *program space* and *parameter space* produces proficient, simple, and interpretable control algorithms that can perform zero-shot adaptation, rapidly changing their behavior to maintain near-optimal control in environments that undergo

radical change. In the remainder of this section, we briefly motivate and speculate about future work.

CADFs and the Distraction Dilemma. In the quadruped robot domain, we have observed that including Conditionally invoked Automatically Defined Functions (CADFs) in our search space improves the expressiveness of evolved control algorithms. In the single best policy, CADFs have been used to discretize the observation space into four states. The action is then completely determined by the internal state of the system and this discretized observation. One interpretation is that this discretization helps the policy define a switching behavior that can overcome the *distraction dilemma*: the challenge for a multi-task policy to balance the reward of excelling at multiple different tasks against the ultimate goal of achieving generalization [1]. By contrast, searching only in the parameter space of a hand-designed MLP or LSTM network did not produce policies that can adapt to more than one unique change event (i.e., a single leg breaking). A deeper study of modular/hierarchical policies and their impact on the distraction dilemma is left to future work.

The Cataclysmic Cartpole Task. Given the computationally intensive nature of simulating a real robot, we felt compelled to also include a more manageable toy task where adaptation matters. This led to the *Cataclysmic Cartpole* task. We found it useful for doing quick experiments and emphasizing the power and interpretability of ARZ results. We hope that it may also provide an easily reproducible environment for use in further research.

Adapting to Unseen Task Dynamics. Looking to the future, we have included detailed supplementary material which raises an open and ambitious question: how can we build adaptive control policies without any prior knowledge about what type of environmental change may occur in the future? Surprisingly, preliminary results with ARZ on the cataclysmic cartpole task suggest that injecting partial-observability and dynamic actuator noise during evolution (training) can act as a general surrogate for non-stationary task dynamics S2. In preliminary work, we found this to support the emergence of policies that can adapt to novel task dynamics that were *not* experienced during search (evolution). This was not possible for our LSTM baselines. If true, this would be significant because it implies we might be able to evolve proficient control policies without complete prior knowledge of their task environment dynamics, thus relaxing the need for an accurate physics simulator. Future work may investigate the robustness of this preliminary finding.

Author Contributions

SK and ER led the project. ER and JT conceived the project and acted as principal advisors. All authors contributed to the methodology. SK, MM, PN, and DP ran the evolution experiments. XS ran the baselines. MM and DP analysed the algorithms. SK, DP, and MM wrote the paper. All authors edited the paper.

Acknowledgements

We would like to thank Wenhao Yu, Chen Liang, Sehoon Ha, James Lee and the Google Brain Evolution and AutoML groups for technical discussions; Erwin Coumans for physics simulations advice; Erwin Coumans, Kevin Yap, Jacob Budzis, Heng Li, Kaiyuan Wang, and Ryan Gillard for code contributions; and Quoc V. Le, Vincent Vanhoucke, Ed Chi, and Erik Goodman for guidance and support.

References

[1] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt, "Multi-Task Deep Reinforcement [...]]." AAAI, 2019.
[2] S. Kelly, T. Voegerl, W. Banzhaf, and C. Gondro, "Evolving hierarchical memory-prediction [...]]," *Genet. Program. Evolvable Mach.*, 2021.
[3] E. Real, C. Liang, D. R. So, and Q. V. Le, "AutoML-Zero: Evolving Machine Learning Algorithms From Scratch," *ICML*, 2020.
[4] "Unitree Robotics." [Online]. Available: http://www.unitree.cc/
[5] J. R. Koza and M. A. Keane, "Genetic breeding of non-linear optimal control strategies [...]]," in *Analysis and Optimization of Systems*, 1990.
[6] Y. Dhebar, K. Deb, S. Nageshrao, L. Zhu, and D. Filev, "Toward Interpretable-AI Policies [...]]," *IEEE. Trans. Cybern.*, 2022.
[7] W. Yu, J. Tan, C. K. Liu, and G. Turk, "Preparing for the unknown: Learning a universal policy [...]," in *RSS*, 2017.
[8] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring [...]]," *CoRR*, 2017.
[9] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *Nature*, 2015.
[10] X. Song, Y. Yang, K. Choromanski, K. Caluwaerts, W. Gao, C. Finn, and J. Tan, "Rapidly Adaptable Legged Robots [...]," in *IROS*, 2020.
[11] X. Song, W. Gao, Y. Yang, K. Choromanski, A. Pacchiano, and Y. Tang, "ES-MAML: simple hessian-free meta learning," in *ICLR*, 2020.
[12] W. Yu, J. Tan, Y. Bai, E. Coumans, and S. Ha, "Learning fast adaptation with meta strategy optimization," *IEEE Robot. Autom. Lett.*, 2020.
[13] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *ICML*, 2017.
[14] A. Kumar, Z. Fu, D. Pathak, and J. Malik, "RMA: rapid motor adaptation for legged robots," *CoRR*, 2021.
[15] E. Najarro and S. Risi, "Meta-Learning through Hebbian Plasticity in Random Networks," *CoRR*, 2020.
[16] D. Floreano and J. Urzelai, "Evolutionary robots with on-line self-organization and behavioral fitness," *Neural Networks*, 2000.
[17] T. Anne, J. Wilkinson, and Z. Li, "Meta-learning for fast adaptive locomotion with uncertainties [...]]," in *IROS*, 2021.
[18] A. Li, C. Florensa, I. Clavera, and P. Abbeel, "Sub-policy adaptation for hierarchical reinforcement learning," in *ICLR*, 2020.
[19] J. X. Wang, "Meta-learning in natural and artificial intelligence," *Current Opinion in Behavioral Sciences*, 2021.
[20] M. Brameier and W. Banzhaf, *Linear Genetic Programming*. Springer, 2007.
[21] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.
[22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic [...]," *IEEE Trans. Evol. Comput.*, 2002.
[23] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *AAAI*, 2019.
[24] H. Mania, A. Guy, and B. Recht, "Simple random search of static linear policies is competitive [...]]," in *NeurIPS*, 2018.
[25] K.-H. Lee, O. Nachum, T. Zhang, S. Guadarrama, J. Tan, and W. Yu, "PI-ARS: Accelerating Evolution-Learned [...]]," in *IROS*, 2022.
[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, 2017.
[27] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, "NeuralSim: Augmenting Differentiable [...]]," in *ICRA*, 2021.
[28] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.
[29] S. Sueda, "Analytically differentiable articulated [...]]," 2021. [Online]. Available: https://github.com/sueda/redmax/blob/master/notes.pdf
[30] E. Marder and D. Bucher, "Central pattern generators and the control of rhythmic movements," *Current Biology*, 2001.
[31] R. Gillard, S. Jonany, Y. Miao, M. Munn, C. de Souza, J. Dungay, C. Liang, D. R. So, Q. V. Le, and E. Real, "Unified functional hashing in automatic machine learning," *arXiv*, 2023.
[32] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, 2002.
[33] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *JMLR*, 2012.
[34] D. Hafner, J. Davidson, and V. Vanhoucke, "Tensorflow agents: Efficient batched reinforcement learning in tensorflow," *CoRR*, 2017.
[35] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016.

Supplementary Material

S1. Methods Additional Details

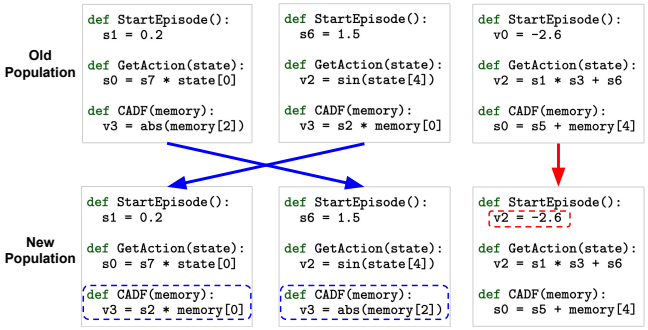


Fig. S1: Simplified example of a population of algorithms, modified via crossover and mutation to produce a new population. Complete list of mutation operators is provided in Table S1

A. Baseline Details

Augmented Random Search (ARS): We used a standard implementation from [24] and hyperparameter tuned over a cross product between:

learning rate: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5]
Gaussian standard deviation: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5]

and used a 2-layer MLP of hidden layer sizes (32,32) with Tanh non-linearity, along with an LSTM of size 32.

Proximal Policy Optimization (PPO): We used a standard implementation from TF-Agents [34], which we verified to reproduce standard Mujoco results from [26]. We varied the following hyperparameters:

nsteps ("collect sequence length"): [256, 1024]
learning rate: [5e-5, 1e-4, 5e-4, 1e-3, 5e-3]
entropy regularization: [0.0, 0.05, 0.1, 0.5]

and due to the use of a shared value network, we used a 2-layer MLP of hidden layer sizes (256, 256) with ReLU nonlinearity alongside an LSTM of size 256. Since PPO significantly underperformed (e.g., obtaining only 100 reward on quadruped tasks), we omitted its results in this paper to save space.

B. Quadruped Tasks

We perform 20 independent repeats for each method with unique random seeds. All repeats are allowed to train until convergence. NSGA-II uses parent and child population sizes of 100 and 1000, respectively. No search restarts or FEC are enabled. The set of operations available for inclusion in any program are listed in Table S2. For ARS experiments, we run a hyperparameter sweep consisting of 36 repeats with unique hyperparameters. We then run an additional 20 repeats using the best hyperparameter configuration.

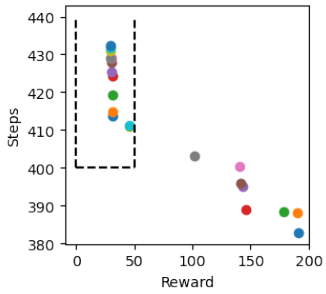


Fig. S2: A typical Pareto front early in NSGA-II search. The dashed box shows policies that are effectively eliminated through fitness constraints.

C. Cataclysmic Cartpole Tasks

Cartpole [28], [35] is a classic control task in which a pole is attached by an un-actuated joint to a cart that moves Left or Right along a frictionless track, Figure S3. The observable state of the system at each timestep, $\vec{s}(t)$, is described by 4 variables including the cart position (x), cart velocity ($\dot{x}$), pole angle relative to the cart (θ), and pole angular velocity ($\dot{\theta}$). We use a continuous-action version of the problem in which the system is controlled by applying a force $2\ [\ \ 1, 1]$ to the cart. The pole starts nearly upright, and the goal is to prevent it from falling over. An episode ends when the pole is more than 12 degrees from vertical, the cart moves more than 2.4 units from the center, or a time constraint is reached (1000 timesteps). A reward of $(1\quad j\theta_{vert}j/12)^2$ is provided for every timestep that the pole remains upright, where θ_{vert} is a fixed reference for the angle of the pole relative to the vertical plane. As such, the objective is to balance the pole close to vertical for as long as possible.

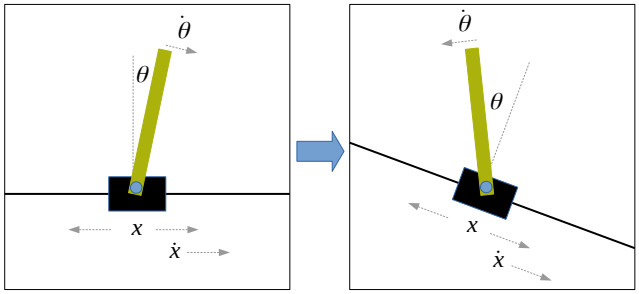


Fig. S3: Illustration of a track angle change in the Cataclysmic Cartpole task with the 4 variables in the state observation $\vec{s}(t)$. Note that θ always represents the angle between the pole and the line running perpendicular to the track and cart, thus the *desired* value of θ to maintain balance ($\theta_{vert} = 0$) changes with the track angle and is not directly observable to the policy.

1) Sudden: A sudden change in each change parameter occurs at a unique random timestep in [200, 800], Figure S4a.
2) Continuous: Each parameter changes over a window with random, independently chosen start and stop timesteps in [200, 800], Figure S4b.

For the ARZ methods, we execute 10 repeats of each experiment with unique random seeds. For ARS, we run a hyperparameter sweep consisting of 36 repeats with unique hyperparameters. In each case, we select 5 repeats with the best search fitness and test the single best policy from each. Plots show mean fitness over 100 episodes for each policy in each task.

S2. Additional Experiments: Cataclysmic Cartpole

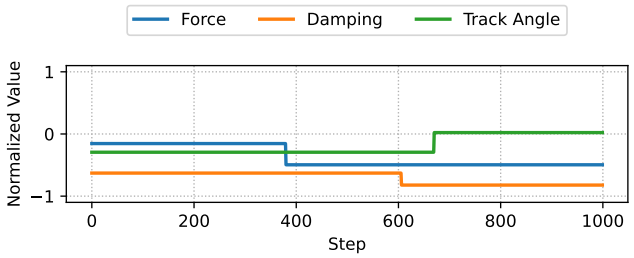
A. Adaptation Gap

In this section we use stateless policies (ARZ and MLP) to confirm that Cataclysmic Cartpole dynamics are significant enough to create an **adaptation gap**: when stateless policies (i.e. generalists) fail to perform well because they cannot adapt their control policy in the non-stationary environment. As mentioned in Section III-C our evolved algorithms are able to adapt partly because they are stateful: the contents of their memory (`sX`, `vX`, `mX`, and `iX`) are persistent across timesteps of an episode. The representation can easily support stateless algorithms simply by forcing the policy to wipe its memory content and re-initialize constants at the beginning of the `GetAction()` function (See Figure 2).

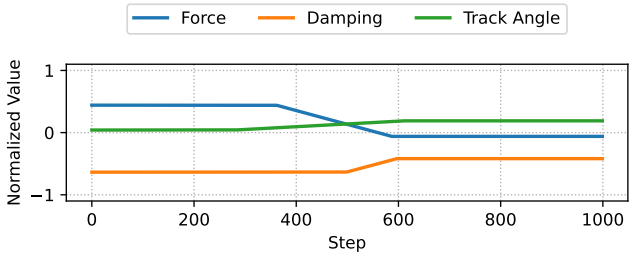
Fig. S5 indicates that, in the continuous change environment, the stateless baselines (MLP and ARZ stateless) fail to achieve sufficient fitness (800) when all types of change occur simultaneously (`ALL`). This confirms that the continuous change paradigm provides a suitably challenging non-stationary problem environments to study adaptation and life-long learning. In the sudden change task (Figure S6), the MLP baseline still fails. Surprizingly, ARZ can discover stateless policies that succeed under this type of non-stationarity.

Operator	Allowed Functions	Prob	Description
Insert Instruction	`GetAction() CADF()`	0.5	Insert randomly generated instruction at uniformly sampled line number
Delete Instruction	`GetAction() CADF()`	1.0	Delete the instruction at a uniformly sampled line number
Randomize Instruction	`GetAction() CADF()`	1.0	Randomize the instruction at a uniformly sampled line number
Randomize Function	`GetAction() CADF()`	0.1	Randomly shuffles all lines of code
Randomize constants	`StartEpisode()`	0.5	Modify a fraction (0.2) of uniformly sampled constants in a uniformly sampled instruction. For each constant, add noise sampled from $N(0, 0.05^2)$.
Randomize Parameter	`GetAction() CADF()`	0.5	Randomize a uniformly sampled parameter in a uniformly sampled instruction
Randomize dim indices	`GetAction() CADF()`	0.5	Randomize a fraction (0.2) of uniformly sampled dim indices in a uniformly sampled instruction. Each chosen dim index is set to a new integer uniformly sampled from $[0, dim)$ where dim is the size of the memory structure being referenced.

TABLE S1: Mutation operators. *Prob* column lists the relative probability of applying each operation. For example, the Delete Instruction op will be applied twice as often as the Insert instruction.



(a) Sudden



(b) Continuous

Fig. S4: A typical randomly-created change schedule.

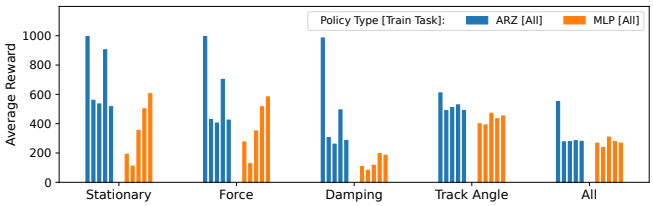


Fig. S5: Stateless baselines fail to achieve sufficient fitness (800) when all types of change occur simultaneously (`ALL`). The plot shows test results for **stateless baselines** in the Cataclysmic Cartpole **continuous change** tasks. The legend indicates policy type and search task. "Stationary" is the standard Cartpole task while "Force", "Damping", and "Track Angle" refer to Cartpole with continuous change in these parameters *only* (See Section IV-B). "All" is the case where all change parameters are potentially changing simultaneously. Y-axis is the average reward of 100 episodes in each task. See Section S2-A for discussion.

B. Adapting to Unseen Dynamics in Cataclysmic Cartpole

How can we build adaptive control policies without any prior knowledge about what type of environmental change might occur? Surprisingly, for ARZ, we find that injecting partial-observability and dynamic actuator noise during evolution (training) can act as a general surrogate for non-stationary task dynamics, supporting the emergence of policies that can adapt to novel task dynamics that were *not* experienced during evolution. This was not possible for our LSTM baselines. It is a significant finding that deserves more attention in future work because it implies we can potentially evolve proficient

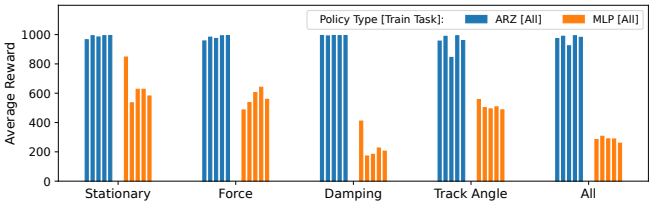


Fig. S6: ARZ can discover stateless policies that succeed in the sudden change tasks. The plot shows test results for **stateless baselines** in the Cartpole **sudden change** tasks. The legend indicates policy type and search task. "Stationary" is the standard Cartpole task while "Force", "Damping", and "Track Angle" refer to Cartpole with continuous change in these parameters *only* (See Section IV-B). "All" is the case where all change parameters are potentially changing simultaneously. Y-axis is the average reward of 100 episodes in each task. See Section S2-A for discussion.

control policies without complete prior knowledge of their task environment dynamics, thus relaxing the need for an accurate physics simulator.

If we assume that no simulator is available for any of the non-stationary tasks in Cataclysmic Cartpole (Force, Damping, Track Angle), can we still build policies that cope with these changes? From a policy's perspective, changes to the physics of the environment will (1) change the meaning of its sensor observations (e.g. pole angle sensor value (θ) corresponding to vertical suddenly changes); and/or (2) change the effect of its actions (e.g. a particular actuator value suddenly has a much greater effect on the cart's trajectory). To prepare policies for these uncertainties, we evolve them with non-stationary noise applied to their actions and introduce a partially-observable observation space. Specifically, we modify the task to add:

> Actuator Noise: Each action value v is modified such that $v = v + n$, where n is sampled from a Gaussian distribution with mean that varies in [-2, 2] following the continuous change schedule in Figure S4b.
> Partial Observability: Positional state variables (cart position (x) and pole angle relative to the cart (θ)) are set to zero prior to passing the state observation to the policy.

Our hypothesis is that this will encourage policies to rely less on their observations and actions, and as a result they might build a stronger, more dynamic internal world-model to predict how their actions will affect future states. That is, there is more pressure to model the environment's dynamic transition function. In Figure S7, `ARZ [PO + Act Noise]` shows test results for an ARZ experiment that uses the stationary task simulator during evolution (i.e. the unmodified Cartpole environment) but applies actuator noise and partial observability as described above. Remarkably, these evolved policies are able to adapt reasonably well under all non-stationary tasks in the Cataclysmic Cartpole environment, achieving an average reward of 700 in all tasks. Using the same search configuration, ARS does not discover parameters for an LSTM network that supports adaptation to all non-stationary tasks (`LSTM [PO + Act Noise]`).

In summary, preliminary data presented in this section suggests that adding partial-observability and actuator noise to the stationary Cartpole task during search allows ARZ to discover policies that can adapt to *unseen* non-stationary tasks, a methodology that does not work for ARS with LSTM networks. We leave comprehensive analysis of these findings to future work.

S3. Cartpole Algorithm Analysis

Here we analyze the algorithm presented in Figure 11:

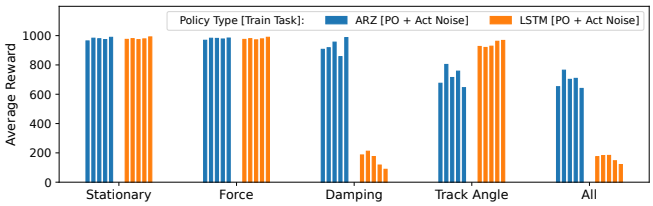


Fig. S7: ARZ can discover policies that adapt to unseen tasks. The plot shows post-evolution test results for *adapting policies* in the Cartpole *continuous change* tasks. The legend indicates policy type and search task. `[All]` indicate policies were exposed to all tasks during evolution. `[PO + Act Noise]` indicates policies were evolved with partial observability and action noise on the stationary task, while the dynamic change tasks were unseen until test. Y-axis is the average reward of 100 episodes in each task. See Section S2-B for discussion.

```
# sX: scalar memory at address X.
# obs: vector [x, theta, x_dot, theta_dot].
# a, b, c: fixed scalar parameters.
# V, W: 4-dimensional vector parameters.
def GetAction(obs, action):
  s0 = a * s2 + action
  s1 = s0 + s1 + b * action + dot(V, obs)
  s2 = s0 + c * s1
  action = s0 + dot(obs, W)
  return action
```

Fig. S8: Sample stateful action function evolved on the Cataclysmic Cartpole task where all parameters are subject to continuous change (ARZ [All] in Fig. 9). Code shown in Python. This figure is a repeat of Figure 11.

Organizing the values of $\mu =$ `s0`, $\nu =$ `s1` and $\xi =$ `s2` at step n into a vector:

$$Z_n = (\mu_n,\ \nu_{n+1},\ \xi_n)^T\ ,$$

and concatenating the observation vector at step $n+1$ and the action at step n into a state vector s_{n+1}:

$$s_{n+1} = (x_{n+1},\ \theta_{n+1},\ \dot{x}_{n+1},\ \dot{\theta}_{n+1},\ \text{act}_n)^T\ ,$$

we can re-write the value of these accumulators at step n in the following way:

$$\begin{aligned} s_{n+1} &= \text{concat}(\text{obs}_{n+1}\ ,\ \text{act}_n) \\ Z_{n+1} &= \widetilde{U}\quad Z_n + \widetilde{P}\quad s_{n+1}\ , \\ \text{act}_{n+1} &= \widetilde{A}^T\quad Z_{n+1} + \widetilde{W}^T\quad s_{n+1}\ . \end{aligned} \tag{2}$$

The particular variables used in this formula map to the parameters of Figure 11 as follows:

$$\widetilde{U} = \begin{pmatrix} 0 & 0 & a \\ 0 & 1 & a \\ 0 & c & a(1+c) \end{pmatrix}\ ,$$

$$\widetilde{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ V_1 & V_2 & V_3 & V_4 & b+1 \\ cV_1 & cV_2 & cV_3 & cV_4 & a+bc+c \end{pmatrix}\ ,$$

$$\widetilde{A} = (1,\ 0,\ 0)^T\ ,$$

$$\widetilde{W} = (W_1,\ W_2,\ W_3,\ W_4,\ 0)^T\ .$$

The numerical values of the parameters of the model found are given by

$$\begin{aligned} a &= \ \ 0.549\ , \quad b = \ \ 0.673\ , \quad c = 0.082\ , \\ V &= (\ \ 1.960,\ \ \ 0.7422,\ 0.7373,\ \ \ 5.284)^T\ , \\ W &= (0.0,\ 0.365,\ 2.878,\ 2.799)^T\ . \end{aligned}$$

Equation (2) can be viewed as a linear recurrent model, where Z_n is the internal state of the model. The action at the n-th step is obtained as a linear function of the internal state, the observation vector and the action value at the previous step. An interesting aspect of the particular model found is that the matrix $\widetilde{U}$ by construction has eigenvalues 0, 1 and $a(1+c)$ 0.594.

Equation (2) being a simple linear model, we may write act_{n+1} explicitly as a sum:

$$\begin{aligned} \text{act}_n =& \widetilde{A}^T\quad \widetilde{U}^n\quad Z_0 + \widetilde{W}^T\quad s_n \\ &+ \widetilde{A}^T\quad \sum_{i=0}^{n} \widetilde{U}^{n-i}\quad \widetilde{P}\quad s_i\ . \end{aligned}$$

When taking the continuous limit of this expression, there is a subtlety in that the s_{n+1} vector is obtained by composing the observation vector at time step $n+1$ and the action value at time step n. We can nevertheless be careful to redefine s to be made up of concurrent components and still arrive at an expression which in the continuous limit, takes the form:

$$\begin{aligned} \text{act}(t) =& c + w^T\quad U^t\quad u + p^T\quad s(t) \\ &+ v^T\quad \int_0^t du\ U^{t-u}\quad P\quad s(u)\ . \end{aligned} \tag{3}$$

We note that when we set $U = \text{Id}$ this expression straightforwardly reduces to a PID controller-like model:

$$\text{act}(t) = (c + w^T\quad u) + p^T\quad s(t) + (v^T\quad P)\quad \int_0^t du s(u)\ .$$

An instructive way of re-writing Equation (3) is to explicitly use the eigenvalues $e^{-\,!\,k}$ of U. The equation can be re-parameterized as

$$\begin{aligned} \text{act}(t) =& c + \sum_{k=1}^{d} c_k e^{-\,!\,k\,t} + p^T\quad s(t) \\ &+ \sum_{k=1}^{d} v_k^T\quad \int_0^t du\ e^{-\,!\,k\,(t-u)} s(u)\ . \end{aligned}$$

Here it is clear that the expression is a straightforward generalization of the PID controller, where only the weight-one cumulant of the history is utilized to compute the action. Now, a multitude of cumulants with distinct decay rates can be utilized.

S4. Complexity Comparison

A. Baselines

As noted in section S1, MLP and LSTM networks have been trained with ARS as baselines for the adaptation tasks in the paper. We can estimate a lower bound for the number parameters and floating point operations required for each model by only counting the matrix variables for the parameter count and matrix multiplications for the floating point operations. This negelects the bias variables and non-matrix multiplication ops such as application of non-linearities or vector component-wise multiplications.

Given the input dimension d_{in}, the output dimension d_{out} and the internal dimension d, we find that the number of parameters and the floating point operations for the MLP and LSTM model per step is given by:

$$\text{FLOPS}_{\text{MLP}}\quad 2\quad \text{Params}_{\text{MLP}} > 2d(d_{\text{in}} + d + d_{\text{out}}) \tag{4}$$

$$\text{FLOPS}_{\text{LSTM}}\quad 2\quad \text{Params}_{\text{LSTM}} > 2d(4d_{\text{in}} + 4d + d_{\text{out}}) \tag{5}$$

B. Quadruped Leg-Breaking

The algorithm presented in Figure 1 contains $16 + 16\quad 37 = 608$ parameters and executes a maximum of $54\quad 37 + 82 = 2080$ floating point ops per step, where we have counted all operations acting on floats or pairs of floats, assuming that all of the "if" statements pass. The input and output dimensions of the tasks are 37 and 12, while the ARS-trained models have internal dimensions $d = 32$. Using the formulae above, we see that the MLP model contains over 2592 parameters and uses more than 5184 FLOPs. Meanwhile the LSTM model uses more than 9216 parameters and 18432 FLOPs.

C. Cataclysmic Cartpole

The algorithm presented in Figure 9 contains 11 parameters and executes 25 floating point ops per step. The input and output dimensions of the tasks are 4 and 1, with internal dimensions $d = 32$ for the neural networks. The MLP model contains over 1184 parameters and uses more than 2368 FLOPs. The LSTM model uses more than 4604 parameters and 9280 FLOPs.

D. Discussion

The efficiency of ARZ policies stems from two characteristics of the system. First, like many genetic programming methods, ARZ builds policies starting from simple algorithms and incrementally adds complexity through interaction with the task environment (e.g., [5], [20]). This implies that the computational cost of action inference is low early in evolution, and only increases as more complex structures provide fitness gains. In other words, the search is bound by *incremental growth*. Second, in ARZ, mutation is twice as likely to remove an instruction than insert an instruction (See Table S1), which has been found to have a regularization effect on the population [3].

S5. Search Space Additional Details

Supplementary Table S2 describes the set of operations in our search space. Note that no matrix operations were used for the quadruped robot domain.

TABLE S2: Ops vocabulary. s, $\vec{v}$ and M denote a scalar, vector, and matrix, resp. Early-alphabet letters (a, b, *etc*.) denote memory addresses. Mid-alphabet letters (*e.g*. i, j, *etc*.) denote vector/matrix indexes ("Index" column). Greek letters denote constants ("Consts." column). $U(\alpha, \beta)$ denotes a sample from a uniform distribution in $[\alpha, \beta]$. $N(\mu, \sigma)$ is analogous for a normal distribution with mean μ and standard deviation σ. $\mathbb{1}_X$ is the indicator function for set X. Example: "$M_a^{(i;j)} = U(\alpha, \beta)$" describes the operation "assign to the i,j-th entry of the matrix at address a a value sampled from a uniform random distribution in $[\alpha, \beta]$".

Op ID	Code Example	Input Args: Addresses / types	Input Args: Consts.	Output Args: Address / type	Output Args: Index	Description (see caption)
OP1	`no_op`	–	–	–	–	–
OP2	`s2=s3+s0`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = s_a + s_b$
OP3	`s4=s0-s1`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = s_a - s_b$
OP4	`s8=s5*s5`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = s_a s_b$
OP5	`s7=s5/s2`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = s_a / s_b$
OP6	`s8=abs(s0)`	*a* / scalar	–	*b* / scalar	–	$s_b = \lvert s_a \rvert$
OP7	`s4=1/s8`	*a* / scalar	–	*b* / scalar	–	$s_b = 1/s_a$
OP8	`s5=sin(s4)`	*a* / scalar	–	*b* / scalar	–	$s_b = \sin(s_a)$
OP9	`s1=cos(s4)`	*a* / scalar	–	*b* / scalar	–	$s_b = \cos(s_a)$
OP10	`s3=tan(s3)`	*a* / scalar	–	*b* / scalar	–	$s_b = \tan(s_a)$
OP11	`s0=arcsin(s4)`	*a* / scalar	–	*b* / scalar	–	$s_b = \arcsin(s_a)$
OP12	`s2=arccos(s0)`	*a* / scalar	–	*b* / scalar	–	$s_b = \arccos(s_a)$
OP13	`s4=arctan(s0)`	*a* / scalar	–	*b* / scalar	–	$s_b = \arctan(s_a)$
OP14	`s1=exp(s2)`	*a* / scalar	–	*b* / scalar	–	$s_b = e^{s_a}$
OP15	`s0=log(s3)`	*a* / scalar	–	*b* / scalar	–	$s_b = \log s_a$
OP16	`s3=heaviside(s0)`	*a* / scalar	–	*b* / scalar	–	$s_b = \mathbb{1}_{\mathbb{R}^+}(s_a)$
OP17	`v2=heaviside(v2)`	*a* / vector	–	*b* / vector	–	$\vec{v}_b^{(i)} = \mathbb{1}_{\mathbb{R}^+}(\vec{v}_a^{(i)})\ \forall i$
OP18	`m7=heaviside(m3)`	*a* / matrix	–	*b* / matrix	–	$M_b^{(i;j)} = \mathbb{1}_{\mathbb{R}^+}(M_a^{(i;j)})\ \forall i;j$
OP19	`v1=s7*v1`	*a*,*b* / sc,vec	–	*c* / vector	–	$\vec{v}_c = s_a \vec{v}_b$
OP20	`v1=bcast(s3)`	*a* / scalar	–	*b* / vector	–	$\vec{v}_b^{(i)} = s_a \quad \forall i$
OP21	`v5=1/v7`	*a* / vector	–	*b* / vector	–	$\vec{v}_b^{(i)} = 1/\vec{v}_a^{(i)} \quad \forall i$
OP22	`s0=norm(v3)`	*a* / scalar	–	*b* / vector	–	$s_b = \lVert \vec{v}_a \rVert$
OP23	`v3=abs(v3)`	*a* / vector	–	*b* / vector	–	$\vec{v}_b^{(i)} = \lvert \vec{v}_a^{(i)} \rvert \quad \forall i$
OP24	`v5=v0+v9`	*a*,*b* / vectors	–	*c* / vector	–	$\vec{v}_c = \vec{v}_a + \vec{v}_b$
OP25	`v1=v0-v9`	*a*,*b* / vectors	–	*c* / vector	–	$\vec{v}_c = \vec{v}_a - \vec{v}_b$
OP26	`v8=v1*v9`	*a*,*b* / vectors	–	*c* / vector	–	$\vec{v}_c^{(i)} = \vec{v}_a^{(i)} \vec{v}_b^{(i)}\ \forall i$
OP27	`v9=v8/v2`	*a*,*b* / vectors	–	*c* / vector	–	$\vec{v}_c^{(i)} = \vec{v}_a^{(i)} / \vec{v}_b^{(i)}\ \forall i$
OP28	`s6=dot(v1,v5)`	*a*,*b* / vectors	–	*c* / scalar	–	$s_c = \vec{v}_a^T \vec{v}_b$
OP29	`m1=outer(v6,v5)`	*a*,*b* / vectors	–	*c* / matrix	–	$M_c = \vec{v}_a \vec{v}_b^T$
OP30	`m1=s4*m2`	*a*,*b* / sc/mat	–	*c* / matrix	–	$M_c = s_a M_b$
OP31	`m3=1/m0`	*a* / matrix	–	*b* / matrix	–	$M_b^{(i;j)} = 1/M_a^{(i;j)}\ \forall i;j$
OP32	`v6=dot(m1,v0)`	*a*,*b* / mat/vec	–	*c* / vector	–	$\vec{v}_c = M_a \vec{v}_b$
OP33	`m2=bcast(v0,axis=0)`	*a* / vector	–	*b* / matrix	–	$M_b^{(i;j)} = \vec{v}_a^{(i)}\ \forall i;j$
OP34	`m2=bcast(v0,axis=1)`	*a* / vector	–	*b* / matrix	–	$M_b^{(j;i)} = \vec{v}_a^{(i)}\ \forall i;j$
OP35	`s2=norm(m1)`	*a* / matrix	–	*b* / scalar	–	$s_b = \lVert M_a \rVert$
OP36	`v4=norm(m7,axis=0)`	*a* / matrix	–	*b* / vector	–	$\vec{v}_b^{(i)} = \lVert M_a^{(i;\,)} \rVert\ \forall i$
OP37	`v4=norm(m7,axis=1)`	*a* / matrix	–	*b* / vector	–	$\vec{v}_b^{(j)} = \lVert M_a^{(\,;j)} \rVert\ \forall j$

[Table continues on the next page.]

TABLE S2: Ops vocabulary (continued)

Op ID	Code Example	Input Args Addresses / types	Consts	Output Args Address / type	Index	Description (see caption)
OP38	`m9=transpose(m3)`	*a* / matrix	–	*b* / matrix	–	$M_b = jM_a^T j$
OP39	`m1=abs(m8)`	*a* / matrix	–	*b* / matrix	–	$M_b^{(i;j)} = jM_a^{(i;j)}j\ 8i;j$
OP40	`m2=m2+m0`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c = M_a + M_b$
OP41	`m2=m3-m1`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c = M_a \quad M_b$
OP42	`m3=m2*m3`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c^{(i;j)} = M_a^{(i;j)}\ M_b^{(i;j)}\ 8i;j$
OP43	`m4=m2/m4`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c^{(i;j)} = M_a^{(i;j)} = M_b^{(i;j)}\ 8i;j$
OP44	`m5=matmul(m5,m7)`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c = M_a\ M_b$
OP45	`s1=minimum(s2,s3)`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = \min(s_a; s_b)$
OP46	`v4=minimum(v3,v9)`	*a*,*b* / vectors	–	*c* / vector	–	$\boldsymbol{v}_c^{(i)} = \min(\boldsymbol{v}_a^{(i)}; \boldsymbol{v}_b^{(i)})\ 8i$
OP47	`m2=minimum(m2,m1)`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c^{(i;j)} = \min(M_a^{(i;j)}; M_b^{(i;j)})\ 8i;j$
OP48	`s8=maximum(s3,s0)`	*a*,*b* / scalars	–	*c* / scalar	–	$s_c = \max(s_a; s_b)$
OP49	`v7=maximum(v3,v6)`	*a*,*b* / vectors	–	*c* / vector	–	$\boldsymbol{v}_c^{(i)} = \max(\boldsymbol{v}_a^{(i)}; \boldsymbol{v}_b^{(i)})\ 8i$
OP50	`m7=maximum(m1,m0)`	*a*,*b* / matrixes	–	*c* / matrix	–	$M_c^{(i;j)} = \max(M_a^{(i;j)}; M_b^{(i;j)})\ 8i;j$
OP51	`s2=mean(v2)`	*a* / vector	–	*b* / scalar	–	$s_b = \text{mean}(\boldsymbol{v}_a)$
OP52	`s2=mean(m8)`	*a* / matrix	–	*b* / scalar	–	$s_b = \text{mean}(M_a)$
OP53	`v1=mean(m2,axis=0)`	*a* / matrix	–	*b* / vector	–	$\boldsymbol{v}_b^{(i)} = \text{mean}(M_a^{(i;\ )})\ 8i$
OP54	`v3=std(m2,axis=0)`	*a* / matrix	–	*b* / vector	–	$\boldsymbol{v}_b^{(i)} = \text{stdev}(M_a^{(i;\ )})\ 8i$
OP55	`s3=std(v3)`	*a* / vector	–	*b* / scalar	–	$s_b = \text{stdev}(\boldsymbol{v}_a)$
OP56	`s4=std(m0)`	*a* / matrix	–	*b* / scalar	–	$s_b = \text{stdev}(M_a)$
OP57	`s2=C1`	–		*a* / scalar	–	$s_a =$
OP58	`v3[5]=C2`	–		*a* / vector	*i*	$\boldsymbol{v}_a^{(i)} =$
OP59	`m2[5,1]=C1`	–		*a* / matrix	*i*, *j*	$M_a^{(i;j)} =$
OP60	`s4=uniform(C2,C3)`	–	,	*a* / scalar	–	$s_a = U(\ ;\ )$
OP61	`m2=m4`	*a* / matrix	–	*b* / matrix	–	$M_b = M_a$
OP62	`v2=v4`	*a* / vector	–	*b* / vector	–	$\boldsymbol{v}_b = \boldsymbol{v}_a$
OP63	`i2=i4`	*a* / index	–	*b* / index	–	$i_b = i_a$
OP64	`v2=power(v5,v3)`	*a*,*b* / vectors	–	*c* / vector	–	$\boldsymbol{v}_c^{(i)} = \text{power}(\boldsymbol{v}_a^{(i)}; \boldsymbol{v}_b^{(i)})\ 8i$
OP65	`v3=m2[:,1]`	*a*,*b* / matrix,index	–	*c* / vector	–	$\boldsymbol{v}_c = M_a^{(\ ;j_b)}$
OP66	`v3=m2[1,:]`	*a*,*b* / matrix,index	–	*c* / vector	–	$\boldsymbol{v}_c = M_a^{(i_b;\ )}$
OP67	`s3=m2[1,5]`	*a*,*b*,*c* / m,i,i	–	*d* / scalar	–	$s_d = M_a^{(i_b;j_c)}$
OP68	`s3=v2[5]`	*a*,*b* / vector,index	–	*c* / scalar	–	$s_c = \boldsymbol{v}_a^{(i_b)}$
OP69	`v3=0`	–	–	*a* / vector	–	$\boldsymbol{v}_a = 0$
OP70	`s5=0`	–	–	*a* / scalar	–	$s_a = 0$
OP71	`i2=0`	–	–	*a* / index	–	$i_a = 0$
OP72	`v2=sqrt(v5)`	*a* / vector	–	*b* / vector	–	$\boldsymbol{v}_b^{(i)} = \text{sqrt}(\boldsymbol{v}_a^{(i)})\ 8i$
OP73	`v2=power(v5,2)`	*a* / vector	–	*b* / vector	–	$\boldsymbol{v}_b^{(i)} = \text{power}(\boldsymbol{v}_a^{(i)}; 2)\ 8i$
OP74	`s1=sum(v5)`	*a* / vector	–	*b* / scalar	–	$s_b = \text{sum}(\boldsymbol{v}_a^{(i)})\ 8i$
OP75	`s5=sqrt(s1)`	*a* / scalar	–	*b* / scalar	–	$s_b = \sqrt{s_a}$
OP76	`s3=s0*s2+s5`	*a*,*b*,*c* / scalars	–	*d* / scalar	–	$s_d = s_a \quad s_b + sc$
OP77	`s2=s4*C1`	*a* / scalar		*b* / scalar	–	$s_b = s_b$
OP78	`m2[1,:]=v3`	*a* / vector	–	*b* / matrix	*i*	$M_b^{(i;\ )} = \boldsymbol{v}_a$
OP79	`m2[:,1]=v3`	*a* / vector	–	*b* / matrix	*i*	$M_b^{(\ ;j)} = \boldsymbol{v}_a$
OP80	`i3 = size(m1, axis=0) - 1`	*a* / matrix	–	*b* / index	–	$i_b = \text{size}(M_a^{(i;\ )}) \quad 1$
OP81	`i3 = size(m1, axis=1) - 1`	*a* / matrix	–	*b* / index	–	$i_b = \text{size}(M_a^{(\ ;j)}) \quad 1$
OP82	`i3 = len(v1) - 1`	*a* / vector	–	*b* / index	–	$i_b = \text{len}(\boldsymbol{v}_a) \quad 1$
OP83	`s1 = v0[3] * v1[3] + s0`	*a*,*b*,*c*,*d* / v,v,s,i	–	*e* / scalar	–	$s_e = \boldsymbol{v}_a^{(i_d)} \quad \boldsymbol{v}_b^{(i_d)} + s_c$
OP84	`s3=dot(v0[:5],v1[:5])`	*a*,*b*,*c* / v,s,i	–	*d* / scalar	–	$s_d = \boldsymbol{v}_a^{T(:i_c+1)} \boldsymbol{v}_b^{(:i_c+1)}$