University of Cape Town

A dissertation submitted in satisfaction of the requirements for the degree Master of Science in Computer Science

Scalable Hierarchical Evolution Strategies

Author:
Sasha Abramowitz

Supervisor:
Geoff Nitschke

February 2022

I know the meaning of plagiarism and declare that all of the work in this document, save for that which is properly acknowledged, is my own.

Acknowledgements

I would like to thank my supervisor Geoff Nitschke for his guidance, feedback and general 24/7 availability throughout the past two years.

I would also like to thank fellow masters student Shane Acton for our discussions about our work and AI in general, which kept me excited about the field throughout my masters.

I also want to thank my parents for their support, both financially and emotionally throughout my entire time at university.

Finally, I would like to thank the National Research Foundation for funding this thesis through the Human and Social Dynamics in Development (Grant Number: 118557).

Abstract

Hierarchical reinforcement learning (HRL) has been steadily growing in popularity for solving the hardest reinforcement learning problems. However, current HRL algorithms are relatively slow and brittle to hyperparameter changes. This paper offers a solution to these slow and brittle HRL algorithms, by investigating a novel method combining *Scalable Evolution Strategies* (S-ES) and HRL. S-ES, named for its excellent scalability, was popularised by OpenAI when they showed its performance to be comparable to state-of-the-art policy gradient methods. However, S-ES has not been tested in conjunction with HRL methods, which empower temporal abstraction thus allowing agents to tackle more challenging problems. We introduce a novel method merging S-ES and HRL, which creates a highly scalable and fast (wall-clock time) algorithm. We demonstrate that S-ES needs no hyper-parameter tuning for the HRL tasks tested and is indifferent to delayed rewards. This results in a method that is significantly faster than gradient-based HRL methods while having competitive task performance. We extend this method using transfer learning with the aim of increasing task performance and novelty search with the goal of improving its exploration characteristics. The paper's main contribution is thus a novel evolutionary HRL method, namely Scalable Hierarchical Evolution Strategies, which yields greater learning speed and competitive task-performance compared to state-of-the-art gradient-based methods, across a range of tasks.

Contents

1 Introduction

Reinforcement learning (RL) [Sutton and Barto, 2018] is one of the three main sub-fields of Artificial intelligence (AI) and is used for sequential decision making tasks. In RL one trains a policy such that an agent can take intelligent actions in an environment in order to achieve some goal. Critically and unlike supervised learning methods, RL does not require any data and learns through repeated interactions with its environment. The fundamental ideas behind RL mirror those of Pavlovian or classical conditioning in psychology [Sutton and Barto, 2018].

RL has been used to create artificially intelligent agents for tasks ranging from robot locomotion [Haarnoja et al., 2018] to video games such as StarCraft [Vinyals et al., 2019] and board games such as chess and Go [Silver et al., 2018]. Many such agents use *Markov Decision Process* (MDP) or *gradient*-based learning methods, such as Deep Q-Networks (DQNs) [Mnih et al., 2015] and the policy gradient family of methods [Sutton et al., 1999a, Sutton and Barto, 2018]. Single policy (flat) RL is generally used for relatively simple problems, however, problems can quickly become complex by requiring multiple unrelated skills in order to be solved or having a sparse reward signal. To solve the type of problem that current flat RL cannot solve (which in this paper we refer to as hard RL problems) one generally uses hierarchical reinforcement learning (HRL). HRL algorithms are a class of RL algorithms that excel at complex RL problems by decomposing them into sub-tasks, which mimics the way we as humans build new skills on top of existing simpler skills. Gradient-based RL methods are also used by HRL algorithms and have seen success in solving some of the hardest RL environments such as Montezumas revenge [Vezhnevets et al., 2017, Badia et al., 2020] and generating complex robot behaviours [Nachum et al., 2018, Vezhnevets et al., 2017]. Another area of reinforcement learning that has enjoyed recent success is evolution strategies (ES). These are a family of black-box evolutionary optimization techniques, which Salimans et al. showed are competitive with flat gradient-based RL methods in the robot locomotion and Atari domain [Salimans et al., 2017]. The success of such approaches has led to wider use of ES for tackling RL problems, but it has yet to be used to solve hard RL problems.

ES has been used as a black-box optimizer for a multitude of problems such as minimizing the drag of 3D bodies [Beyer and Schwefel, 2002], optimizing designs in structural and mechanical engineering problems [Datoussaïd et al., 2006], robot locomotion [Salimans et al., 2017, Conti et al., 2017, Katona et al., 2021] and loss function optimization [Gonzalez and Miikkulainen, 2020]. There are many different flavours of ES [Beyer and Schwefel, 2002] each with a different selection, mutation and self-adaption properties, for example, CMA-ES [Hansen and Ostermeier, 2001] and $(1+\gamma)$-ES [Beyer and Schwefel, 2002]. However, in this work, we are concerned with the version proposed by Salimans et al., namely *Scalable Evolution Strategies*

(S-ES) because of its proven performance in the domain of robot locomotion and Atari game playing [Salimans et al., 2017]. All flavours of ES follow the scheme of sample-and-evaluate, where it samples a *cloud* of policy variants around its current policy's parameters, evaluates these sampled policies to obtain many fitness values and uses this local knowledge of the fitness landscape to inform an update to the current policy. S-ES specifically uses fitness to approximate the gradient and moves the current policy parameters in the direction that maximizes the average reward. Given that ES is both a black-box process and is a gradient-free method, it suffers from suboptimal sample efficiency, however Liu et al. showed promising results addressing this inefficiency using trust regions which allow for more of a monotonic improvement [Liu et al., 2019].

S-ES obtained results comparable to gradient methods on a set of standard MuJoCo [Todorov et al., 2012] and Atari [Mnih et al., 2015] benchmarks [Salimans et al., 2017]. However, there are many RL problems harder than these standard benchmarks, some of which are near impossible to solve using flat RL and others that are unsolved using flat RL. These environments can range from games such as Montezuma's revenge [Mnih et al., 2015] which is challenging because it requires long-term credit assignment, to robot locomotion, navigation and interaction [Nachum et al., 2018, Florensa et al., 2017] which requires complex multi-level reasoning.

One facet of gradient-based RL that *vanilla* evolutionary algorithms do not require is an explicit method to cope with the exploration vs exploitation trade-off. Evolutionary algorithms usually get around this issue because of the sheer size of the population which is able to perform the exploration without being explicitly made to explore. However, this does not always work for hard exploration problems [Lehman and Stanley, 2011a] and one needs to include specific exploration mechanisms in order to achieve good performance in these types of problems. One such mechanism is called novelty search (NS) [Lehman and Stanley, 2011a, 2008], this was proposed by Lehman and Stanley and is the process of searching explicitly for novel behaviours instead of following the objective function. There are two reasons this approach is effective: first, objective functions can be highly deceiving having many local minima that can be challenging to escape. Second, objective functions provide no reward for interim steps taken towards the objective. NS is able to avoid deceptive local minima, by simply rewarding *unseen* behaviours and it is able to reward not only the outcome of the behaviour but the intricacies of the behaviour itself, which allows it to reward important steps taken towards the end goal. Conti et al. have already shown that NS works well with S-ES and that it is able to be used in conjunction with objective functions to create a classic RL style explore/exploit trade-off parameter [Conti et al., 2017].

HRL has long held the promise of solving much more complex tasks than flat RL methods. It allows policies to abstract away large amounts of complexity and focus on solving simpler sub-goals. One of the earliest and most popular HRL methods is known as the options framework [Sutton et al., 1999b] which allows the controller policy to select the most appropriate primitive policy from a pool of primitive policies, this primitive passes control back to the controller once its actions are completed and the process repeats. Another competing HRL framework is feudal-RL [Dayan and Hinton, 1993], this framework allows for communication between the controller and primitives by having the controller set goals for the primitive to complete. Recent feudal-RL methods such as FeUdal Networks for HRL (FuN) [Vezhnevets et al., 2017] and HRL with Off-Policy Correction (HIRO) [Nachum et al., 2018] have shown a lot of promise for learning sparse reward problems and hierarchies requiring complex primitives, especially in the robot locomotion domain. HIRO in particular takes the approach of using a two-level hierarchy (one controller and one primitive) where the controller sets the goal and reward for the primitive. For example, the goal can take the form of a position an agent must reach and the reward is based on the agent's distance to the goal position. HIRO, FuN and most modern HRL algorithms use gradient-based RL methods to optimize their hierarchy of policies [Vezhnevets et al., 2017, Nachum et al., 2018, Sutton et al., 1999b, Badia et al., 2020] and to the best of the author's knowledge, non-gradient based RL solvers, such as ES, have not been extensively tested on hard RL problems that are typically reserved for gradient-based HRL solvers.

HRL is not the only way to train RL agents on difficult tasks, another option is transfer learning. This is the process of initially training an agent to perform some adjacent task and then transferring that knowledge to the difficult task. This allows the agent to get a *warm start* in the hard task by using the knowledge it gained from the adjacent, but easier task. There are many ways one can transfer knowledge to an RL agent, the simplest is to use the pretrained policy's parameters as a *warm start*, however, this is only possible if the observation and action spaces of the two tasks match. Another popular approach is to use the trained expert as a *teacher* by providing the probability it would take certain actions in certain states. Finally one could use the policy trained on the adjacent task to directly provide demonstrations on the new tasks that can easily be used to train off-policy RL algorithms. All these approaches are used to boost an RL agents performance on hard tasks and can be combined with the policies in an HRL system to improve the performance or sample efficiency of the learning process [Nachum et al., 2018, Florensa et al., 2017].

ES has multiple advantages over gradient-based RL methods, but two of these advantages make ES especially suited for hard RL problems. First, it is invariant to delayed rewards and second, it has a more structured exploration mechanism [Sal-

imans et al., 2017, Conti et al., 2017] relative to gradient-based RL methods. Its robustness to delayed rewards is especially useful for hard RL problems as much of the difficulty of these environments can come from the long term credit assignment problem. Similarly, hard RL problems often have many large local minima, requiring intelligent exploration methods in order to be solved. These advantages suggest that ES and specifically S-ES should perform well on challenging RL problems, however to the best of the author's knowledge S-ES has not yet been applied to hard RL problems.

Furthermore, contrary to current state-of-the-art RL and HRL frameworks S-ES is highly robust to hyper-parameter changes [Salimans et al., 2017], given that HRL methods only introduce more hyperparameters, the brittleness of current RL methods [Haarnoja et al., 2018, Paine et al., 2020] greatly increases the amount of time needed to tune the HRL methods that use them. The proposed framework aims to address this by leveraging the robustness of S-ES to create a HRL method that needs little to no hyperparameter tuning. Thus, we introduce a new method[1] for training two-level policy hierarchies, optimized using the S-ES method: *Scalable Hierarchical Evolution Strategies* (SHES).

We compare SHES task-performance to other gradient-based HRL methods, also evaluated on the same tasks [Nachum et al., 2018, Vezhnevets et al., 2017, Houthooft et al., 2016, Florensa et al., 2017]. Additionally, we extend SHES with novelty search and transfer learning in an attempt to improve task performance and learning speed. The main objective is to demonstrate that SHES performs well on tasks that are challenging for gradient-based HRL methods and hence that S-ES is suitable for training hierarchies of policies. Our SHES method addresses various RL and HRL deficiencies by leveraging the benefits of S-ES to create an HRL method requiring minimal hyper-parameter tuning, that is able to learn faster than current gradient-based HRL methods and is competitive with state-of-the-art HRL methods (in terms of task performance) across four hard RL task environments.

1.1 Research goals

This work aims to address the lack of fast, scalable and performant HRL algorithms through the introduction of a new HRL method: Scalable Hierarchical Evolution Strategies. This method also address the brittleness of current HRL methods to hyperparameter changes [Nachum et al., 2018, Vezhnevets et al., 2017, Paine et al., 2020], given S-ES' high robustness to parameters. This thesis will go on to test two extensions to the proposed framework with the goal of improving the overall performance and exploration characteristics of the framework.

[1] `https://github.com/sash-a/ScalableHrlEs.jl`

1. Create an evolutionary HRL framework that can scalable to numerous CPUs, such that when given enough computing power it is faster (in terms of wall clock time) than gradient based methods
2. Create an evolutionary HRL framework which is competitive with state-of-the-art gradient based methods on relevant benchmarks (see chapter 4) and can generalise across multiple tasks (outlined in chapter 4)
3. Investigate the impact of novelty search and transfer learning on the frameworks exploration, task performance and sample efficiency, with respect to the benchmark tasks described in chapter 4

1.2 Contributions

1. A novel evolutionary HRL framework, which excels at locomotion and navigation style problems
2. One of the few HRL frameworks that is robust to hyperparameters and requires little to no tuning
3. A fast HRL algorithm that can be used when when wall clock time is critical
4. An open source, fast and efficient implementation of this framework, S-ES and all environments used in the Julia language

1.3 Scope

This thesis is primarily focused on providing the HRL space with a high task performance evolutionary method for complex locomotion and navigation problems, which is able to address some of the shortcomings of current gradient based HRL methods. Because of the number of hierarchical environments, evolutionary methods and HRL paradigms that exist we limit the scope of this thesis specifically to:

- Locomotion based, continuous control environments
- Evolution strategies as the evolutionary method
- Feudal-RL as the HRL paradigm

Due to this limiting of scope there is much room to extend this work and explore this research question without some or all of these limitations.

1.4 Structure

This thesis is structured as follows. Chapter 2 provides related work and fundamentals required for the understanding of this research. It explores previous work in the fields of hierarchical reinforcement learning, transfer learning for reinforcement learning, evolution strategies and novelty search. Chapter 3 provides an overview on the design and implementation of the framework. Chapter 4 details the design of the environments used to evaluate the framework and their significance to the wider RL community as well as comparing them to similar environments. It also provides a visualization of the results of our experiments. Chapter 5 presents and discusses the results of the experiments conducted, which compare the framework and some extensions to existing methods and the results of tuning the most sensitive and important hyperparameters. Chapter 6 concludes and gives future directions for this work.

2 Background

This section introduces artificial neural networks (ANNs), reinforcement learning (RL) with a specific focus on hierarchical reinforcement learning (HRL) and evolutionary computation, specifically focusing on evolution strategies (ES).

2.1 Neural networks

Arguably one of the most impactful catalysts for AI research was the 2012 breakthrough usage of artificial neural networks (ANNs) for image recognition in the image net competition [Krizhevsky et al., 2012]. The ANN approach well outperformed existing approaches and moved the field of AI research out of an *AI winter*. ANNs are a very rough approximation of how the neurons in our brains connect and interact, however they are mainly very good function approximators and with enough hidden layers can theoretically approximate any function [Hornik et al., 1989]. ANNs with many large layers are called deep neural networks (DNNs) and are widely used in state-of-the-art image recognition and natural language processing methods [Vaswani et al., 2017, Zhai et al., 2021, Dosovitskiy et al., 2021, Tan and Le, 2021, Kolesnikov et al., 2020, Cui et al., 2020].

At a basic level, they consist of connections (weights), nodes and activation functions, these can be seen in figure 1. Where the input layer (green) takes in two values, these values are passed through the connections to the hidden layer (blue) while being multiplied by the weights associated with the connections. All values passed to the same node are summed and then passed through a non-linear *activation function*, commonly this is a sigmoid or some kind of rectified linear unit [Nair and Hinton, 2010]. After activation, the hidden values are passed to the output node (yellow), through their connections and multiplied by the corresponding weights. Similar to the previous layer, all values passed to the same node are summed, however, depending on the task the output node may not use an activation function. This small example shows how ANNs perform a forward pass (inference).

There are many different types of neural networks, the one seen in figure 1 is a feed-forward, fully connected neural network. Meaning that connections form an acyclic graph by only connecting nodes to other nodes closer to the output, more formally: every node in the layer $n-1$ is connected to every node in layer n. Convolutional neural networks have seen success in image recognition tasks [Tan and Le, 2021, Kolesnikov et al., 2020, Cui et al., 2020] and is what Krizhevsky et al. used in the 2012 breakthrough. However recently a new architecture called the transformer [Vaswani et al., 2017] has also seen a lot of success in this area [Zhai et al., 2021, Dosovitskiy et al., 2021], this architecture was originally used for natural language processing [Vaswani et al., 2017] and followed the success of recurrent architectures

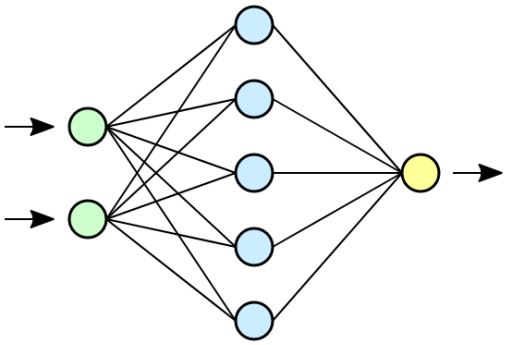


Figure 1: A simple example of a neural network with two input nodes (green), five hidden nodes (blue) and one output node (yellow)

in this domain [Dhingra et al., 2018, Graves et al., 2005]. This shows how many different types of architectures exist for ANNs, in this work we are simply concerned with fully connected feed-forward models as seen in figure 1, however since all other architectures are simply defined by a weight matrix there is no reason the proposed method could not also use these architectures, although larger models may require more samples to be taken and thus take more time to train.

The most common way for an ANN to learn is through a process called backpropagation. More broadly this is the process of passing the error signal (difference in the output and expected output) backwards from the output to the input layer and is done using partial derivatives to guide the weight update in the direction of lowest error.

However, this is not the only way in which ANNs can learn, they can also use evolutionary methods (discussed in more detail in section 2.3). There are many different types of evolutionary methods, but generally, they use a population of ANNs and move the weights directly to or in the direction of the highest performing ANN or ANNs. This can be quite beneficial in terms of hardware constraints since large scale backpropagation based learning is highly dependent on graphics processing units (GPUs) which the largest models need hundreds of and require large amounts of memory in order to track the gradients, while evolutionary methods usually run on the central processing unit (CPU) which is comparably cheaper and require no gradient tracking.

2.2 Reinforcement Learning

Reinforcement learning (RL) is one of the three main sub-fields of artificial intelligence, the other two being supervised and unsupervised learning. RL differs from

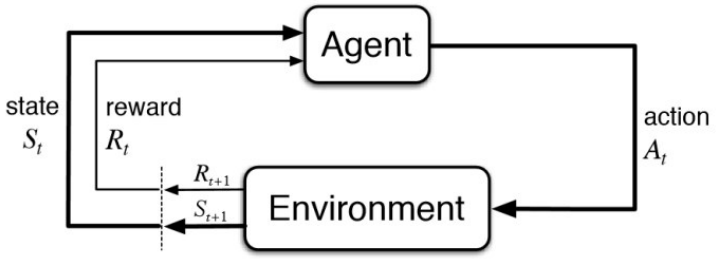


Figure 2: Simple example of the flow of an RL algorithm or an MDP. The agent observes it's environment state S_t and interacts with its environment through action A_t, the environment provides the agent with a reward R_{t+1} and the agent observes the updated environment S_{t+1}. (Image from Bhatt [2018])

these fields by requiring no data since learning occurs through interactions with its environment. Following the structure seen in figure 2 an agent interacts with its environment through actions A_t and receives rewards R_t for those actions while observing the effects that those actions had on the state S_t of the environment. It is important to note that the current state S_t is independent of past states, in other words, it captures all relevant information about the world. The goal of the agent is to maximize the reward R_t provided by the environment through the actions A_t it takes. This process is known as a Markov Decision Process (MDP) and underpins most RL algorithms.

A problem unique to RL is the *exploration vs exploitation* trade-off. For an RL agent to perform well, it should take actions it knows have yielded high rewards in the past, however to discover such actions it needs to try new actions which may yield this high reward. Thus an agent needs to *exploit* its current knowledge, but also *explore* the space of possible actions. The problem that arises is that the agent cannot focus solely on exploration or exploitation, it must find a balance between both to maximize its reward. A common strategy is to start by favouring exploration and move towards favouring exploitation as the agent learns more about its environment.

The final basic RL concept that will be explained is on-policy and off-policy learning. On-policy learning attempts to improve the policy that is interacting with the environment, whereas off-policy learning attempts to improve a different policy from the one used to interact with the environment such that one policy learns and another policy generates experience. Off-policy learning generally requires extra complexities such as *off-policy correction*, however, it is usually much more *sample efficient* meaning that while learning it requires fewer interactions with the environment as it can re-use the generated experience, which is stored in a *replay buffer*.

Modern RL generally uses ANNs and DNNs as opposed to a tabular approach and is usually referred to as deep RL. It is heavily based on Markov Decision processes (MDPs) and has shown to be useful in solving challenging problems from achieving expert performance in the DOTA video game [Berner et al., 2019] to learning how to walk [Fujimoto et al., 2018, Haarnoja et al., 2018, Yu et al., 2018]. Modern RL also commonly uses temporal difference learning, which allows the agent to learn by bootstrapping the current estimate of the value of a state. In this work, we are most concerned with the twin delay deep deterministic policy gradients (TD3) method [Fujimoto et al., 2018], which is an off-policy, actor-critic style method and when published achieved state-of-the-art results in the standard Atari and MuJoCo domain [Fujimoto et al., 2018].

There are many more aspects to RL which are out of the scope of this section, the goal of this section is to provide information on the parts of RL most relevant to this work. In the following sections, we discuss hierarchical reinforcement learning and the many forms it takes and transfer learning in RL.

2.2.1 Transfer learning for Reinforcement Learning

Transfer learning is a promising way to accelerate the learning process of a reinforcement learning agent. Provided that one of the research goals of this work is to create a run-time efficient HRL algorithm, transfer learning is a logical extension to speed up learning. There are multiple different transfer learning methods [Zhu et al., 2020, Hawasly and Ramamoorthy, 2013] for use in RL, but the key idea behind all is that transfer learning attempts to take knowledge from a set of source domains and transfer it to a target domain.

One popular RL method is called imitation learning, this is where an agent learns from demonstrations and attempts to mimic its teacher. This is likely one of the most popular forms of transfer learning for RL, however, a policy is trained in a supervised learning manner, instead of through gathered experience. Another option for transfer learning in reinforcement learning is to learn the world dynamics in a source domain that are still applicable in the target domain. This allows the agent to use the learned dynamics to speed up training or as a source of extra information about the world. Two other transfer learning approaches are teacher policies and teacher value functions. These allows a policy to *consult* a trained policy or value function, which can predict the outcome of its actions thus teaching the policy which actions are best to take. Transfer learning can also be used in the HRL domain and have shown success when it is used to pretrain lower level policies [Florensa et al., 2017]. This will be discussed more in section 2.2.2.4 and is especially relevant to this work as it can be combined with the proposed method for potentially improved learning speed and higher task performance.

2.2.2 Hierarchical Reinforcement Learning

The concepts behind HRL are based on the ways we interact with our environment. When picking up a ball one does not think about which muscles to move to grasp that ball, at least at a high level one is only thinking *I must pick up this ball*. HRL tries to mimic this behaviour by breaking down a given task into logical sub-tasks (primitives). Each *primitive* has its own policy and communicates with a higher-level *controller* policy. The hierarchy of controllers and primitives can have multiple levels with multiple controllers/primitives in each level or only a single level with a single primitive and controller. The former has the benefit of allowing more fine-grained control by breaking up the task as much as possible, but for many tasks, it can be difficult to define numerous primitives and controllers. Taking the approach with a single controller and primitive is a more common and modern approach and is what this work will focus on, it allows the task to be broken up into two simpler subtasks, which do not need as much engineering time when compared to multilevel hierarchies, but still provides performant hierarchies [Nachum et al., 2018, Vezhnevets et al., 2017, Florensa et al., 2017]. More importantly than how many controllers/primitives a hierarchy will have is how the controllers will control and communicate with the primitives. There are two prevailing, yet contrasting ideas of how to handle the controller primitive interaction, namely the Options framework [Sutton et al., 1999b] and Feudal-RL [Dayan and Hinton, 1993].

2.2.2.1 Options framework Likely the most common way of handling the controller-primitive interactions in a HRL method is the Options framework [Sutton et al., 1999b], introduced by Sutton et al. in 1999b. Unlike many RL methods the Options framework is not based around an MDP [Bellman, 1957], but rather a semi-Markov decision process (SMDP) [Sutton et al., 1999b], this allows for actions to take varying lengths of time as seen in figure 3, a feature which is critical to this framework. The key conception behind the Options framework is an option, this encapsulates the idea that certain actions are composed of sub-actions. An option could be something as simple as twitching a muscle or something as complicated as scoring a goal, this allows the output of all policies in the Options framework to be considered options thus allowing primitives and controllers to behave in the same manner.

Initially, control of the agent is given to the root policy in the hierarchy, given the current state, this policy will then decide to pass control to one of its child policies. If the new policy is a primitive it directly takes actions in the world, otherwise, this process is repeated until a primitive policy is given control. Once a policy is active it remains active until a termination condition is met, at which point it passes control back to its parent policy, this is why SMDPs are crucial to the Options framework as

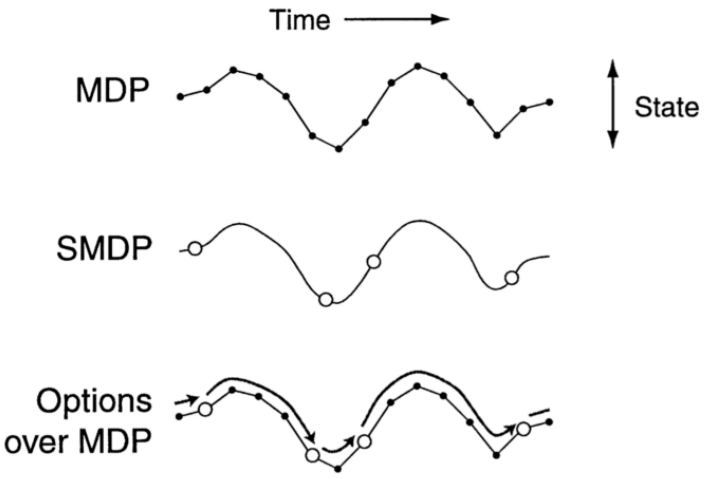


Figure 3: The relationship between an MDP, SMDP and Options over an MDP, showing how the options framework allows policies to smoothly interact over varying lengths of time. Each dot is an agent taking an action in the environment. In the bottom image the large white circles would be a controller policy and the black dots would be a primitive policy that interacts with the environment at regular time intervals. (Image from: Sutton et al. [1999b])

the parent policy has performed a single action during the multiple timesteps that the child policy is active.

The Options framework allows for parent policies to abstract away the temporal details of their children as they are merely concerned with the outcome of activating the child policy. It does not require any communication between child and parent policies as parent policies simply decide which is the most appropriate child policy to activate, but do not pass any extra information to that child policy other than the current state. The Options framework has shown excellent results [Bai et al., 2015] in classic HRL domains such as RoboCup [Kitano et al., 1997]

2.2.2.2 Feudal RL A competing HRL framework is Feudal-RL [Dayan and Hinton, 1993], this was introduced in 1993 by Dayan and Hinton to specifically tackle navigation style problems and has seen a rise in popularity with the advent of deep-HRL methods. The feudal-RL literature uses slightly different terminology to what is used in this work, instead of controllers and primitives, feudal-RL has managers and sub-managers or workers. This is an apt description because relative to the Options framework, there is much more inter-policy communication similar to how managers communicate desired outcomes with their employees. This is the key

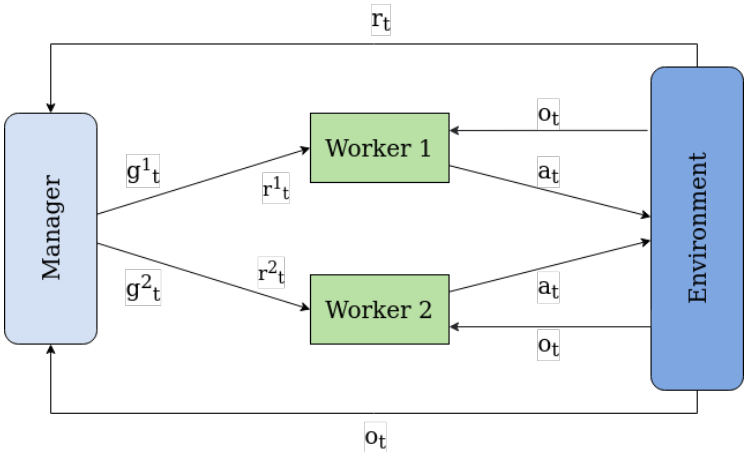


Figure 4: An example of a single level feudal-RL hierarchy. The manager (controller) and its workers (primitives) are passed an observation o_t from the environment, using o_t the manager passes a goal g_t to its workers who produce actions a_t which affect the environment. The environment passes a reward r_t to the manager who then rewards its works with rewards r_t^1 and r_t^2 given the updated observation it receives from the environment. This should serve to highlight the fact that it is the manager's job, not the environment's, to reward the workers.

idea behind feudal-RL: managers not only select the sub-manager as in the Options framework but also assign them a goal and reward.

Initial control is given to the root policy or the highest manager in the hierarchy (in figure 4 this is the policy labelled manager), this manager then sets goals for its sub-manager (the policies labelled workers in figure 4). This process repeats until a leaf policy in the hierarchy is activated, which directly interacts with the environment. Each policy is usually activated for a constant number of steps until it passes control back to its super-manager. Each step a policy is rewarded by its super-manager, this allows for one of the key ideas behind feudal-RL: *reward hiding*. Reward hiding is simply the fact that a manager must reward its active sub-manager even if its performance does not satisfy the goals set by its super-manager. This creates a separation of concerns and allows lower-level managers to master a skill before high-level goals are achieved. The second principle behind feudal-RL is *information hiding*, this is the fact that certain managers can observe more or less of the state depending on their needs. Higher-level managers will usually observe a coarse-grained view of the state, while lower-level managers will usually have a finer-grained view of the state. Information hiding is not as prevalent in modern iterations of feudal-RL, but reward hiding is a key ingredient that is still very important in these methods [Nachum et al., 2018, Vezhnevets et al., 2017].

One of the biggest drawbacks of the feudal-RL approach is that it creates a non-stationary problem for the manager to solve. In RL a non-stationary problem is one where the underlying values are constantly changing and thus the best action to take also constantly changes. This is the case for managers in a feudal-RL system since their sub-managers are learning at the same time as they are, thus constantly changing their behaviour. This is of course a drawback because it makes the manager policies harder to learn, however it is also an issue for off-policy algorithms which will be discussed in the following section. This drawback stems from *reward hiding* since each primitive is given the freedom to only optimize for its goal and not its manager's goal. Therefore one of feudal-RLs strengths also contributes to one of its weaknesses.

2.2.2.3 Data efficient hierarchical reinforcement learning HIRO is currently the state-of-the-art feudal RL method [Nachum et al., 2018]. It was tested on the same complex locomotion environments used in this paper (chapter 4) which require the agent to learn both how to control the body of the robot and how to navigate the robot around obstacles. It uses a simple two-level hierarchy of one controller (μ_c) and one primitive (μ_p), where the policies are represented by ANNs with parameters θ_c and θ_p respectively. As in feudal RL, the controller communicates the goal to the primitive, however, HIRO takes this a step further by making the primitive goal (g_t) the entire state space (s_t) of the agent, thus the goal of the primitive is to exactly match all the environment's state (s) to the goal passed by the controller. This makes HIRO exceedingly general as it requires minimal human effort to devise a new goal for each new environment since all goals are the same since the goal is simply the euclidean distance between the current state s_t and the goal g_t. However, this does come at the cost of being harder to learn [Nachum et al., 2019] and limits the types of ways the primitive can be rewarded to only the distance between the goal g_t and the environment's current state s_t. As general as HIRO's goal encoding is, it does still require human engineering since ranges must be specified for each value in the state. In a future work Nachum et al. improve on their approach to HIRO by learning an optimal representation for the goal g_t at the same time as learning to solve the problem [Nachum et al., 2019]. Nachum et al. showed that this approach does improve task performance when compared to passing the entire state as in HIRO and gives similar performance to an optimal goal encoding.

A single episode of HIRO runs similarly to the process described in the feudal RL section (2.2.2.2), the main difference being HIROs static goal transition function $h(s_t, g_t, s_{t+1})$. This is necessary because HIRO was tested on navigation style environments and the goal g_t produced by the controller (every 10 steps) is relative to the agent's current position, thus the static goal transition function simply transforms this to be relative to the agent's new position each step that the controller is

not active.

$$h(s_t, g_t, s_{t+1}) = s_t + g_t - s_{t+1}$$

Since the primitive goal g_t is the entire state and is relative to the agents current state it follows that HIROs reward is:

$$r(s_t, g_t, s_{t+1}) = -||s_t + g_t - s_{t+1}||$$

One of HIRO's main contributions is that it provides the HRL space with a highly sample efficient method. To accomplish this sample efficiency it uses twin delay deep deterministic policy gradients (TD3) [Fujimoto et al., 2018] to train its policies, which is an off-policy temporal difference RL technique making it quite sample efficient. However, this introduces a problem: because HIRO uses an off-policy learning method it requires a replay buffer which would ideally store the transition tuples from old rollouts, but the primitive behaviour is constantly changing, as such learning from old experience in the replay buffer will likely lead to issues since newer iterations of the primitive will take different actions leading to different transition tuples. Nachum et al. addresses this by creating a novel method called *off-policy correction*. This method achieves a more stable replay buffer by relabeling the controller action (or goal g_t) to a new action $\widetilde{g}_t$ such that the transition tuple in the replay buffer (which was produced by an old primitive) is more likely to be produced with the current primitive. More formally given a transition tuple $(s_t, g_t, \sum R_{t:t+c-1}, s_{t+c})$ HIRO re-labels g_t to $\widetilde{g}_t$ such that the probability $\mu^p(a_{t:t+c-1}|s_{t:t+c-1}, \widetilde{g}_{t:t+c-1})$ is maximised (where μ^p is the current primitive policy).

2.2.2.4 Stochastic neural networks for HRL Stochastic neural networks for HRL (SNN4HRL) is a hierarchical method that is most closely related to the options framework, that leverages pretrained policies to improve sample efficiency when learning multiple complex skills that utilize similar primitives [Florensa et al., 2017].

This framework takes the opposite approach to HIRO by separating the training of primitives and controllers. Initially, multiple primitives are trained (pretrained) to learn a diverse set of skills. This is done using a simpler reward than the target environments. Once primitives are trained they can be used for multiple different environments by training one controller which learns when and for how long to activate each primitive. It is important to note that once the *pretraining* of primitives is complete they no longer learn, meaning that the controller must simply learn which primitive is best in which scenario, but primitives cannot specialize into specific scenarios. As the name implies one of SNN4HRL's main contributions is the use of

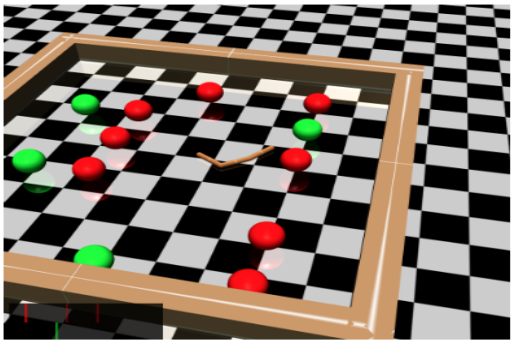
stochastic neural networks [Ginzburg and Sompolinsky, 1994] as policies, they claim their increased expressiveness greatly benefits the task performance gains observed, however, their use is beyond the scope of this work.

The general HRL methods followed in SNN4HRL are typical to the options framework, but is in stark contrast to HIRO, which learns a controller and primitive simultaneously and a new primitive for each task. Each method has its benefits, pretraining for multiple tasks can reduce the number of samples required, thus increasing sample efficiency, but because the primitives are frozen once pretraining is complete they are unable to specialize to specific scenarios and may end up producing sub-optimal results in certain circumstances.

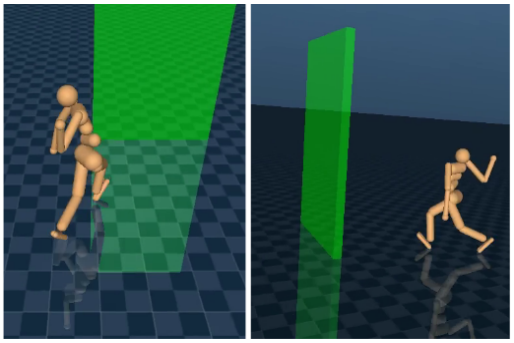
The way in which SNN4HRL pretrains its primitives is not uncommon in HRL [Heess et al., 2016] and is not incompatible with HIRO and feudal-RL frameworks in general. It is simply a matter of generating a simpler representation of the hard environment and using that environment to train one or more primitives. Once the primitives have mastered the simple environment they are frozen and can be transferred to any HRL framework by allowing the controller to activate or instruct the primitive(s). This is a form of transfer learning and a similar method will be investigated as a way to improve both the sample efficiency and performance of the method this work proposes.

2.2.2.5 Hierarchical reinforcement learning environments A large issue facing the HRL community is that there seems to be a lack of standardised environments, especially for HRL methods that target locomotion and navigation problems. However, there has been some recent effort in tackling this in the form of more grand challenge problems [Guss et al., 2019a,b, Kuttler et al., 2020]. The use of standardised environments is commonplace in flat RL. The MuJoCo [Todorov et al., 2012] suite of physics-based locomotion tasks and the arcade learning environment [Bellemare et al., 2013] has become an excellent benchmark for flat RL methods, however, equivalent benchmarks do not yet exist for HRL. Many of the current environments (which can be seen in figure 5) were created to show the efficiency of a specific method against other methods, which lead to many environments being created with minor variations on existing environments. This gave rise to environments that lack benchmarks and are being slowly forgotten, but more grand challenge style environments should solve this issue and provide some standardised HRL benchmarks.

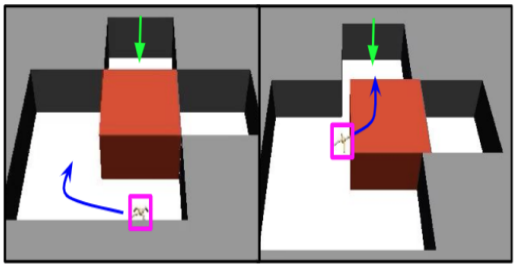
A contributing factor to the lack of standardised environments is that there are many different areas where HRL is applicable and thus many different environments that can and have been created to simulate these areas. One area that has seen much attention is complex robotic tasks. These kinds of tasks require that an agent can both control the fine motor functions of the robot and perform some complex



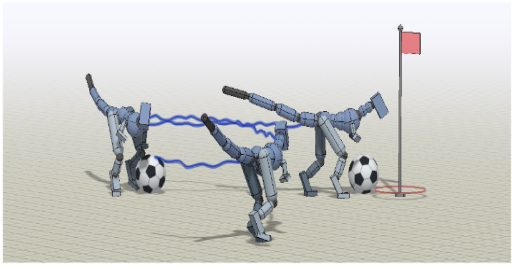
(a) [Florensa et al., 2017]



(b) [Heess et al., 2016]



(c) [Nachum et al., 2018]



(d) [Peng et al., 2019]

Figure 5: Image (a) is the snake gather environment used in SNN4HRL [Florensa et al., 2017]. The snake must gather green food and avoid red food. (b) is the humanoid slalom task by Heess et al. where the humanoid must run as fast as possible through the green areas without falling over. (d) is HIRO's [Nachum et al., 2018] ant push environment, the ant must push the red block out of its way to reach its goal. (d) is the T-rex dribble environment used by Peng et al., the T-rex must learn to dribble the ball towards the flag.

tasks such as navigating a maze or moving an obstacle. This also naturally lends itself to HRL style control since primitives can be created to learn motor control and higher-level controllers can direct the primitives in such a way that they perform the desired complex actions. However, many different environments have been created to show the performance of agents in these kinds of areas. SNN4HRL used a maze and food gathering style task [Florensa et al., 2017] with multiple different robot bodies. HIRO used a subset of SNN4HRLs environments and added more complex mazes with moving walls [Nachum et al., 2018]. An extension to HIRO, Near-Optimal Representation Learning [Nachum et al., 2019] uses environments that require agents to move block-like objects to specific locations. In work by both Heess et al. and Peng et al. agents learn to play football-like games by dribbling balls to certain locations. To add to this certain works use the same environments, but with different robots making it impossible to compare results as a humanoid or T-rex style robot has many more joints than an ant or snake style robot, greatly increasing problem difficulty. An example of these environments can be seen in 5.

All this is meant to show how fragmented the landscape of HRL environments is and that there is a lack of standardised benchmarks making it difficult to compare multiple different methods against one another. It should be noted that many different environments are not necessarily a bad thing, however, the fact that many papers use slightly modified versions of each environment and most papers do not use the same environments as previous papers is where the problem lies.

As mentioned above grand challenge style environments may address this issue as they are difficult to solve and as such can survive as a benchmark for multiple years. Two of the most significant RL grand challenge style problems are the *MineRL* and *NetHack* environments [Guss et al., 2019a,b, Kuttler et al., 2020]. The MineRL is an exceedingly challenging problem based on the popular Minecraft game [Guss et al., 2019a,b]. This game is open-ended and allows the player to perform many diverse actions, such as building, mining, farming and fighting. The *open-endedness* of Minecraft leads to a very challenging game since the agent can take many different actions in a large world, along with this the agent needs to be rather sample efficient as the environment is not particularly fast to simulate. However, given the multitude of skills required to complete this environment, it lends itself very well to being tackled by an HRL algorithm. The NetHack environment is based on the very challenging role-playing game of the same name. The challenge of this game comes not only from the fact the player has very many actions that it can take, but also that the player requires certain items (such as keys) to progress to certain areas, meaning that it is challenging in terms of long term credit assignment. This game is considered to be an incredibly hard game by human standards [Kuttler et al., 2020], which means it's an excellent testing ground for RL agents. One of the most

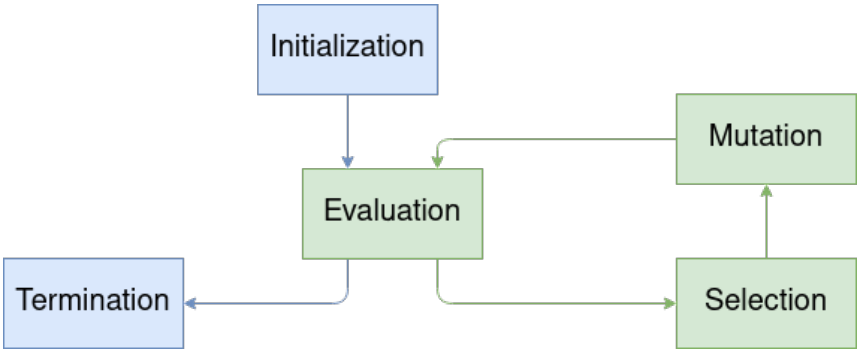


Figure 6: The broad process that most evolutionary algorithms follow: the population is initialized and run through the evaluation, selection and mutation loop, in order to apply selective pressure onto all individuals that should lead to improved overall task performance. This process is repeated until a termination condition is met.

significant things about this game is that it is procedurally generated, meaning that each run generates a new world for the agent to explore, which forces the agent to learn general skills that transfer to any possible world that could be generated. NetHack has the added benefit of being ASCII rendered meaning that it is fast to simulate, thus not requiring expensive hardware to run and allowing quick iteration on algorithms.

These grand challenge environments provide an excellent benchmark for HRL algorithms and are hard enough to remain a relevant benchmark for several years. However, this work is heavily based on previous locomotion HRL methods and as such experiments are performed using some of the locomotion environments mentioned above.

2.3 Evolutionary computation

Evolutionary computation is a family of optimization techniques that are strongly or loosely inspired by biological evolution and Darwin's survival of the fittest [Eiben and Smith, 2015a,b]. Genetic algorithms [Whitley, 1994, Vose, 1999], swarm intelligence [Brambilla et al., 2013, Şahin, 2004, Dorigo and Di Caro, 1999] and evolution strategies [Beyer and Schwefel, 2002] all fall under the evolutionary computation banner and all loosely follow the process outlined by figure 6, however, each has specific tasks to which they are best suited. Genetic algorithms take the most inspiration from biological evolution relying on operators such as mutation, selection and crossover to evolve its genotype and have been extensively used in areas such as autoML [Acton et al., 2020, Real et al., 2019] and neuro-evolution [Stanley and

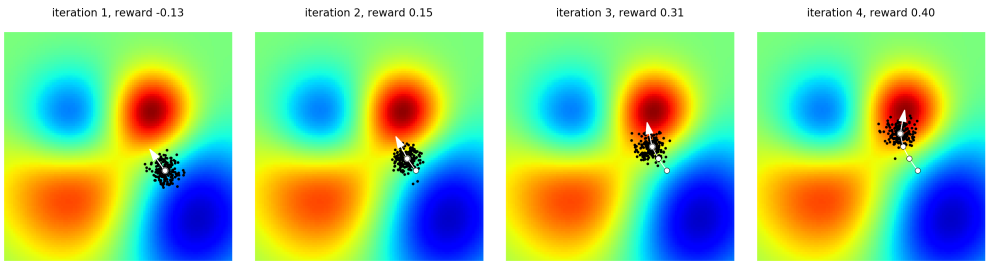


Figure 7: A visual depiction of ESs guess and check approach. Where the white dot is the current policy ($\mu = 1$) and the black dots are the *guesses* or perturbations of the current policy. The red regions are areas of high reward. Notice how the policy updates in the direction of the highest reward experienced by the *guesses*. (Image from Karpathy et al. [2017])

Miikkulainen, 2002, Stanley et al., 2009]. Swarm intelligence is inspired by biological systems, such as ant colonies [Dorigo and Di Caro, 1999] and works by giving many agents simple rules and allowing them to interact locally, which often leads to emergent global intelligence. This has seen success in swarm robotics [Brambilla et al., 2013, Şahin, 2004] and forecasting problems [Schumann et al., 2019]. Evolution strategies (ES) is likely the least based on biological evolution processes and mostly a mathematical optimization process with loose inspiration from biological evolution. It works similarly to genetic algorithms, but has a more simplistic mutation operator and usually no crossover. ES has many applications such as minimizing the drag of 3D bodies [Beyer and Schwefel, 2002], optimizing designs in structural and mechanical engineering problems [Datoussaïd et al., 2006], robot locomotion [Salimans et al., 2017, Conti et al., 2017, Katona et al., 2021] and loss function optimization [Gonzalez and Miikkulainen, 2020].

In the following sections, we discuss evolution strategies, their history as an optimization technique, the current state of the art evolution strategies. We go into depth about Scalable Evolution Strategies (S-ES) upon which most of this work is based. We go on to introduce novelty search and quality diversity, both are powerful behavioural diversity and exploration vs exploitation techniques for evolutionary computation.

2.3.1 Evolution strategies

Evolution strategies are a family of evolutionary computation techniques first published by Rechenberg in 1965 [Rechenberg, 1965]. More specifically ES is a black box optimization approach that uses a population of individuals along with a guided random search in order to optimize a given objective. Individuals are encoded as a list

of parameters, similar to other evolutionary computation methods. Early ES were relatively simple and had only two rules: first, randomly add a small perturbation to all parameters of every individual and second, only keep a new set of parameters if it improves performance. This can be simplified to a *guess and check* approach as seen in figure 7. Initially, the ES *guesses* new parameters (the black dots in figure 7) and then *checks* to see if those parameters improve the task performance.

More rigorously, in each generation of a general ES, a population of μ individuals (encoded by parameter vectors $\theta \in R^n$) are mutated to produce a population of γ child individuals. These individuals are evaluated according to an objective function and assigned a fitness value. The γ child individuals are added to the μ parents then the γ worst performing individuals are discarded from the population to keep it a constant size. This outlines the typical $(\mu + \gamma)$-ES [Beyer and Schwefel, 2002]. Another popular classical form of ES is (μ,γ)-ES where instead of adding the γ children to the μ parents and performing selection on the combination of the two, the parents are forgotten and replaced by the best performing children each generation. This requires that $\gamma > \mu$ and if hyperparameters are not tuned well can lead to divergent behaviour.

In any ES the distribution from which noise is sampled in order to perturb an individual plays a pivotal role. The most commonly used distribution is the Gaussian distribution, as this was shown very early on to have desirable performance [Beyer and Schwefel, 2002]. Since ESs are usually applied to high dimensional problems, a multi-variate Gaussian is used. The distribution is closely tied to the mutation strength parameter σ as they both play a critical role in the convergence of the ES. The mutation strength is the magnitude at which each parameter is perturbed and is similar to a learning rate from deep learning. If the mutation strength σ is too high then the ES will likely diverge and if σ is too low then the ES will likely take too long to converge or will not be explorative enough to find areas of high performance. In the case of a Gaussian, the mutation strength is the standard deviation of the distribution and in the case of a multi-variate Gaussian, it is its covariance matrix. Since mutation strength is so important to the task performance of an ES, mutation strength adaption methods have been developed which change the mutation strength over the course of evolution. Rechenbergs one-fifth rule has been shown to work well here for some problems [Beyer and Schwefel, 2002], however a more performant subset of ES have emerged in order to solve the mutation strength problem called self-adaptive ES. These are ES that evolve their own hyperparameters along with the problem-specific parameters. This is most commonly used on the mutation strength to ensure that it is at a near-optimal value in every generation, however, it can be applied to any hyperparameter of the ES.

Since 1965 there have been many variants of ES, three of the most popular being

covariance matrix adaption evolution strategies (CMA-ES) [Hansen and Ostermeier, 2001], natural evolution strategies (NES) [Wierstra et al., 2014] and Scalable evolution strategies (S-ES) [Salimans et al., 2017, Conti et al., 2017]. CMA-ES is a self-adaptive ES that automatically adapts its mutation strength, represented by a covariance matrix, allowing for optimization of the mutation strength of each parameter of the ES. A large downside of this approach is that it can be slow for large search spaces as adapting a large covariance matrix is computationally expensive. NES are a family of evolution strategies that utilize an estimated natural gradient [Wierstra et al., 2014] in order to stabilize their search direction. S-ES is simpler and like NES approximates a gradient, however, unlike NES it does not approximate the natural gradient, but rather the stochastic gradient and uses modern techniques such as virtual batch normalization [Salimans et al., 2016] which greatly improves its task performance [Salimans et al., 2017]. All of the methods mentioned above have shown state-of-the-art performance in at least one domain [Wierstra et al., 2014, Hansen and Ostermeier, 2001, Salimans et al., 2017, Conti et al., 2017].

2.3.1.1 Scalable Evolution Strategies S-ES was popularised by OpenAI when they showed that its performance was comparable to the state-of-the-art policy gradient methods on robot locomotion and Atari game playing tasks [Salimans et al., 2017]. Interestingly it was also used to test how well novelty search [Lehman and Stanley, 2008, 2011a] works on hard problems requiring large ANNs [Conti et al., 2017]. S-ES is very similar to $(\mu + \gamma) - ES$. It has a population size of a single individual ($\mu = 1$). Individuals are encoded by a vector of parameters ($\theta \in R^n$) which, in the case of this work, are the weights and biases of an ANN. The main policy is mutated γ times by sampling *noise vectors* $\epsilon_i \; for \; i = 1, ..., \gamma$ from an n-variate Gaussian distribution and adding them to θ_t.

$$\theta_{t_i} = \theta_t + \sigma\epsilon_i \mid \epsilon_i \sim \mathcal{N}(0, I)$$

where σ is the mutation strength, θ_{t_i} is the i^{th} perturbation of the main policy's parameters (θ_t in the t^{th} generation) and I is an n x n identity matrix representing the diagonal of the covariance matrix of the n-variate Gaussian. Conceptually the mutation process can be thought of as creating a cloud of γ points around θ_t in parameter space. All γ individuals are evaluated and assigned a fitness value, but instead of removing the γ worst performing individuals like (μ+γ)-ES, S-ES instead uses this information to approximate the gradient and in the process removes all variants of θ_t, similar to natural evolution strategies [Wierstra et al., 2014]. This is done by weighting each individual's parameters by the fitness it achieved, summing the weighted parameters and moving θ_t in this direction.

Algorithm 1: S-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ

for *t = 0,1,2...* **do**

  Sample $\epsilon_1, ...\epsilon_n \sim \mathcal{N}(0, I)$

  for *i = 1,...,n* **do**

    Compute fitness $F_i = F(\theta_t + \epsilon_i * \sigma)$

  end

  Set $\theta_{t+1} = \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$

end

This on its own is not a particularly novel approach to ES, in many ways, it is a simpler version of NES [Wierstra et al., 2014], the main contribution comes from the way in which the algorithm is parallelised which provides what Salimans et al. claim is a linear speedup. Given that S-ES operates on complete episodes and each perturbed policy is not concerned with the performance of other policies it is embarrassingly parallel Herlihy et al. [2020], thus the approach to parallelising S-ES aims to minimize the amount of communication across processors on a CPU and nodes in a cluster. This is achieved by sharing random seeds instead of sharing the entire noise vector (ϵ_i) used to perturb the parameters θ_t. Thus one can share a large noise vector (of over one million items) with only a single number i which greatly reduces communication bandwidth. In practise this is done by creating a large randomly generated *noise table* in memory and sharing it across each node only once, thus each seed becomes an index i into this *noise table* representing a subarray starting from i until the length of parameter vector $|\theta|$. This allows for very quick access to the random numbers on all nodes during training and updating of the policy.

The speed-up of the parallelised version was shown to be linear in trend up to 1440 cores at which point testing stopped. However, it is unlikely that S-ES is perfectly linear as there are unavoidable sequential parts of the algorithm that would not allow this. When given all 1440 cores S-ES was able to solve the humanoid MuJoCo environment (see figure 8) in 10 minutes [Salimans et al., 2017], compared to Asynchronous Actor-Critic (A3C) which took a day to obtain the same results [Mnih et al., 2016]. This speed allows S-ES to be applied to much harder tasks without time being a limiting factor in experimentation.

Parallelism was not the only contribution of Salimans et al. while the parallelism

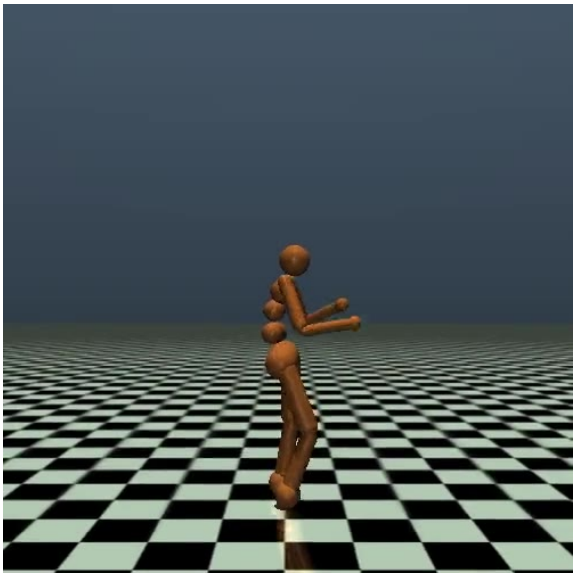


Figure 8: The MuJoCo humanoid, is one of the common benchmarks for RL algorithms. The agent must learn to walk as far forward as possible while using as little *energy* as possible.

greatly improved the speed at which S-ES could solve RL problems, Salimans et al. also brought two improvements from modern RL that greatly improved performance, namely the ADAM optimizer [Kingma and Ba, 2014] and virtual batch normalization [Salimans et al., 2016]. Both algorithm 1 and 2 have been simplified for readability, but in the final line of both of these algorithms, the policy is updated directly by the perturbations weighted by the fitness it achieved, however in practice this value is used as the approximate gradient and fed into the ADAM optimizer as if it was a real gradient, in informal experiments this greatly improved performance. Virtual batch normalisation was the key improvement that allowed S-ES to match the performance of policy gradients. Virtual batch normalisation normalises the input observations based on a reference set of observations, in S-ES the reference observations are updated using random observations throughout training. Usually, virtual batch normalisation is quite expensive however in the case of S-ES its performance impact is negligible given the number of steps taken during an episode.

Salimans et al. state that one of the main advantages of S-ES is its more structured exploration when compared to trust region policy optimization (a flat RL method by Schulman et al.) [Salimans et al., 2017]. This is likely because S-ES has a fundamentally different method of exploration to most gradient-based RL methods. There is no explicit exploration hyper-parameter, simply a mutation strength σ which determines how far, in parameter space, the child parameters should be from

Algorithm 2: Parallelized S-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ

Initialize: n workers each with access the same *noise table* and initial parameters θ_0

for $t = 0, 1, ...$ **do**
  for *each worker* $i = 1, ..., n$ **do**
    Sample ϵ_i, $seed_i \sim \mathcal{N}(0, I)$
    Compute fitness $F_i = F(\theta_t + \epsilon_i * \sigma)$
  end
  Share all fitnesses and perturbations (F_i, $seed_i$) between all n workers
  for *each worker* $w = 1, ..., n$ **do**
    for $i = 1, ..., n$ **do**
      Reconstruct each ϵ_i using $seed_i$
    end
    Set $\theta_{t+1} = \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
  end
end

the parent parameters θ_t. This can be thought of as a brute force style of exploration, by moving to a region of high reward and exploring the surrounding region for even higher reward. This is especially beneficial to HRL because environments which require a hierarchical approach are often difficult because they have sparse rewards, which usually requires special exploration methods in order to be solved.

2.3.1.2 Improving sample efficiency Since ES is a black box method that is invariant to temporal details (meaning it requires no temporal discounting) it naturally takes many samples in order to learn a highly performant policy [Sigaud and Stulp, 2019]. S-ES is no exception to this, it attempts to approximate the gradient from multiple rollouts of slightly perturbed policies. Each of these rollouts, of course, increases the number of samples taken leading to its poor sample efficiency. S-ES does make up for this with its excellent scalability to multiple processors and thus short wall clock execution times when given enough CPU cores, in many cases being faster than gradient-based RL methods [Conti et al., 2017]. However, there has been work that aims to address this sample inefficiency. Most notably Liu et al. introduced a method for integrating trust regions into S-ES. Trust regions

[Schulman et al., 2015] are an optimization technique distinct from line search (of which gradient methods are a subset) which seek to optimize the function F through a simpler function $\widetilde{F}$ in a region where $\widetilde{F}$ is approximately equivalent to F. One of the most notable uses of trust regions is TRPO [Schulman et al., 2015] an RL algorithm that was used as the base of comparison for S-ES by Salimans et al.. Liu et al. showed that this provided S-ES with a monotonic improvement and improved its sample efficiency while having minimal impact on the scalability and speed of S-ES [Liu et al., 2019]. In three out of the five environments tested S-ES with a trust region was more sample efficient than one of the most popular gradient-based RL algorithms: proximal policy optimization (PPO) [Schulman et al., 2017]. This method was not used in this work but is a testament to the fact that S-ES can be a sample efficient algorithm.

2.3.2 Novelty Search

Novelty search [Lehman and Stanley, 2011a] falls under a broader category of algorithms called behavioural diversity [Mouret and Doncieux, 2012]. These algorithms do exactly as the name implies, they explicitly search in the behaviour space for diverse behaviours, by making novelty the objective of an evolutionary algorithm. Novelty search allows an algorithm to avoid any deception (local minima/maxima) that may have been caused by the fitness function itself [Lehman and Stanley, 2011a]. This can clearly be seen in the middle of figure 9, where a typical fitness function, such as distance to the goal, guides most solutions directly towards the goal and gets stuck on the walls without being able to reach the goal, this represents a local minima/maxima in the fitness landscape. However, novelty search is able to reach the goal and explore the maze much more evenly as can be seen on the right of figure 9. This is the main benefit of novelty search it is an excellent tool for encouraging exploration in evolutionary algorithms. In theory, novelty search can be added to any evolutionary algorithm where a domain-specific novelty metric can be defined as it is a drop-in replacement for an objective function.

Novelty search determines the behaviour of an individual using a domain-specific *behavioural descriptor*. This is a disadvantage of novelty search as it may require domain knowledge to create the behavioural descriptor and in some cases, it may not be possible to create one at all. A behavioural descriptor is commonly a vector that is able to describe an agents behaviour throughout an episode, for locomotion tasks this may be the path an agent has walked throughout the episode or simply its ending position. Once a behavioural descriptor has been obtained it is compared to the k nearest individuals (in behaviour space) and given a novelty score based on its distance to these individuals. If it is novel enough it is added to an *archive* and used as a basis of comparison for new behaviours.

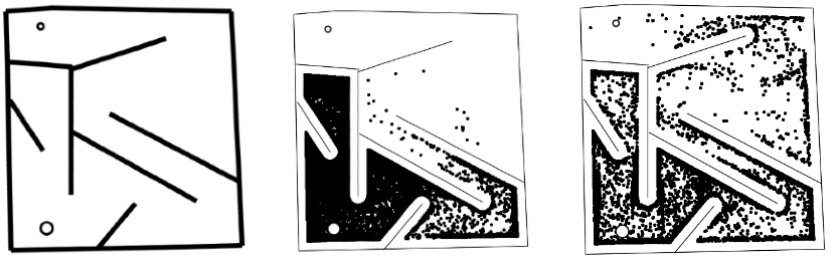


Figure 9: (Lehman and Stanley [2011a]) The *hard map* made by Lehman and Stanley used to show how a fitness function can be deceptive, the goal is for an agent to get from the large circle to the smaller circle. Left: the map with no results on it, middle: the results of a fitness function, where each dot is the ending point of the agent, right: the results of novelty search.

To explain more in-depth as to why this works, consider the locomotion behavioural descriptor mentioned earlier: the ending position or the path of the agent. Initially, an agent that simply falls over would get a high novelty due to the lack of similar behaviours in the archive. But this would quickly become less novel and newer agents would have to learn to walk further and further away in order to obtain higher novelty, as most agents will have ended up close to the starting position. This would also encourage agents to walk in different directions, thus ending up in novel locations, which in many cases is desirable.

Novelty search has a key disadvantage since it optimizes only for the novelty of an individual it does not guarantee good performance, in fact, a purely novelty based algorithm will often perform worse than a purely objective-based algorithm for non-deceptive tasks. This has led to the creation of quality-diversity based algorithms, which merge novelty search and objective search in order to gain the advantages of both. One can think of novelty search forming the exploration side of the exploration-exploitation trade-off in a quality diversity algorithm.

2.3.2.1 Quality diversity Quality diversity algorithms [Pugh et al., 2016] are evolutionary algorithms that, as the name implies, search for diverse **and** high-quality behaviours. This is in contrast to novelty search which only searches for diverse behaviours. That being said quality diversity algorithms do use novelty search in order to maintain diversity. The simplest form of a quality diversity algorithm would simply use novelty in place of the current speciation technique, such as a multi-objective evolutionary algorithm [Mao-Guo et al., 2009] with one objective being novelty and the other being a domain-specific fitness function. However, algorithms have been developed which extend this relatively simple multi-objective quality diversity method, namely MAP-elites [Cully et al., 2015] and novelty search

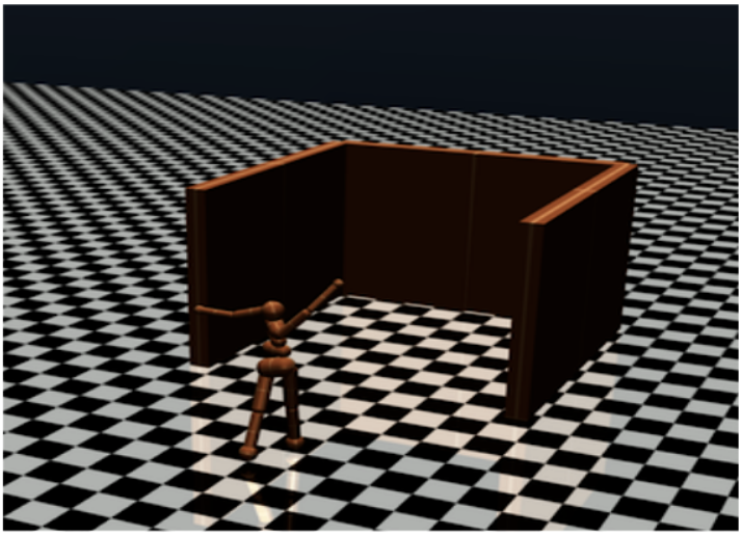


Figure 10: (Conti et al. [2017]) The deceptive walker task used in NSRA-ES to show the types of environments where novelty search is most beneficial.

with local competition (NSLC) [Lehman and Stanley, 2011b]. MAP-elites keeps track of the best solutions in terms of numerous user-defined objectives, whereas NSLC tracks novelty globally, but separately compares objective fitness in a local niche.

In this work, we will use a very naive implementation of novelty search, which attempts to directly tackle the exploration/exploitation trade-off. By normalizing both objective fitness and novelty to the same scale one can simply weigh novelty highly when exploration is more important, conversely when exploitation is more desirable objective fitness can be weighted higher. This allows one to tackle the exploration-exploitation trade-off in a more traditionally RL manner, such as using an ϵ-greedy approach or simply increasing exploration when a policy seems *stuck*.

2.3.2.2 Novelty and Scalable Evolution Strategies Conti et al. extended S-ES by integrating novelty search in a similar way to the naive method mentioned in the previous paragraph. It was done to test how novelty search would scale to large ANNs and deceptive locomotion problems like the one seen in figure 10. Along with this, it adds a new method of directed exploration to ES, as prior to this directed exploration was not possible with ES [Conti et al., 2017].

Conti et al. introduced three methods one purely novelty search: NS-ES, and two quality diversity methods: novelty search reward ES (NSR-ES) and novelty search reward adaptive ES (NSRA-ES). This family of methods modifies S-ES slightly by

adding a population of *main* policies, but still retains S-ES's scalability and speed. A *main* policy is selected as the policy to be optimized in the current generation based on its novelty in previous generations, such that higher novelty leads to a higher selection chance. This encourages each policy to explore different areas of the search space thus providing a more accurate novelty metric.

NS-ES is simply S-ES with a novelty reward instead of an objective reward. NSR-ES is S-ES with a hyperparameter defined weighting between objective and novelty. This works by initially normalizing the fitness and novelty of each individual such that they are on the same scale (for example $[0, 1]$) and then multiplying fitness by w and novelty by $1 - w$. NSRA-ES is a bit more intricate in how it works, initially $w = 1$ so all the weighting is given to objective fitness. If after n generations there has been no improvement in the fitness of an individual then w is reduced by 0.05, this puts more weighting on novelty thus encouraging the agent to explore and ideally escape the local minimum. If the agent improves on its previous best, indicating that it is escaping the local minima, then w is increased by 0.05.

NSR-ES and NSRA-ES showed high task performance on both a deceptive locomotion task and a subset of the Atari suite of games, improving on both gradient methods and vanilla S-ES for most tasks tested. However, this does come at the cost of sample efficiency since this family of methods requires more than one main policy, the performance of all but the best main policies are essentially discarded as they only serve to improve the novelty calculation. That being said, when performing $10\times$ the number of steps the NS-ES family of methods only took around 2 hours to complete their tasks, whereas gradient-based solvers took one or multiple days to learn from 10x fewer samples [Conti et al., 2017].

2.3.2.3 EvoRBC Another method that makes use of novelty search is Evolutionary Repertoire-based Control (EvoRBC) [Duarte et al., 2016, Gomes et al., 2018] it is an evolutionary algorithm intended to control robots with arbitrary locomotive complexity. It achieves this by mapping desired behaviours to primitives, so that the low-level actions of a primitive are abstracted away from the controller. Mapping is done in such a way that the controller needs to only specify an angle and a distance and the closest matching primitive will be selected to carry out the action, this could be seen as a variant of the Options framework (discussed in section 2.2.2.1) [Sutton et al., 1999b]. In order to abstract away the low-level components evoRBC needs to generate a large number of primitives (called a repertoire) so that the controller has access to a primitive for every action, it may want to take. The repertoire is generated using MAP-elites, although other similar implementations have simply used novelty search [Gomes et al., 2018, Gomes and Christensen, 2018]. In the original implementation of evoRBC [Duarte et al., 2016], both the primitives and controller

used NeuroEvolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002]. However, Gomes et al. found that decision trees worked significantly better as controllers [Gomes et al., 2018, Gomes and Christensen, 2018]. The use of decision trees provided not only a performance boost, but the controller became much more understandable as ANNs are a black box method. This is likely because controllers simply have to make discrete choices between each policy, decision trees would be unlikely to work if the controller needed to pass any information to the primitive.

2.4 Summary

In this section, we discussed the basics of neural networks, the use of evolution strategies (ES) as an optimization technique and the sub-field of reinforcement learning (RL) known as hierarchical reinforcement learning (HRL).

ES are a set of black-box optimization techniques that apply a guess-and-check approach to optimize some set of parameters, often belonging to an artificial neural network. We discuss in depth the scalable evolution strategies (S-ES) method, which is notable as it is highly scalable leading to fast (run-time) solutions when compared to gradient-based methods. Along with discussing evolution strategies we also cover novelty search, an exploration and *deception avoidance* which can allow evolutionary RL algorithms an *exploration-exploitation* trade-off in the style of a gradient-based RL algorithm.

Most of the focus of the HRL section is Feudal reinforcement learning, a HRL method that allows communication between parent and child policies in a policy hierarchy. The framework proposed by this work is based on a modern iteration of Feudal RL, namely HIerarchical Reinforcement learning with Off-policy correction (HIRO). This framework brings an off-policy approach to HRL thus greatly improving sample efficiency when compared to other methods and achieving state-of-the-art results on the four environments that are used in this work. We also investigate transfer learning and its application in RL and HRL.

While novelty search has been shown to work for simple deceptive problems, such as navigating a maze [Lehman and Stanley, 2011a], it has not been shown to work in complex HRL environments. It provides an interesting way to improve exploration for problems, which by their nature are particularly hard exploration problems. Transfer learning for HRL provides a possible avenue for both improving the sample efficiency and performance of HRL methods, thus transfer learning will be investigated as a potential alternative to a purely feudal RL approach.

As of yet, S-ES has not been extensively tested on hard HRL problems, especially in the navigation and locomotion domain. The reason for this is likely twofold: first,

S-ES's poor sample efficiency would be compounded in hard HRL task environments and second, Salimans et al. showed that S-ES was competitive with gradient-based methods, but importantly it did not achieve higher performance in all cases thus providing little incentive to use S-ES as a policy optimizer in HRL algorithms. Merging S-ES with an HRL method also has the potential to bring S-ES's advantages (indifference to delayed reward, robustness to hyperparameters and scalability) to HRL algorithms. This would complement the current state-of-the-art HRL algorithms as while they are sample efficient, they are much less scalable and thus have long run-times. As such the following section describes a method for using S-ES in a feudal reinforcement learning context.

3 Method

In this section, we present our framework for learning hierarchical policies using S-ES, which we call Scalable Hierarchical Evolution Strategies (SHES). We show how current gradient-based HRL methods needed to be adapted in order to work with S-ES and explain the important design choices taken, such as choice of the primitive reward function, the goal encoding, controller architecture, how policies are mutated and how noise is sampled.

At a high level SHES operates using two policies, a controller and a primitive. The controller is responsible for outputting a goal position that a robot is required to reach and the primitive directly takes actions in the world in order to reach this goal. The controller and primitive receive different rewards, but are trained at the same time using Scalable Evolution Strategies (S-ES) [Salimans et al., 2017]. We test SHES using environments where an ant robot needs to navigate around an area to achieve some goal.

3.1 Policy Hierarchy

SHES is a Feudal RL [Dayan and Hinton, 1993] style method where a higher-level policy sets goals for and provides rewards to a lower-level policy. In the original feudal RL work Dayan and Hinton, use a multilevel feudal hierarchy, but SHES uses a two-level hierarchy consisting of a higher-level controller policy μ^c and a lower-level primitive policy μ^p, similar to HRL with Off-Policy Correction (HIRO) [Nachum et al., 2018], which is a more recent iteration of feudal RL. The controller is responsible for setting goals and cannot directly perform actions in the world, while the primitive directly controls the agent by taking actions in the world with the aim of achieving the goals set by the controller. More formally, given a state s_t from the environment the controller produces a goal $g_t \in \mathbb{R}^d$, where d depends on the goal encoding (see section 3.3 for more information regarding SHES' goal encoding). The controller produces g_t every c steps, where c is a hyperparameter known as controller interval. In the interim, the goal is transformed using a static function such that it is always relative to the current state, an example of this can be seen in lines 8 and 9 of algorithm 4. The controller interval c is kept as a hyper-parameter since it has been observed that learning c often leads to it degenerating into the simplest cases where c becomes 1 or the maximum episode length [Vezhnevets et al., 2017].

The controller's time between actions provides it with a level of temporal abstraction which, in the case of the environments tested in this work, allows the controller to plan a path without worrying about how the agent will follow this path. The primitive is passed the goal g_t and the state s_t and is tasked with reaching the goal. It samples an action $a_t \sim \mu^p(s_t, g_t)$ from its policy which is applied to the agent. The

controller receives a reward from the environment, however, it is also responsible for rewarding the primitive. As discussed in section 3.2 the primitive is sensitive to this reward and thus it should be chosen carefully. In HIRO, FeUdal Networks for HRL (FuN) [Vezhnevets et al., 2017] and this work, the primitive reward is based on its distance to its goal g_t, however, all such previous work makes use of different rewards [Nachum et al., 2018, Vezhnevets et al., 2017].

In a strict feudal RL setting, rewards are not shared between controller and primitive. For example, if the primitive reaches the goal set by the controller, but this does not provide a high environmental reward then the primitive will receive a high reward, but the controller will not, this is known as *reward hiding* and was discussed in section 2.2.2.2. Both this work and HIRO follow this strict style of primitive rewards, however, FuN does not as it shares rewards between its primitives and controllers [Vezhnevets et al., 2017], this was decided against as it introduces a hyper-parameter to balance primitive and controller reward scales.

Our method is similar to HIRO, though SHES differs in its lack of off-policy correction, its primitive goal encoding and the use of S-ES to train the primitive and controller. The main difference between SHES and S-ES being that there are now two policies that co-evolve instead of a single policy. SHES stores a set of parameters for both the controller θ^c and primitive θ^p. Every generation it creates n new pairs of controllers and primitives by perturbing the parameters θ^c and θ^p. The perturbation is done by adding a small amount of noise sampled from an n-variate Gaussian to the parameters $\theta_i^c = \theta^c + \epsilon^c \sim \mathcal{N}(0, \sigma^2)$. It should be noted that S-ES and as a result SHES uses antithetic or mirrored sampling [Ren et al., 2019] of the noise vector in order to reduce variance, this will be discussed further in section 3.7. The primitive is perturbed similarly using a different noise vector which is sampled from the same Gaussian $\epsilon^p \sim \mathcal{N}(0, \sigma^2)$. When the perturbation is performed using random seeds, as it is in this work, it allows for the sharing of common random numbers at negligible extra memory cost when compared to a single policy S-ES. The sharing of common random numbers using random seeds was shown by Salimans et al. to allow for near-linear speedup when scaling up to 1440 CPU cores [Salimans et al., 2017]. Seeing as SHES shares these random numbers in the same manner as S-ES we expect SHES to have similar speedup characteristics to S-ES, which would provide SHES with one of its main benefits: a scalable and fast HRL algorithm.

Every pair of the perturbed controller and primitive is evaluated in the environment such that the controller is given a fitness equal to the cumulative environmental reward and the primitive is given its fitness as its cumulative reward from its controller. Both primitives and controllers are separately ranked and shaped according to their fitness using the method discussed in section 2.3.1.1. The ranked and shaped fitnesses are then used to approximate the gradients for the controller and primitive,

which are separately optimized using the ADAM optimizer [Kingma and Ba, 2014] and updated in the same manner as S-ES.

In a feudal RL method the controller and primitive learn simultaneously, this means that the controller is required to learn not only how to solve the problem, but also how best to recommend goals to the current iteration of the primitive. This amounts to a non-stationary problem for the controller since while it is trying to learn how to recommend goals to the primitive, the primitive is learning how best to accomplish its goals, thus constantly changing its behaviour. These non-stationary problems are particularly challenging, hence Nachum et al. developed an off-policy correction method for HIRO in order to combat the non-stationary problem and allow for off-policy training leading to better sample efficiency. We found that a special method to combat this problem was not necessary for SHES as S-ES's robustness to noise makes this problem trivial to solve. The controller can simply interpret the primitives changing behaviour as noise, however, this does come at the cost of sample efficiency.

Algorithm 3: SHES

Input: Learning rate α, noise standard deviation σ, rollouts n, initial policy parameters θ^c and θ^p

for *t = 0,1,2...* **do**

  for *i = 1,2,...,n* **do**

    Sample $\epsilon_i^c, \epsilon_i^p \sim \mathcal{N}(0, I)$

    $f_i^c, f_i^p = F(\theta_t^c + \epsilon_i^c * \sigma, \theta_t^p + \epsilon_i^p * \sigma)$

  end

  $\theta_{t+1}^c = \theta_t^c + \alpha \frac{1}{n\sigma} \sum_{i=1}^n f_i^c \epsilon_i^c$

  $\theta_{t+1}^p = \theta_t^p + \alpha \frac{1}{n\sigma} \sum_{i=1}^n f_i^p \epsilon_i^p$

end

3.2 Primitive Reward

The manner in which the primitive is rewarded can have a large impact on the overall performance of SHES, at the most basic level the reward needs to incentivize the agent to reach a target. In the literature, there are many different ways that the primitive reward has been formulated [Vezhnevets et al., 2017, Nachum et al., 2018, Coumans et al., 2013], from this and our own informal experiments we've found the main components of a well-performing primitive reward are incentive to reach the target consistently and quickly while avoiding local minima.

Algorithm 4: Controller/primitive evaluation (F from algorithm 3)

Input: primitive ANN *pnn*, controller ANN *cnn*, controller interval c, environment *env*, max environment steps n
abs_g_t = [0, 0]
controller_reward, primitive_reward = 0, 0
for t = *0,1,...,n* **do**
    obs = getobs(env)
    if t mod c == 0 **then**
        g_t = cnn(obs)
        abs_g_t = g_t + agent_position
    g_t = abs_g_t - agent_position
    setaction(pnn(g_t, obs))
    step(env)
    controller_reward += controller_reward(env)
    primitive_reward += primitive_reward(cnn, env)
end
controller_reward, primitive_reward

HIRO uses the most simple primitive reward, by rewarding the primitive with its negative distance to the goal g_t [Nachum et al., 2018], this encourages the agent to move to the target quickly, however, it introduces a challenging local minima where the agent can simply *die* instantly (by falling over) thus avoiding accumulating anymore negative reward. FuN rewards its primitive based on the cosine similarity of the path the agent has taken since the goal was suggested and the straight line from the agent's position to the goal [Vezhnevets et al., 2017]. This encourages the agent to be very consistent which makes it more predictable for the controller, but this reward puts little emphasis on speed. Another possible reward that is used in both pyBullet [Coumans et al., 2013] and MuJoCo [Todorov et al., 2012] locomotion environments is the agent's velocity towards the target, while this does encourage fast movement this often comes at the cost of consistent paths, making it difficult for the controller to recommend positions. Rewarding the agent based on the percent of the total distance it covered (since the position was recommended) plus a bonus for reaching the target was found to be the best performing primitive reward and is what SHES uses.

$$R_t^p = 1 - d_t/d_c + (1\ if\ d_t < L\ else\ 0)$$

where d_t is the euclidean distance between the agent and the goal g_t at timestep t, d_c is the distance at timestep c (the most recent time-step at which the controller recommended a goal) and L is a distance threshold ($L = 1$ in the case of this work). This improves upon simply rewarding the primitive with the negative distance by allowing it to be positive if the primitive performs well thus avoiding local minima and normalizing the distance thus making it agnostic to target distance. Also, adding an extra reward for being close to the target incentivises the agent to reach the goal as quickly as possible in order to maximize the amount of time it receives this extra reward.

3.3 The Primitive Goal

SHES manages the primitive goal similarly to HIRO, in that a new goal is recommended once every c steps by the controller and for the next $c - 1$ steps this goal is transformed using a fixed goal transition function. In each step the current goal g_t is concatenated onto the primitive's observations as seen on line 10 of algorithm 4. In SHES the primitive goal g_t is the vector from the agent's position to the target recommended by the controller. However, the SHES goal differs from the HIRO goal in that it only recommends an x and y position in space, whereas in HIRO the goal passed to the agent is the entire state space, such that the primitive must attempt to match the position of all of the agent's joints as well as the overall position of the agent. This approach is very general, but it limits types of primitive rewards that are usable and it is often more challenging for the primitive to learn this representation.

Similar to primitive rewards we found goal encoding to have a large impact on performance. The most obvious goal encoding to use would be a vector from the agent's position to the goal g_t. We found that this did not work since the values are not normalized, thus the primitive ANN performs worse because of non-normalized input data [Sola and Sevilla, 1997]. However, normalizing the goal vector comes with its own issue since the agent no longer has any notion of distance to the goal g_t. We solve this by concatenating distance onto the normalized goal, the distance to the target which is scaled down to an appropriate range (by dividing it by 1000).

This simple normalized vector goal encoding can be improved upon, taking inspiration from pyBullet's directional encoding [Coumans et al., 2013] we encode the primitive goal as the *sin* and *cos* of the angle from the agent to the goal g_t. This was done by allowing the controller to output a relative vector from the agent to the target and transforming this vector into an angle from the agent to the target, the *sin* and *cos* of this angle is the goal g_t that is passed to the primitive. Similar

to the normalized vector encoding this also provides the primitive with no notion of distance to the target and as such, we pass the scaled distance to the target along with this goal encoding. This angle encoding can be seen as a normalization step.

3.4 Transfer learning based SHES

Transfer learning presents another option for learning hierarchies of policies and as such will serve as an apt comparison to SHES, it could also be seen as an extension to SHES. There are many different options when it comes to transfer learning for RL and HRL [Zhu et al., 2020, Hawasly and Ramamoorthy, 2013, Schaal, 1999] but the approach used for this paper will be akin to pretraining in supervised learning. In order to keep the comparison as fair as possible for SHES, S-ES will be used to train all policies, communication between policies will occur in a feudal RL manner and it will use the same hierarchical layout, we will call this method SHES-TL.

The pretraining will take place only for the primitive as one cannot pretrain a controller without already having a trained primitive. Thus training is split into two parts, first the primitive pretraining and then combined training where the controller makes use of the pretrained primitive. Pretraining will be done in such a way that it allows all the controllers from each of the environments tested to use the same primitive, thus improving sample efficiency since the primitive can be trained once for many similar tasks. In feudal RL information is passed from the controller to the primitive in order for the controller to indicate how it would like the primitive to act. To make this a fair comparison the communication needs to be taught to the primitive during pretraining so it understands the communication during controller training. Given a primitive observation o made up of the environment observation o_e and the controllers communicated instructions o_c, pretraining needs to automatically generate the controller's instructions o_c in a way that promotes generalisation to any instructions the controller could provide. This is done by obtaining the upper and lower bounds for each value in the controller instructions o_c and randomly generating them from a uniform distribution at the same interval that a controller would provide its instructions. One can view this as using feudal RL to train the primitive using a completely random controller. We train the primitive in the ant flagrun environment (section 4.1.2). Once the primitive has achieved the desired performance it is saved and used as the primitive for controllers during training on hierarchical environments.

During hierarchical training there are two options as to how the pretrained primitive can be treated, its weights can be unfrozen or frozen meaning that it either continues to learn with the controller or it keeps its performance from the pretraining stage. Informal experiments showed that a frozen primitive performed substantially better

than an unfrozen primitive and as such, this is what will be used when comparing to SHES. We use the same pre-trained primitive in all environments when training SHES-TL.

This method has some benefits over feudal RL training. First, since the primitive does not learn at the same time as the controller it removes the non-stationary problem thus making it an easier problem for the controller to optimize. Second, the pretrained primitive is very general as it was trained using what is essentially a random controller. Third, because of its generality, it can be reused across multiple similar tasks, which in turn improves the sample efficiency of the method. However, pretraining is not without its drawbacks. First, it is likely to use more samples when considering both the pretraining and hierarchical training, because the primitive and controller are not trained at the same time. Second, the generality that the primitive gains from pretraining may impact the overall performance in hierarchical environments where it could be beneficial for a primitive to specialize to the similar instructions it is given each episode by the controller.

As alluded to earlier there are multiple different ways one could measure the sample efficiency of a pretrained RL method. The question is whether to count the pre-trained samples, this becomes especially tricky when the pretrained primitive can be used across multiple environments. In this work, we will discuss results that both include and exclude primitive pretraining samples.

A limitation of pre-training is that the primitive observation space during pre-training must match its observation space during hierarchical training. Since the observation spaces of task environments used in our experiments differ, one cannot train a primitive with the exact same observations as required in each of the environments. As such we extract the common observations for all the environments and create a pre-training environment using these observations. The common observations happen to be all observations directly related to the agent and not the environment, thus a simple environment is suitable for pretraining. This required that the primitive's observations in the hierarchical environment are sliced so the primitive only receives the observations it was pre-trained with. This mandates manual selection of observations and gives transfer learning an advantage as the primitive only receives the observations it needs for learning. Given this advantage and transfer learning mitigating the non-stationary problem, SHES-TL is a suitable comparative method for SHES.

3.5 Novelty based SHES

HRL is generally applied to problems that are too difficult for flat RL to solve, and one of the most common threads in difficult RL problems is that they are *hard*

exploration problems. Thus it is quite natural to try and improve the exploration capabilities of methods being applied to harder RL problems, such as SHES. This is done in much the same way that Conti et al. added novelty search to S-ES in the form of Novelty Search Reward Adaptive ES (NSRA-ES) [Conti et al., 2017], which is discussed in chapter 2.3.2.2.

When considering the types of problems that novelty search was applied to and the types of problems that will be used to test SHES, novelty search seems like a logical extension to add to SHES. Consider the deceptive walker task used to display the exploration capabilities of NSRA-ES as seen in figure 10, this is quite similar to the maze environment that will be used to test SHES (section 4.1.4). Another reason to pursue this avenue is that novelty search hasn't been used in this way for HRL problems. Evo-RBC used novelty search to generate a repertoire of diverse primitives [Duarte et al., 2016], but in this work, it is intended to be used as a means of improving the exploration capabilities of the controller. We name the novelty search extension to SHES developed in this work Scalable Hierarchical Evolution Strategies with Novelty Search (SHES-NS).

The behaviour classification for SHES-NS is the ending position of the agent, this is what is added to the archive and used to obtain the novelty of an individual. Thus using the ending position of the agent encourages the controller to recommend positions to the primitive which the agent has not yet explored. Novelty is applied to SHES similarly to the NSRA-ES method (section 2.3.2.2) [Conti et al., 2017], where there is a weighting w which determines the influence of objective-based search as opposed to novelty search and if the agent's performance stagnates for more than n generations then w is decreased. Our method differs from NSRA-ES in two ways, firstly we set a minimum w value of 0.5 and instead of increasing w when the agents performance improves we immediately set it to 1. Both of these are because it was observed that without these changes the agent would often degenerate into states where novelty was the only objective ($w = 0$), and it never escaped from this state, these changes help alleviate this problem. There is also the question of what measure to use to determine when the fitness is stagnating. The logical options are to use the maximum or mean fitness of the individuals in a generation or the fitness of the main policy. SHES uses the mean fitness of the individuals in a generation although an argument could be made for either of the other options and prior testing showed little difference.

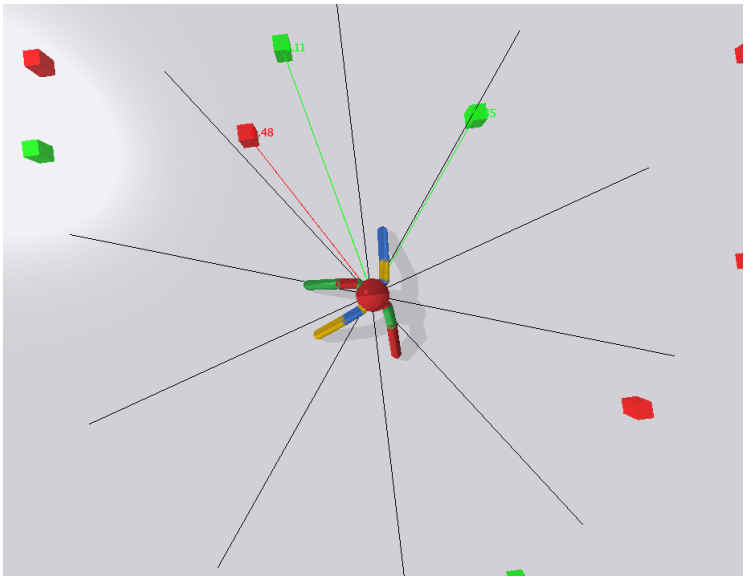


Figure 11: Sensor example in the *Ant Gather* environment. Two black lines enclose an area correlated with a position in the input vector. The closest object of interest between the black lines is given an intensity based on its distance to the agent and is added to its input vector position. If no objects are between two lines or within sensor range the value of the input vector at the related position is 0.

3.6 One Hot Controller

Using a *one-hot* inspired encoding for the output of the controller helps simplify its problem-space via matching the output of the ANN controller to part of its input. To explain the one-hot inspired encoding consider the output of the controller network (μ^c) is a vector $o \in \mathcal{R}^d$ such that each element of o corresponds to one of d equally spaced angles around the agent, this can be seen as the black lines in figure 11. In the same style of one-hot encoding popularly used as the output of image classifying convolutional neural networks [Potdar et al., 2017] the element with the highest value o_h is selected and used to determine the point that the controller will recommend. To use figure 11 as an example, d would be 10 and each element in the output vector o would correspond to one of the black lines. If each line is k units long, given the highest output $o_h \in [-1, 1]$ the controller recommends the point $o_h * k$ units along the relevant line, where k is a hyper-parameter representing the maximum distance from the agent that the controller can recommend.

This isn't a vital part of SHES, but it is especially relevant to one of the environments tested and can be used in agent-environment sensory interactions, as such it is treated as an extension and SHES does not normally use this encoding. This ANN

architecture choice only works well for environments with sensors, since the input to the ANN is similar to its output. For example, if the agent observed a vector representing absolute positions of objects (instead of their sensor readings) this style of ANN architecture would not perform well. Only one of the three environments tested uses agent sensors and as such the one-hot encoding is only used for this environment. When one-hot encoding is not used the controller simply outputs a vector representing the relative (x, y) position from the agent's current position that the agent should move towards.

3.7 Mutation Policy

Since S-ES only optimizes a single policy the choice of what to perturb and when to perturb it is trivial, however, given SHES' policy hierarchy there are numerous choices of how to perturb the policies. One can view the different options of how to perturb the policy hierarchy as either a many-to-many, one-to-many or many-to-one relationship between the number of controllers and primitives that are perturbed.

The many-to-many scenario occurs when each time a controller is perturbed a primitive is perturbed alongside it. One can observe this in algorithm 3 on line 5 where F evaluates a perturbed controller $\theta_t^c + \epsilon_i^c * \sigma$ and a perturbed primitive $\theta_t^p + \epsilon_i^p * \sigma$. This is similar to, but crucially not exactly the same as, concatenating the controller and primitives parameters and treating it as a single policy in regular S-ES, if this was exactly the case then it would not be possible to perform feudal RL's reward hiding, since there could only be one reward.

The major benefit of this approach is that it decreases the wall-clock time and increases the sample efficiency since both the primitive and controller are able to learn at the same time. The many-to-many style does introduce a major disadvantage: a potentially high performing controller could suffer from a badly perturbed primitive. The combination of a high performing controller and low performing primitive would lead to a low reward for the controller, consequently, the controllers would be informed to move away from (in parameter space) this intelligent controller that appeared to perform poorly because of its assigned primitive. Nonetheless the primitive does still benefit from this outcome as it would also receive a low reward, allowing it to inform the optimization step of its true poor performance. In later generations, the primitive becomes more stable and thus has less of an impact, but this effect can be quite prevalent in early generations.

A related, but more subtle issue the many-to-many scheme introduces is that controllers are being unfairly compared to one another. During the optimization step controllers are ranked by their fitnesses, however, a controller's fitness is heavily influenced by the performance of its primitive thus controllers are being unfairly

ranked since each controller has a different primitive variant, therefore, controllers are not being compared to each other on equal grounds. Both of these problems can be mitigated, but not entirely solved, by increasing the number of times the controller and primitive are perturbed within a generation, which decreases the influence of a single controller with a badly performing primitive.

The other two options are a one-to-many or many-to-one approach. This would occur when only perturbing the primitive while using the main controller and only perturbing the controller while using the main primitive. Thus during each step, we optimize either the controller or the primitive. One could apply this scheme to algorithm 3 by changing line five to either:

$$f_i^c, f_i^p = F(\theta_t^c + \epsilon_i^c * \sigma, \theta_t^p)$$

or

$$f_i^c, f_i^p = F(\theta_t^c, \theta_t^p + \epsilon_i^p * \sigma)$$

This shows that either the controller is being perturbed (top) or the primitive is being perturbed (bottom), but never both at the same time.

The biggest issue with this approach is that it would require two generations in order to optimize both the primitive and controller, as one would have to alternate between each approach in order to optimize both. This may not be an issue since the approach may speed up learning, but it is at the very least inefficient. However, it brings with it the advantage of a more fair learning environment for the controller as it lacks the conceptual drawbacks of the many-to-many method described in the previous paragraph since all controllers are optimized using the same primitive and thus are on a level playing field. In practice SHES uses the many-to-many approach as the stability benefits of the many-to-one/one-to-many approach did not outweigh its need for more samples and the performance of the many-to-many approach stabilizes when creating enough perturbations of the controller and primitive in a generation.

3.8 Noise Sampling

S-ES uses antithetic sampling in order to reduce the variance of the algorithm. This is commonly called mirrored sampling in the ES literature [Ren et al., 2019, Brockhoff et al., 2010] since for any perturbation S-ES samples both $+\epsilon$ and $-\epsilon$, for a given Gaussian noise vector ϵ. SHES needs to adapt this to a hierarchical context with two policies instead of one. Since the controller and primitive both sample their own noise vectors (ϵ_c, ϵ_p) an obvious way to perform antithetic sampling in SHES is to evaluate the pair of negatively perturbed policies $(-\epsilon_c, -\epsilon_p)$ and positively

perturbed policies $(+\epsilon_c, +\epsilon_p)$, although this leaves out two potential combinations when combining the positive perturbations with the negative perturbations, which in theory should further reduce the variance. In spite of this, during informal experiments, we saw very little increase in task performance when using all four possible combinations of perturbations. To add this to algorithm 3 one would repeat line five, four times, each time using different signs for ϵ_i^c and ϵ_i^p.

Interestingly using four perturbations did lead to a minor speed increase because of how it allows one to simplify the final matrix dot product when approximating the gradient. To demonstrate this consider a simplified version of the gradient calculation: $N \circ F$ where N is a vector of noise vectors (an $n \times k$-matrix) and F is a vector for fitnesses (a $k \times 1$-matrix). Since antithetic sampling is used N contains multiple pairs of the same noise vector $\pm\epsilon$, as such one can simplify the dot product by summing the fitness values associated with the same noise vectors. For example, given noise vectors $+\epsilon$ and $-\epsilon$ associated with fitnesses f_+ f_-, one can remove $-\epsilon$ from N and replace the f_+ with $f_p - f_n$ in F, thus reducing the first dimension of F by 1 and second dimension of N by 1, but the resulting dot product will remain the same. Thus when perturbing four times instead of two one can simplify the dot product even further, since there are four fitness values using the same noise vector. For large dot products this greatly reduces the complexity and time to compute by reducing the k dimension of each matrix by a factor of four, hence SHES performs four perturbations instead of two.

3.9 Speedup

One of the main benefits of S-ES is its speed and scalability, and it is important that SHES maintains these benefits in order for the algorithm to be a beneficial addition to the space of HRL algorithms. Salimans et al. claim that their implementation of S-ES achieves a *linear speedup*. This means that the number of cores used will decrease the run-time of the program by a factor of the total number of cores. S-ES as a whole is embarrassingly parallel (meaning it can be easily separated into subtasks), however, there are certain parts of the algorithm that must be performed serially, namely ranking and optimization. Thus, due to Amdahl's law [Gustafson, 1988], one would not expect a perfectly linear speedup. It is expected that when testing the speedup of S-ES (implemented for this work) we will observe a sub-linear speedup with a linear trend, meaning that increasing the core count does increase the speed, but it is unlikely to be a one-to-one relationship.

As mentioned previously SHES should have the same speedup properties as S-ES, since the only significant difference, in terms of overhead, is communication. However, the difference in communication is minimal, since S-ES communicates an extra

three numbers (one 32 bit float and two integers) for each evaluation in order to represent the performance of two policies instead of one: the fitness of the primitive, the noise seed for the primitive perturbation and the number of samples used by the primitive. Given the small amount of extra data needing to be sent between nodes in a cluster, we do not expect the extra communication of SHES to significantly impact its speedup.

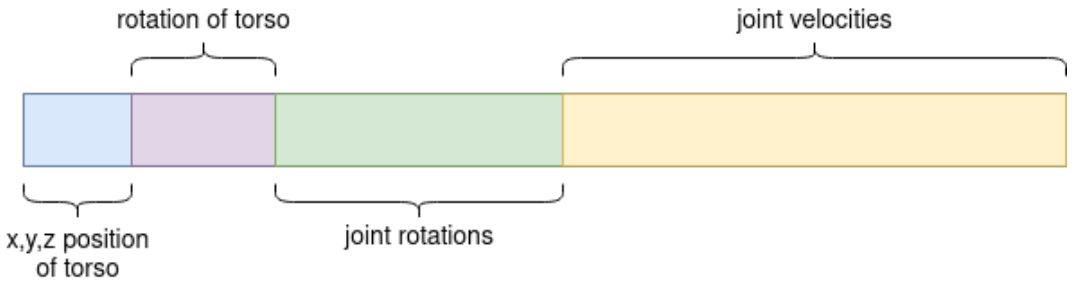


Figure 12: Visualization of the observations that are available from the ant robot. A total of 29 observations are available, starting with the torso position and rotation and ending with the rotation and velocity of the other joints of the ant.

4 Experiments and Results

This chapter details the experiments that have been run to show the task performance of SHES compared to other algorithms and shows the results of these experiments.

4.1 Environments

This section describes the environments used to obtain the task performance of SHES. These four environments (ant gather, ant maze, ant push and ant fall) are especially suited for hierarchical learning since they require two distinct and easily separable skills, namely: locomotion and navigation. Another reason to use the chosen environments is that they have already been used to evaluate previous HRL methods, thus providing a benchmark to gauge the relative performance of SHES. The environments are implemented in the MuJoCo physics engine [Todorov et al., 2012] this is done for ease of comparison to previous methods which used the same environments, the only difference between the environments used in this work and previous works is that these environments[2] are written in the Julia language [Bezanson et al., 2017] while the previous works used the python implementation of the environments.

4.1.1 Quadruped robot

The robot used in this experiment, which can be seen in figure 13, is a quadruped robot that will be referred to as an ant since this is how it is often referred to in the literature. The ant is a relatively simple robot having four legs connected to a torso in the middle, with each of the four legs having two actuation points. This robot

[2] `https://github.com/sash-a/HrlMuJoCoEnvs.jl`

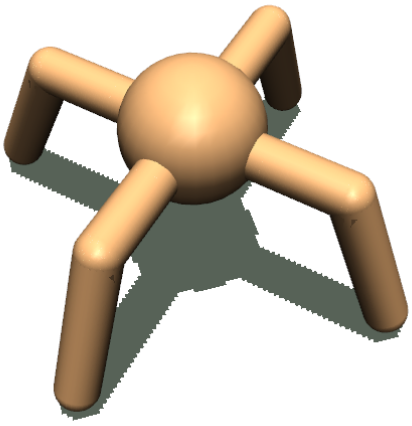


Figure 13: The quadruped or ant robot is used in all of the environments in this work. It has four legs connected to a torso, with each of the four legs having two actuation points.

is a good middle ground of the 3D robots available from the MuJoCo suite, falling between the more complex humanoid and the less complex swimmer. The humanoid was used in HRL related work by Peng et al. and Heess et al., the swimmer was used in work by Florensa et al., however, the ant is by far the most popular robot for HRL locomotion style tasks being used in at least five other significant works in the field [Florensa et al., 2017, Nachum et al., 2019, 2018, Vezhnevets et al., 2017, Heess et al., 2016].

As can be seen in figure 12 the ant has a total of 29 observations which are supplied to the ANN as a vector. The first three elements are the x, y and z positions of the ant's torso, the following four elements are the torso rotation as a quaternion, the following 8 elements are joint angles and the final 14 elements are joint velocities. To move the ant, it takes an action vector of length eight with each of the values corresponding to a torque to be applied to either a hip or knee joint along the four legs of the ant. The hip joints of the ant exist at the join between each leg and the torso and the knee joints are present at the bend in each of the legs as can be seen in figure 13.

The ant used in this work is modified from the standard ant found in MuJoCo, this was done in order to match the evaluations done by HIRO so the results could be fairly compared. The gear range for all of the joints has been reduced from $[-150, 150]$ to $[-30, 30]$ this makes the ant easier to control as each input has less impact on the amount the joint moves, thus making movements less *jerky* and more

smooth, contributing to a more natural and functional gait.

4.1.2 Ant Flagrun

The flagrun environment is the most simple of all the environments used and does not serve as a testbed for the hierarchical side of this work, but rather as an environment for pretraining the primitive for harder tasks. However, it does come with a limitation, a primitive that has been pretrained on flagrun cannot be provided with the full set of observations from the environment it is transferred to since the observations are not available in flagrun. Thus, when using a pretrained ANN it only receives a slice of the observations from its new environment which only correspond to the ant's 29 observations and a target position. This makes the learning process easier for the primitive since it does not need to concern itself with other observations, however, it also makes the approach less generalisable to other environments since observations must be carefully sliced to provide the pretrained primitive with the correct observations in the correct order. This environment differs from the hierarchical environments in that it has no boundaries and is completely open, giving the ant more space to move around and not impeding any possible targets that are suggested.

Ant flagrun requires that the ant reach a target (flag) every c time steps, which is uniformly randomly generated by selecting a point at least 1 and at most d units away from the ants current position. The reward for this environment is the same as the controller provides to the primitive in SHES:

$$R_t = 1 - d_t/d_c + (1 \; if \; d_t < L \; else \; 0)$$

where d_t is the current distance to the target at time step t, d_c is the distance to the target at the time it was recommended and L is a constant threshold (1 for the flagrun environment). This reward translates into 1 - the normalised distance to the target plus 1 if the agent is close enough to the target, which encourages the agent to reach the target as soon as possible and remain close to it. The environment terminates at 500 steps or when the ant falls over.

There are existing benchmarks for similar versions of this environment, however, they mostly use a humanoid robot and a different reward, thus it is difficult to compare existing results to this environment and is beyond the scope of this work.

4.1.3 Ant Gather

In the ant gather environment, depicted in figure 14 the goal of the ant is to collect as many food items (green) as possible and avoid the poison items (red). For each

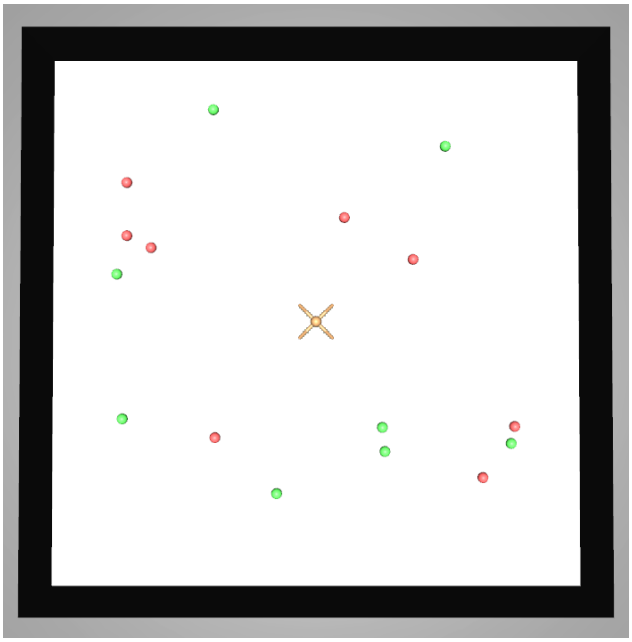


Figure 14: The ant gather environment where the green spheres represent food items and the red spheres represent poison items.

food item the agent comes into contact with it receives a +1 reward and for each poison item it comes into contact with it receives a -1 reward. This is a rather sparse reward and the ant has to initially explore the surrounding area to obtain any reward.

At the start of an episode the agent is placed in the middle of an enclosed environment of size 10x10 units. The food and poison items are randomly placed around the agent with a minimum distance from the agent being 2 units, thus not allowing the agent to get a *free* reward by placing the food or poison on top of the agent.

To observe the position of the food and poison items, the agent is given two proximity sensors one of which returns depth values for food items and the other for poison items. An example of this sensor can be seen in figure 11. A sensor has ten *bins*, thus ten depth readings, for each of the two sensors, are appended to the ant's joint observations (see figure 12). If a food item is within one of the bins of the sensor and within the sensor range then the corresponding element in the observation vector is given a depth value based on the food items distance to the ant:

$$depth = 1 - d/r$$

where d is the distance of the food item to the ant and r is the range of the sensor.

Parameter name	Parameter value
number of apples	8
number of poisons	8
activity range	10
catch range	1
number of bins	10
sensor range	6
sensor span	2π

Table 1: Parameter names and values for the ant gather environment. These are the same values used by Nachum et al. for HIRO [Nachum et al., 2018]. The sensor span corresponds to how much vision the sensors give the agent with 2π giving full 360° vision.

This means that the closer the item is the higher the value, thus the agent needs to learn to associate each *bin* with its corresponding direction. If two of the same item type fall into the same bin then only the closer item's depth is added to the observation vector.

As can be seen in table 1 many variables can contribute to the difficulty of the environment. Likely the most difficult is the catch range, where lower values force the agent to be closer to the item to pick it up. Sensor span can also make the environment challenging, the lower the span the less agent can see, forcing it to turn around to see items in its blind spot.

4.1.4 Ant Maze

In the ant maze environment depicted in figure 15, the agent must learn to walk from one corner of the *u-shaped* maze to the other. The agent spawns in the same corner each time, however during training target positions are randomly sampled from points in the maze, thus teaching the ant to reach general positions in the maze. During testing, the agent must reach the end position in the opposite corner of the maze. More formally the agent spawns at $(0, 0)$ and during training must reach a point uniformly sampled from $g_x \sim (-4, 20)$ and $g_y \sim (-4, 20)$ such that the point is reachable within the maze. However during test time the agent is only evaluated by its ability to reach $(0, 16)$, the *red dot* in figure 15.

The agent receives no special observations related to the walls in this environment, thus it needs to implicitly learn the layout of the maze. Other than the normal ant observations shown in figure 12 the ant maze environment provides the agent with the position of the target and the current time step. The reward provided during training time is the negative L2 distance to the target: $-\sqrt{(g_x - x)^2 + (g_y - y)^2}$

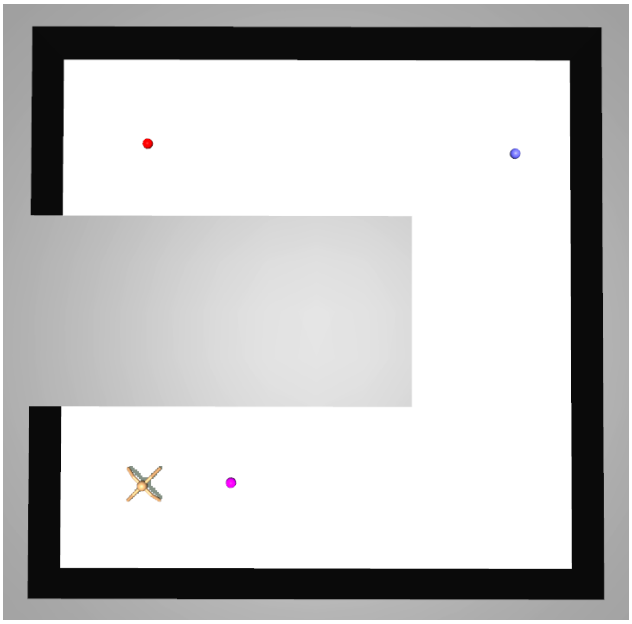


Figure 15: A top-down view of the ant maze environment. The ant starts in the bottom left corner and the evaluation target is the red dot in the top left corner, however, this is separated by a wall in the middle. The purple dot represents the controller's recommended position and the blue dot is the training target position.

where g_x and g_y is the goal/target position. Whereas during test time the agent receives a test reward of 1 if it is within an $L2$ distance of five on the ultimate step of the episode, otherwise it is given a test reward of zero. The episode only ends after 500 steps, even if the ant falls over the episode continues until the 500^{th} step.

This environment is challenging because the ant will have to find its way out of the obvious local minima of walking into the wall when the target position is on the other side of the maze. Couple this with the fact that the target position is randomly generated each episode and it becomes difficult to learn the general layout of the maze and consistently reach the target.

4.1.5 Ant Push

This environment is very similar to the ant maze environment (section 4.1.4) however it requires that the ant learn to push a block out of the way before it can reach the goal. As can be seen in figure 16 the ant must reach the red dot directly in front of it, however, it is blocked by the large red square. Thus the ant must learn to first move to its left, then push the red block, which is the only moveable block, to the right to access the target location at the top of the environment. The agent

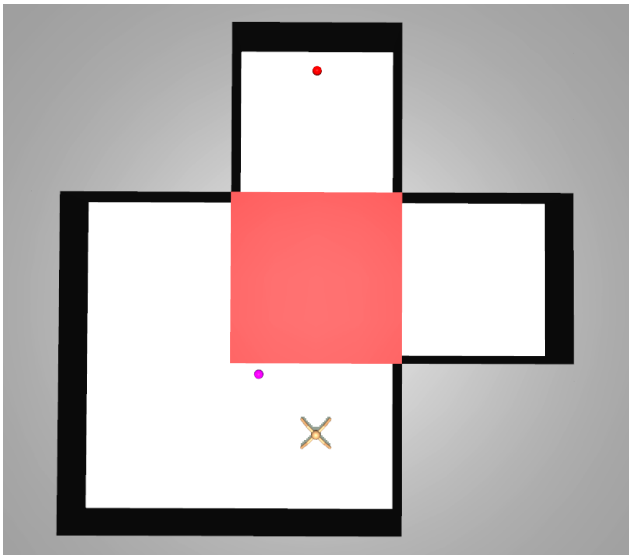


Figure 16: A top-down view of the ant push environment. The ant must reach the red dot at the top of the maze, but is blocked by the moveable red square, thus the ant must learn to first push the square to the right so it can access the area with the red dot. The pink dot near the ant is the goal g_t.

is initialised at $(0, 0)$ and must learn to reach $(0, 19)$, however a moveable block is placed at $(0, 8)$. Unlike ant maze the evaluation and training targets are the same, consequently, the ant is always required to reach $(0, 19)$.

The agent receives an extra observation of the target position $(0, 19)$, but it does not have any sensors so it must implicitly learn the layout of the maze. The ant receives extra observations for the red block in the form of its xy position and its velocity. The same reward is used during train time as in ant maze: $-\sqrt{(g_x - x)^2 + (g_y - y)^2}$ where g_x and g_y is the goal/target position. Similar to ant maze, the ant is considered to have successfully solved the maze if it is within five units of $(0, 19)$ on the final step of the episode and will be given a test reward of 1 otherwise it receives 0. Even if the ant falls this environment does not end, it only ends on the 500th step.

This environment is challenging for two reasons: first, the agent needs to be able to explore around the obvious local minima of simply walking forward, thus getting close to the target, but not as close as possible. Second, the environment requires that the agent learns to interact with the moveable block. Thus the agent needs to learn to move left to avoid the local minima and then learn to push the block in the correct direction to reach its goal.

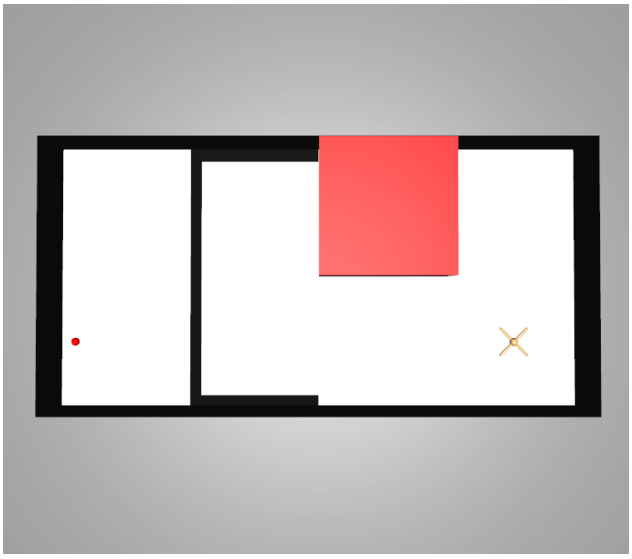


Figure 17: A top-down view of the ant fall environment. The agent must reach the red dot on the other side of the chasm. This is achieved by pushing the red block into the chasm and walking over it.

4.1.6 Ant Fall

This environment is very similar to the ant push environment, however, instead of having to push a block to the right to reach the goal, the agent must push the block left. As can be seen in figure 17 the agent must learn to push the block into the chasm to cross it so that it can reach its goal (the red dot in figure 17). Thus the agent must first move upwards to be behind the block and push the block into the chasm allowing it to cross over the chasm and reach the goal on the other side.

The agent receives the same observations as in the ant push environment except that the target position it must reach is $(0, 27, 4.5)$, as it needs an elevation for the target given that there are different reachable areas (on the z-axis) in this environment. Training reward, test reward and termination conditions are also the same as ant push.

While this environment is very similar to ant push, it is a notably more challenging environment as can be seen by the results of HIRO [Nachum et al., 2018] (tables 5 and 6). This is likely because the block needs to be pushed much further for the agent to be able to reach its goal thus making it pivotal for the agent to learn to interact with the block without falling over. Couple this with the fact that the agent also has to travel further in this environment making it important that it learns a fast-moving gait and one can see why this is a more challenging environment than

Hyperparameter	Value 1	Value 2	Value 3
Target distance	2	**4**	8
Interval	10	**25**	100
Controller distance	2	**4**	8
Learning rate	0.001	**0.01**	0.1
Sigma	0.002	**0.02**	0.2
Episodes	3	**5**	10
Policy per generation	**240**	512	1000

Table 2: Values for all the hyperparameter tuning experiments. Each value is tested ten times in all four environments. The bolded values are the base values used when not testing that parameter.

ant push.

4.2 Experiments

4.2.1 Hyperparameter tuning

To provide useful recommendations for users of SHES, hyperparameter searches are performed for the most important parameters, which can be seen in table 2. Each parameter test is repeated ten times on each of the four environments, thus 40 times total. This allows for a clear picture of how each parameter behaves across all of the environments. To tune the hyperparameters a simple grid search is performed, which exhaustively tests each parameter value against a base set of parameter values. Table 2 details all the hyperparameters tuned and the values which were used to tune them. All tuning runs are run for 10 hours on a single node or until 3000 generations, thus some curves may end early as certain parameters allow for faster run-times, which allows the experiment to reach the 3000 generation limit before the 10-hour deadline. Table 7 shows the chosen parameters after the grid search was completed. The hyperparameter tuning experiments are also used to test the claim made in the introduction: SHES will be robust to hyperparameter changes given that it uses S-ES, which is contrary to current HRL methods.

4.2.2 Determining SHES performance

The performance of SHES will be judged from three perspectives: test reward, train reward, sample efficiency and wall clock time. Test reward differs for the four environments, for ant maze, ant push and ant fall it is simply one if the agent is within five units of the goal on the ultimate step of the episode otherwise zero, this is then repeated ten times (to obtain the mean test reward) and an average is taken

to better understand the consistency of the performance. Ant gather has a different test reward, it is the highest cumulative reward the agent achieved throughout the episode. Nachum et al. refer to this metric as success, while this aptly describes the maze, push and fall environments, test reward is a more descriptive term when also considering the gather environment. Train reward is also used when comparing different variants of SHES as it is a more descriptive reward. Reporting the relative sample efficiency and learning speed of SHES is an important factor in determining its usefulness and thus these values will be reported alongside train and test rewards.

Once the most optimal hyperparameters are obtained SHES will be run ten times on each environment, with different random seeds, to avoid random results influencing the performance. From these results average test reward will be compared to gradient-based methods [Nachum et al., 2018, Vezhnevets et al., 2017, Florensa et al., 2017, Houthooft et al., 2016] results which can be seen in tables 3, 4, 5 and 6. All experiments are run 10 times to show that the performance of SHES is repeatable and to have statistical confidence that one method performs better than another. To gain this statistical confidence, we use a Mann–Whitney U test and report the *P-values* from this test.

Once SHES has been compared to existing HRL algorithms, we will show the performance of the transfer learning and novelty search extensions. These extensions will be compared using the training reward instead of the test reward, which is used when comparing SHES to gradient-based methods. For the transfer learning extension, sample efficiency is difficult to measure as it is questionable whether to include the samples used to train the primitive since the same primitive is used throughout all four environments. As a result, we graph the transfer learning extension ignoring the samples/time taken in the pretraining phase.

To show the scalability of SHES, experiments have been conducted using different numbers of cores from 1 to 600. All experiments are performed on a cluster with each node having an Intel Xeon 24 core CPU at 2.6GHz, 64GB of RAM and nodes are connected by FDR InfiniBand.

4.3 Results

This section displays the results obtained from our experimentation in the form of tables and graphs. Given that our main comparison is to the HIRO method we use results from the HIRO publication [Nachum et al., 2018] and also rerun a subset of their experiments using their official repository[3]. The graphs and tables are followed by an explanation of some of the exact values in the graphs and hypothesis testing to determine if the results are statistically significant.

[3] `https://github.com/tensorflow/models/tree/master/research/efficient-hrl`

To obtain training curves for HIRO we have rerun their ant gather, maze, push and fall experiments, these are labelled *HIRO (ours)* in the graphs and the original results are labelled *HIRO (Nachum et al.)*. We specifically do not plot HIRO training curves against samples in the same graph as SHES training curves, as it is difficult to see the HIRO training curve given its higher sample efficiency, however, we do plot the HIRO training curves against time. HIRO was run on the same hardware as SHES, but a single node with a 24 core Intel Xeon CPU at 2.6GHz and an Nvidia V100[4].

Interestingly the results of our HIRO experiments differ from the results of the original experiments, even though we are reproducing their experiments as described in their official repository. Unfortunately, ant gather was not made as reproducible as ant maze, push and fall thus the poor performance on the environment, out of ten runs only two runs got more than 0.5 test reward. Ant push and fall were made to be more reproducible, however, the test rewards from our experiments are notably lower than those reported by Nachum et al. [Nachum et al., 2018]. Interestingly the ant maze test reward matches the original reported value.

[4] `https://www.nvidia.com/en-us/data-center/v100/`

4.3.1 Hyperparameter Tuning

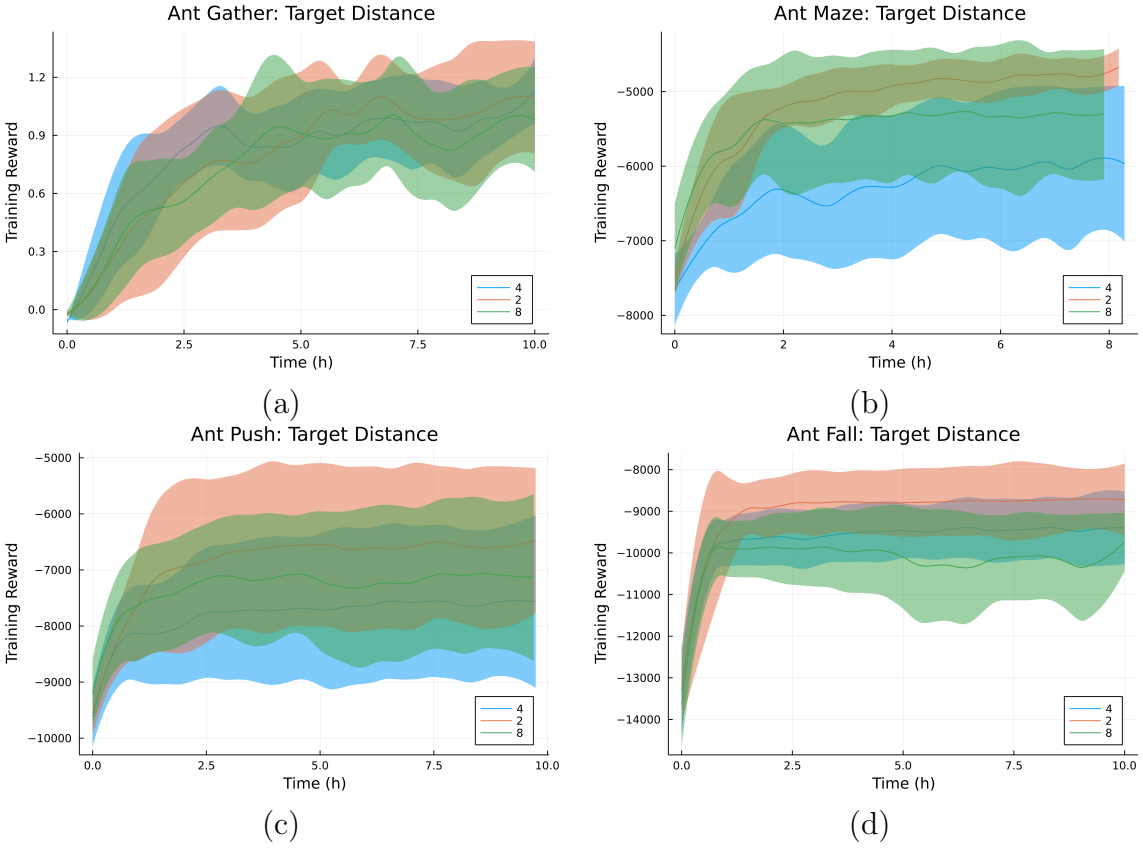


Figure 18: Results of tuning the *target distance* parameter of SHES across all environments. The values used are two four and eight.

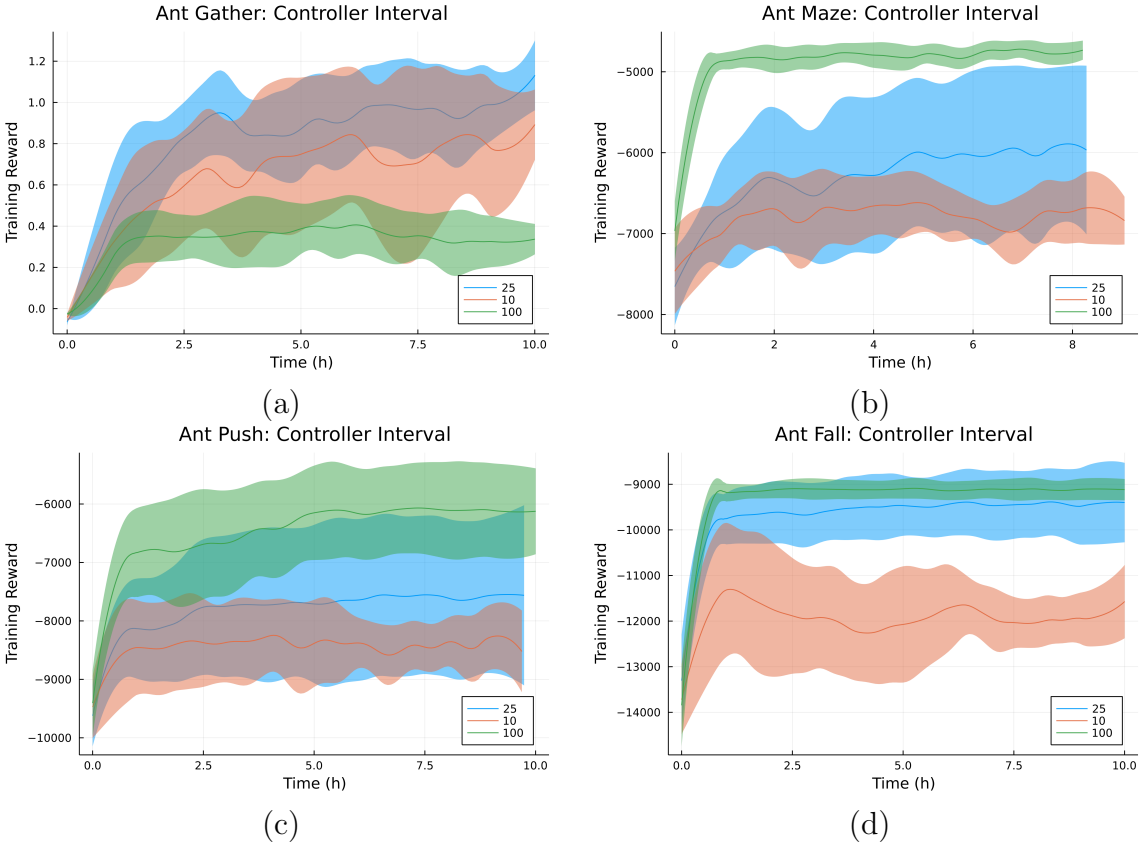


Figure 19: Results of tuning the *controller interval* parameter of SHES across all environments.

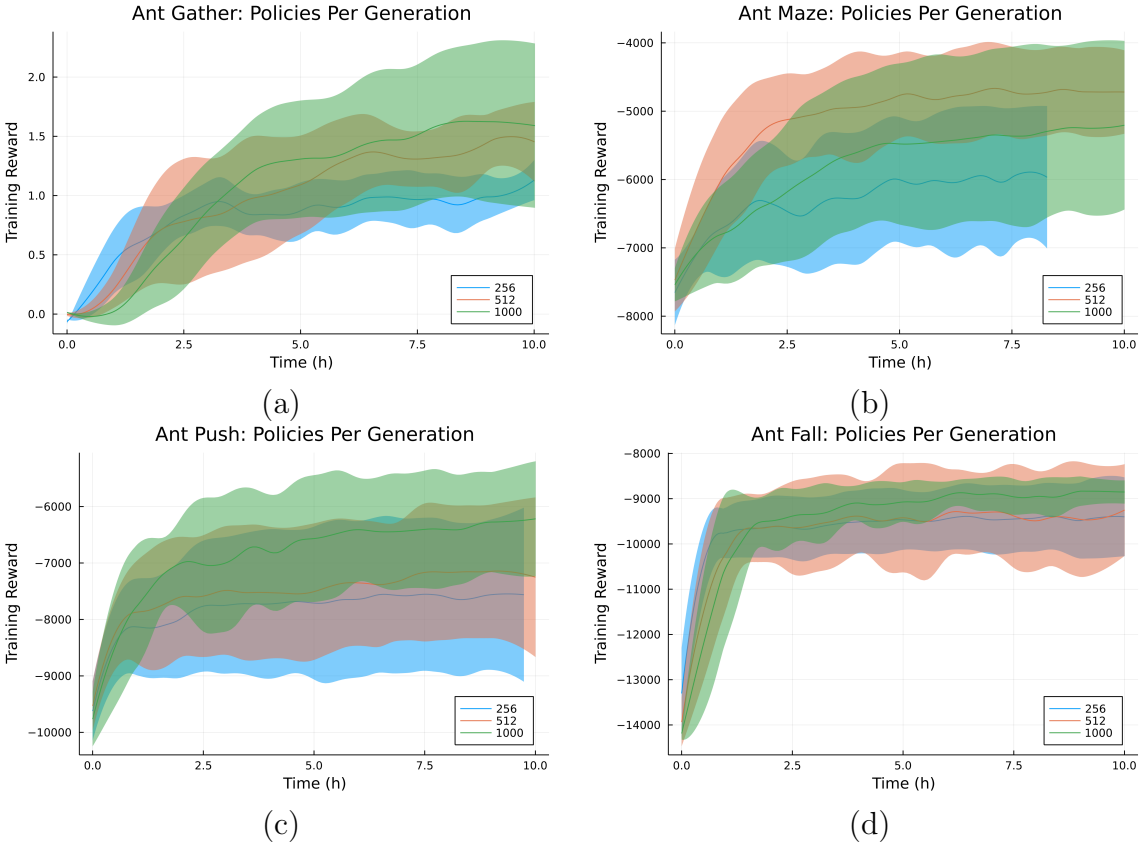


Figure 20: Results of tuning the *policies per generation* parameter of SHES across all environments. This parameter has a high impact on sample efficiency and runtime, thus some runs can finish 3000 generations before the 10-hour time limit.

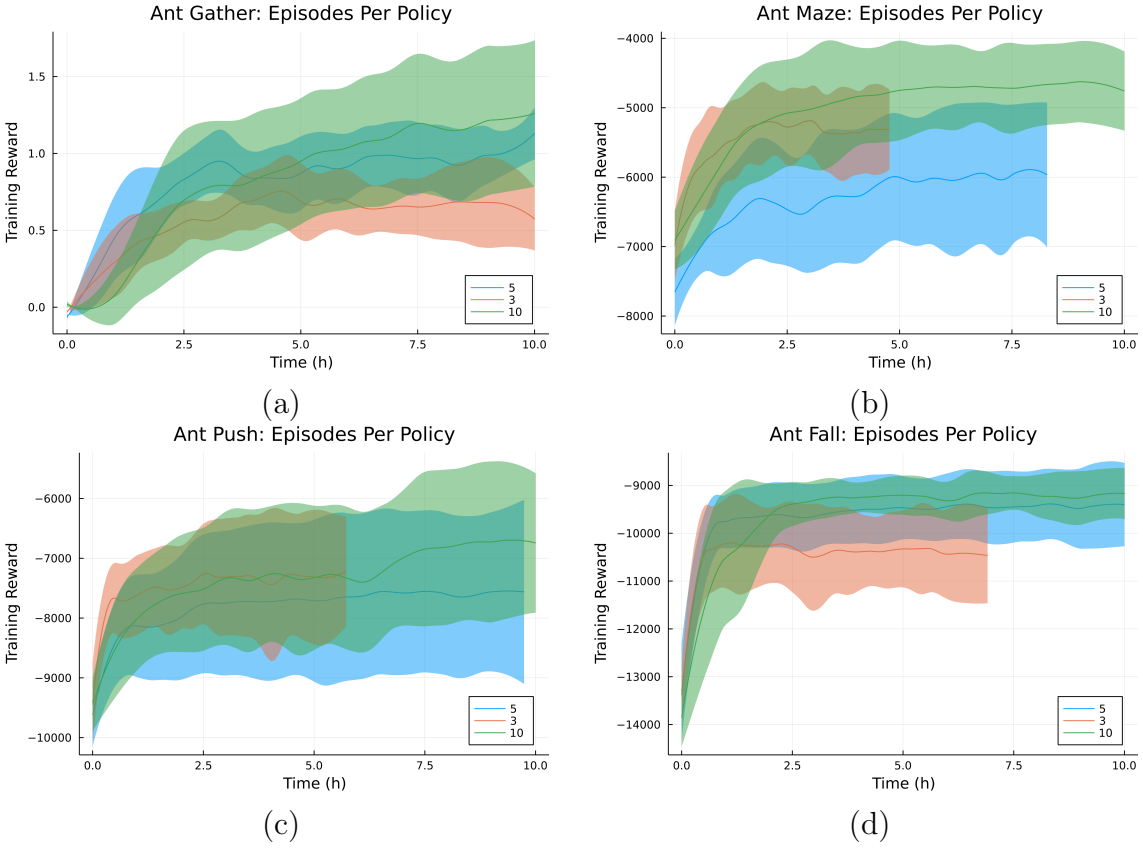


Figure 21: Results of tuning the *episodes per policy* parameter of SHES across all environments. This parameter has a high impact on sample efficiency and run-time, thus some runs can finish 3000 generations before the 10-hour time limit.

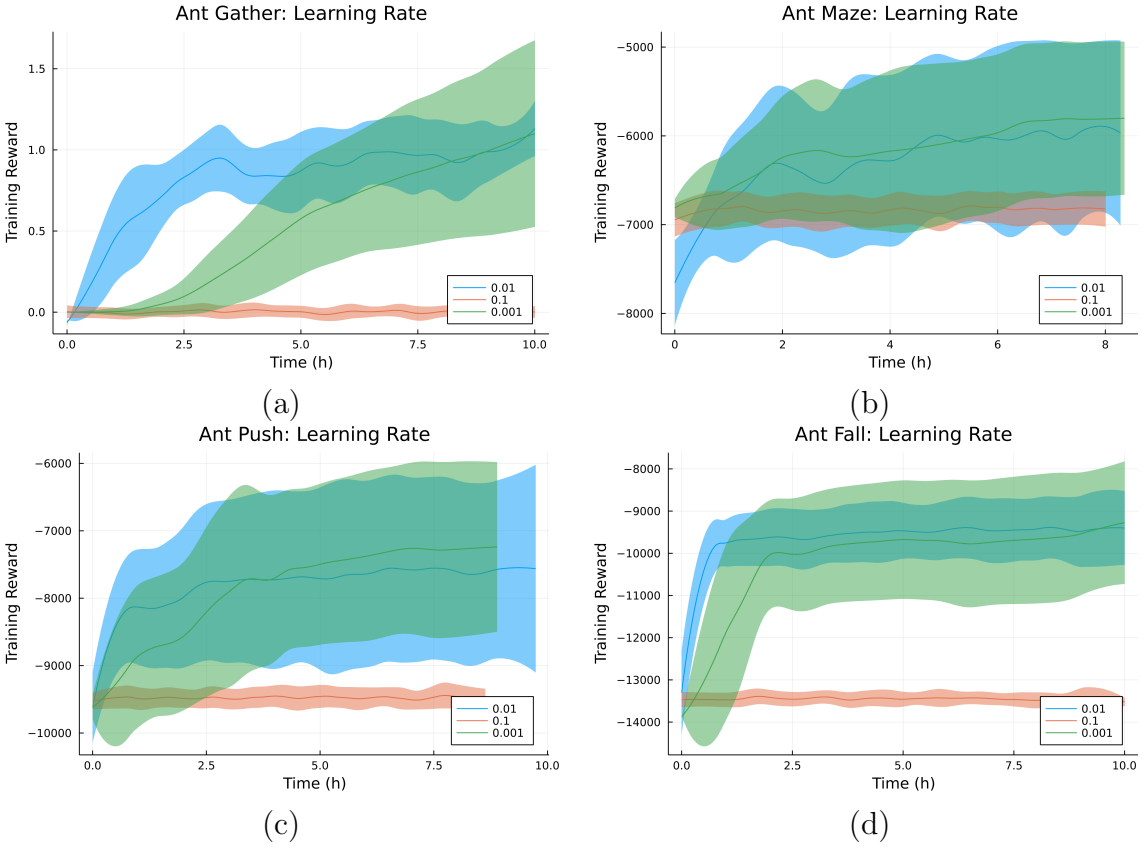


Figure 22: Results of tuning the *learning rate* parameter of SHES across all environments. A value of 0.01 was used by Salimans et al. to obtain competitive results in the MuJoCo and Atari domain.

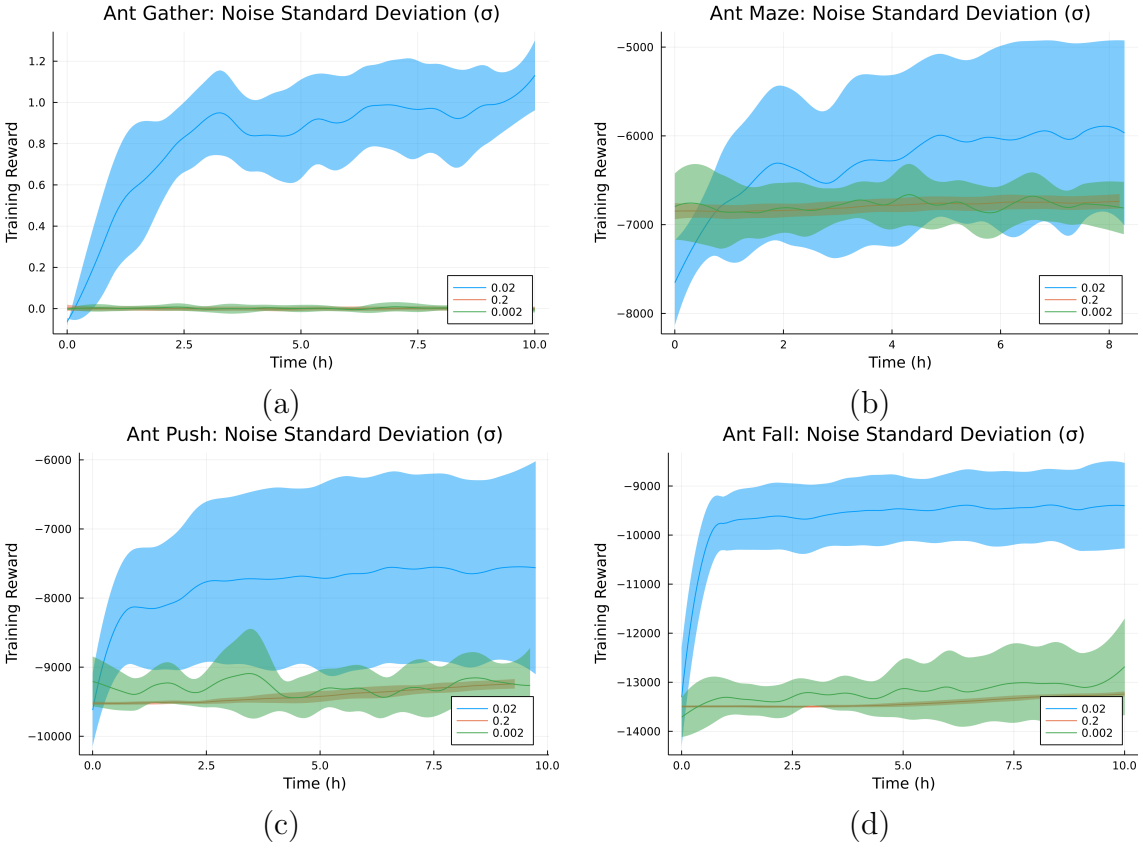


Figure 23: Results of tuning the *sigma* parameter of SHES across all environments. A value of 0.02 was used by Salimans et al. to obtain competitive results in the MuJoCo and Atari domain.

4.3.2 Experiment graphs

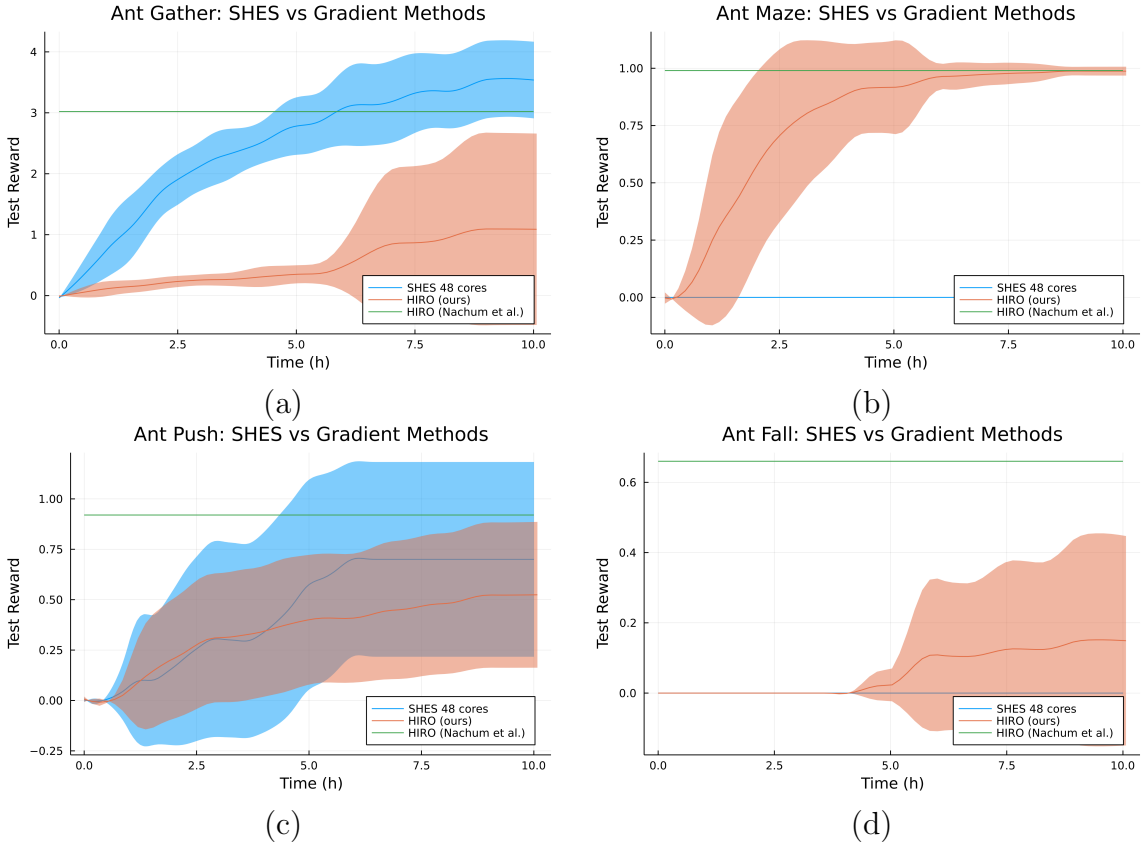


Figure 24: Comparing SHES to gradient-based HRL methods on the ant gather (a), ant maze (b), ant push (c) and ant fall (d) environments. Comparisons use the test reward as the performance metric and measure this performance over time. HIRO (Nachum et al.) maximum test reward (not reward over time) is also plotted.

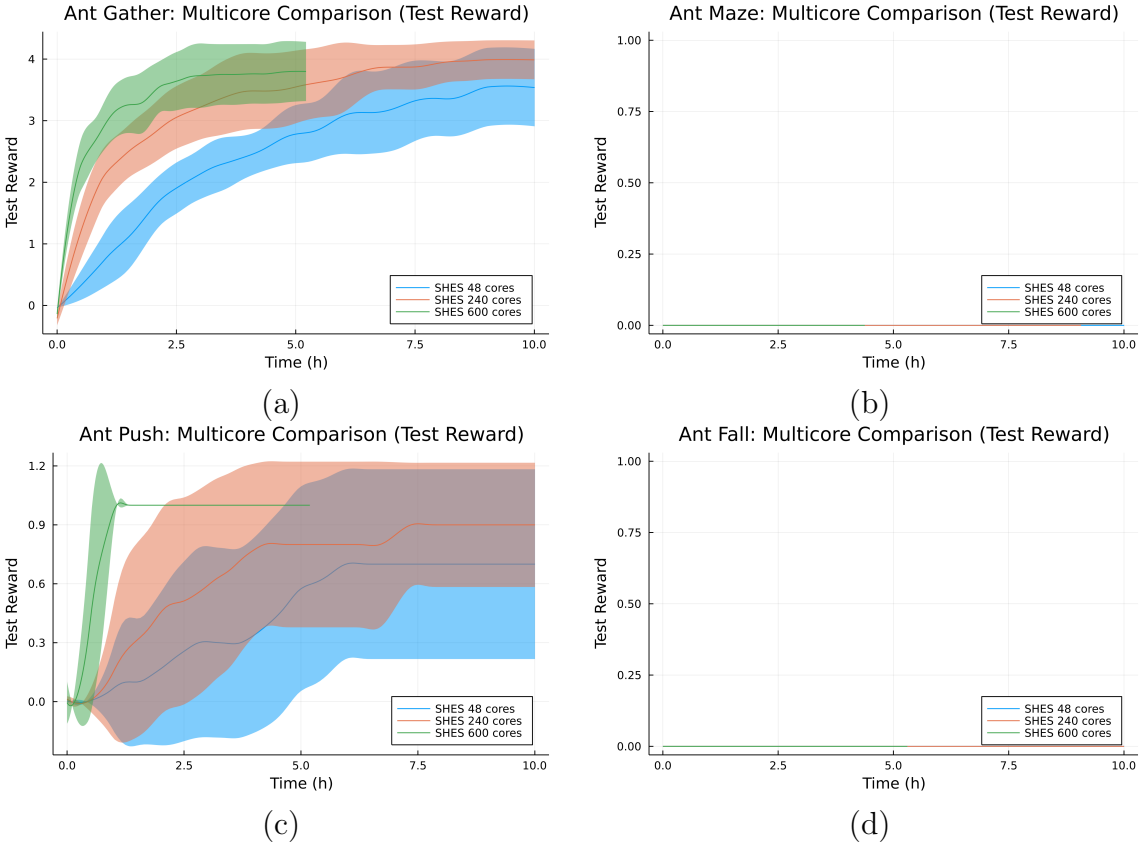


Figure 25: Contrasting SHES' performance when given different amounts of compute, which shows the scalability of SHES to higher CPU core counts. Note the steeper rise of the green curve in all graphs indicating that SHES on 600 cores has a higher learning speed than SHES on 240 or 48 cores. Figures b and d appear flat because this graph is plotted against test reward and SHES was unable to obtain high test reward for these environments, see figure 26 for a more informative view of the scalability of SHES on these environments.

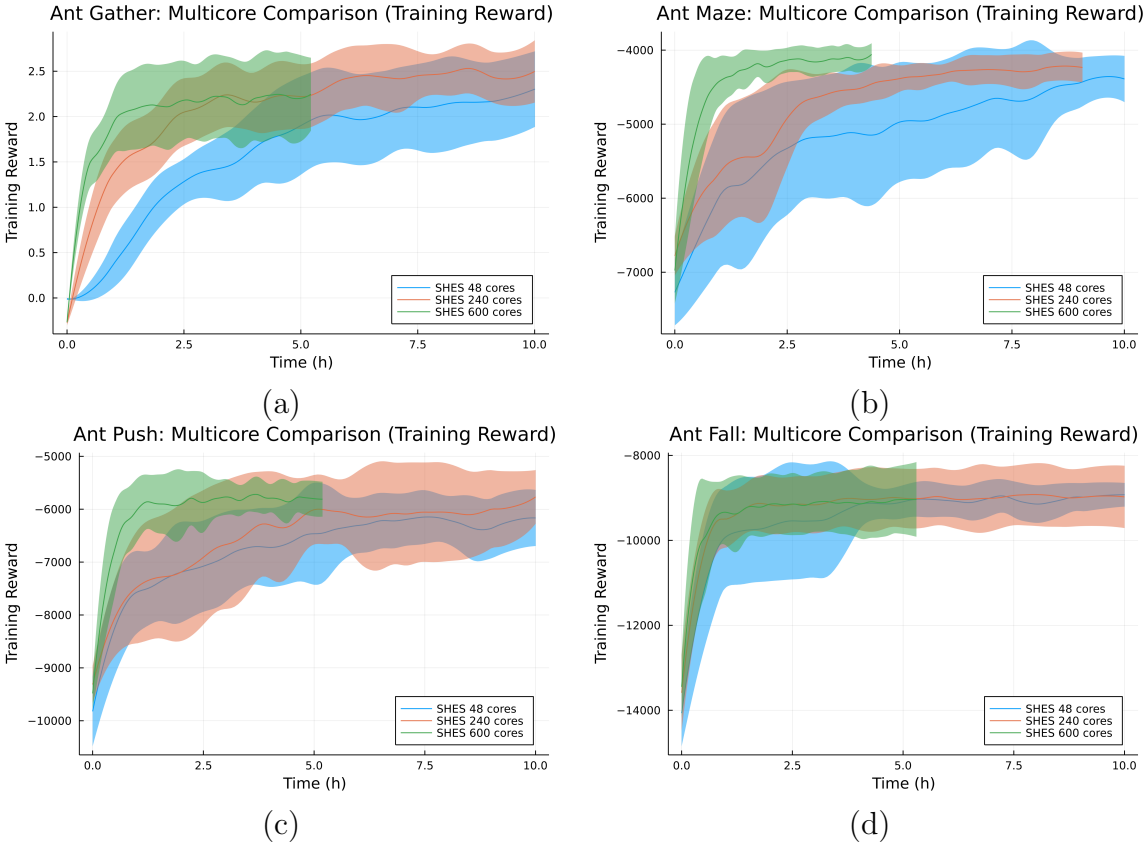


Figure 26: Contrasting SHES' performance when given different amounts of compute, which shows the scalability of SHES to higher CPU core counts. Note the steeper rise of the green curve in all graphs indicating that SHES on 600 cores has a higher learning speed than SHES on 240 or 48 cores. Note, unlike figure 25 these graphs are plotted as the training reward over time, not test reward over time.

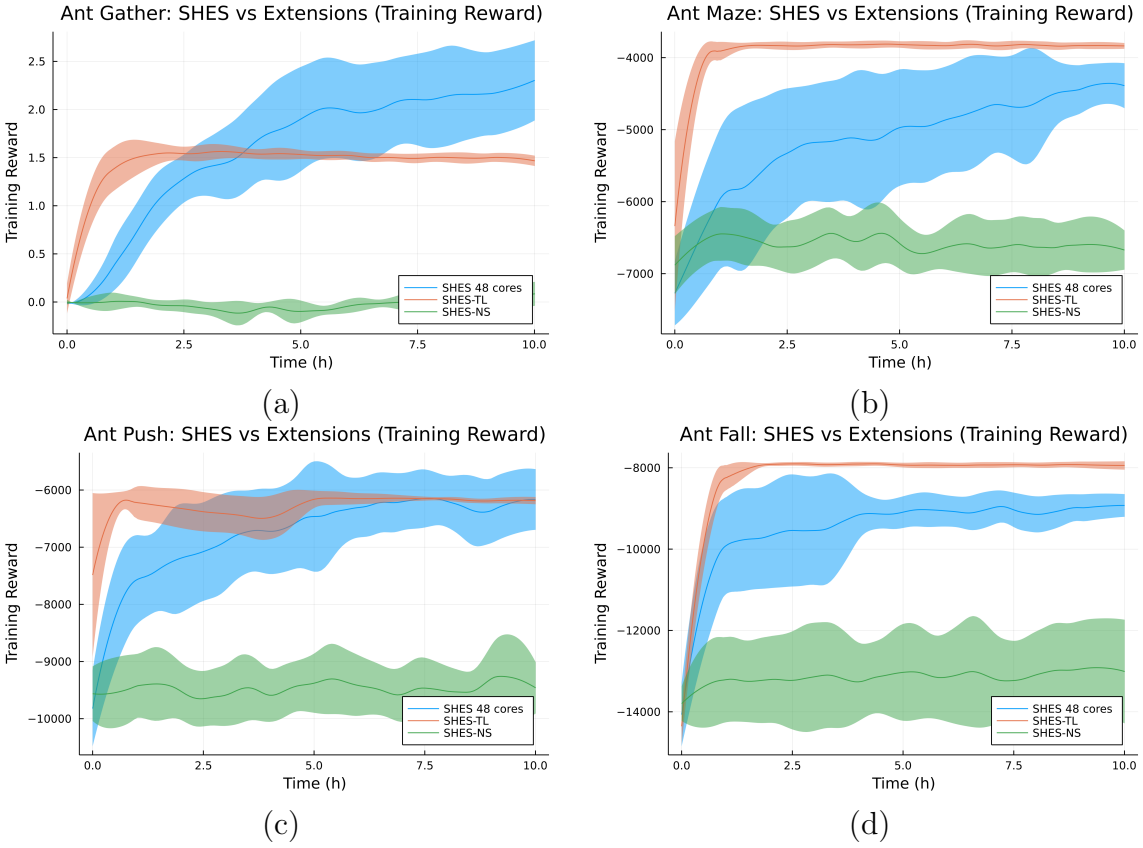


Figure 27: Displays the performance of the transfer learning (SHES-TL) and novelty search (SHES-NS) extensions compared to base SHES. Graphs are plotted as training reward over time. All methods are run on 48 cores.

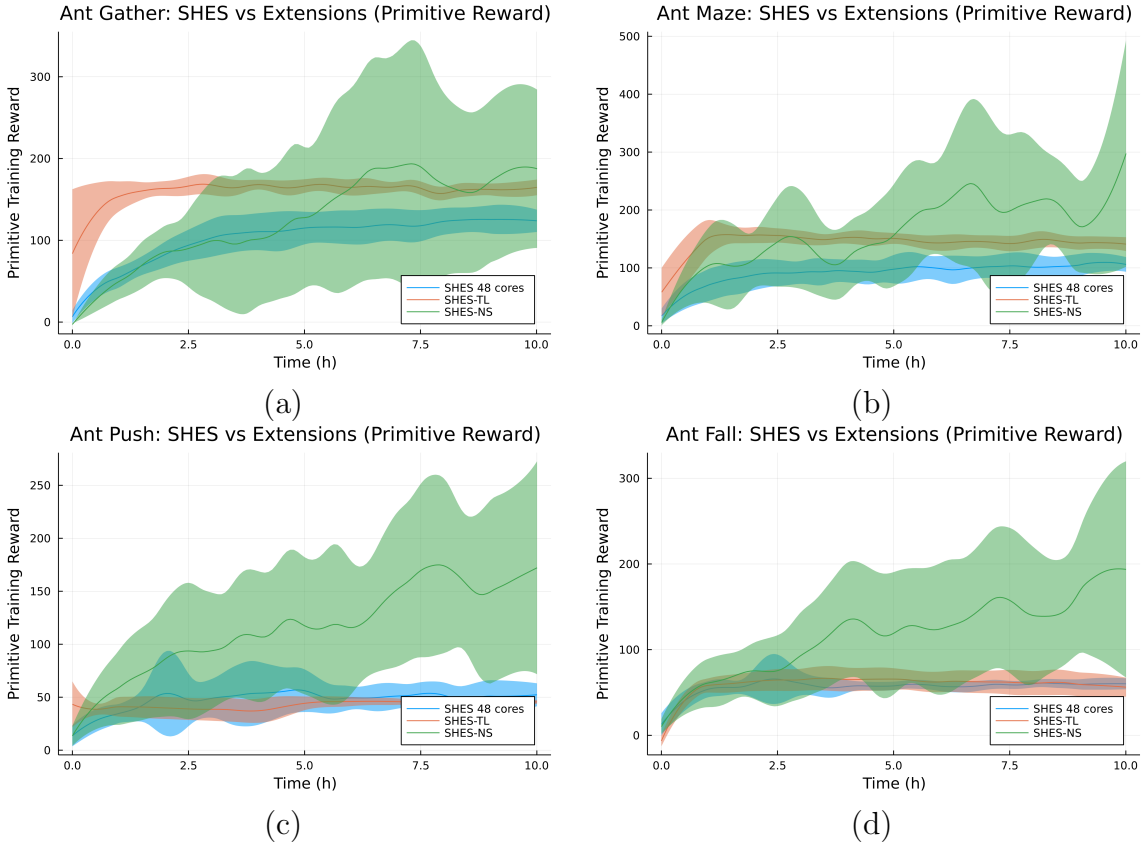


Figure 28: Displays the performance of the transfer learning (SHES-TL) and novelty search (SHES-NS) extensions compared to base SHES. Graphs are plotted as primitive reward over time. All methods are run on 48 cores

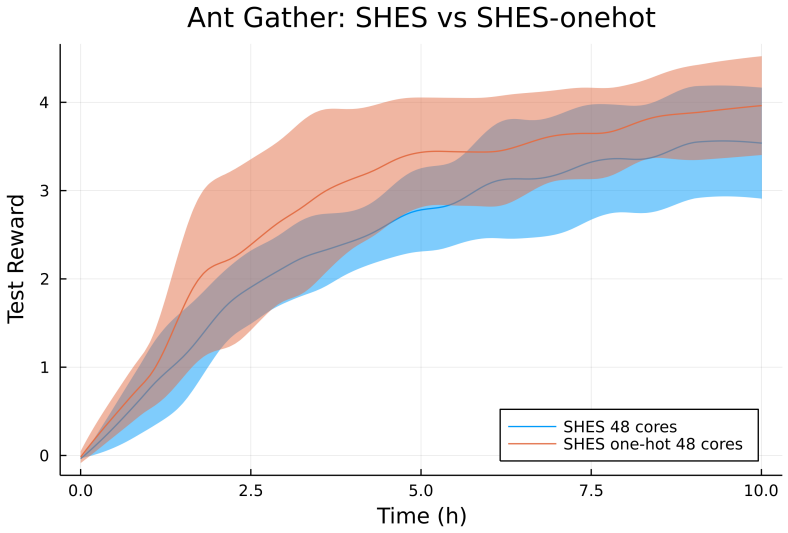


Figure 29: A comparison of SHES-onehot to SHES on the ant gather environment. This extension is only tested on ant gather as it is the only environment in which the robot uses a sensor, and SHES-onehot only applies to environments with a sensor.

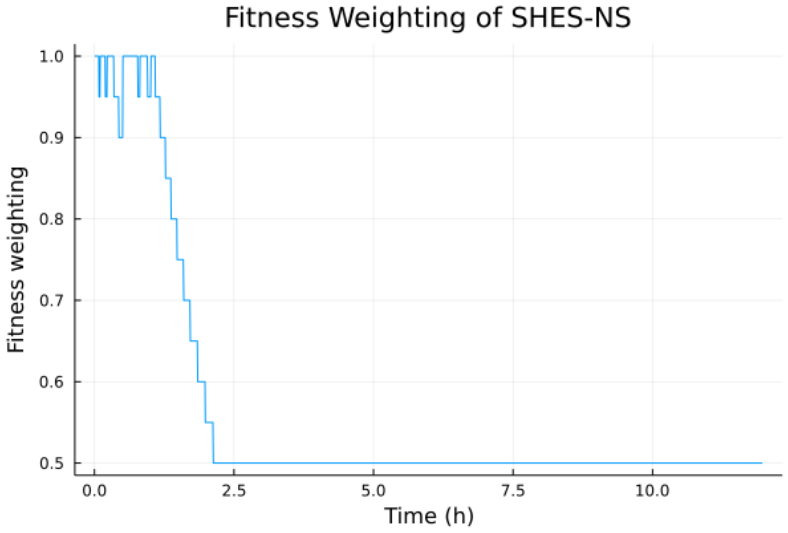


Figure 30: The fitness weighting of SHES-NS on a single run of the ant maze environment (which was selected randomly from all runs). A lower fitness weighting means more influence toward the novelty objective.

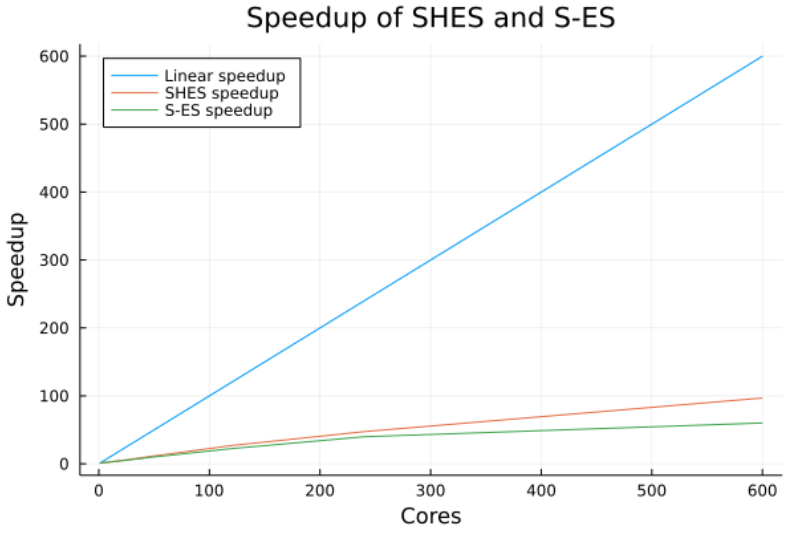


Figure 31: SHES and S-ES speedup versus perfect linear speedup: SHES offers approximately a one-fifth speedup for each core added up to 250 cores and one-sixth speedup up to 600 cores, notably this is better than S-ES.

Method	Test reward	Steps	Time (h)
SHES (24 cores)	2.75 ± 1.08	8.96×10^8	10
SHES (48 cores)	3.7 ± 0.65	1.76×10^9	10
SHES (100 cores)	3.68 ± 0.44	4.29×10^9	10
SHES (240 cores)	3.96 ± 0.36	7.43^9	10
SHES (600)	3.87 ± 0.5	7.43^9	**5.74**
SHES one-hot (48 cores)	$\mathbf{4.18 \pm 0.63}$	1.85×10^9	10
SHES-TL (48 cores)	2.69 ± 0.11	1.8^9	10
SHES-NS (48 cores)	0.98 ± 0.08	1.95^9	10
HIRO (ours)	1.22 ± 1.82	10^7	12
HIRO (Nachum et al.)	3.04 ± 1.49	$\mathbf{10^7}$	*Unknown*
FuN	0.85 ± 1.17	10^7	*Unknown*
SNN4HRL	1.93 ± 0.52	10^7	*Unknown*
VIME	1.42 ± 0.90	10^7	*Unknown*

Table 3: Task performance of SHES, SHES variants and gradient-based methods on **Ant Gather**. Performance is the average of 10 randomly seeded trials with standard error. Performance of all methods that are not SHES is taken from Nachum et al. [2018].

Method	Test reward	Steps	Time (h)
SHES (24 cores)	0.2 ± 0.41	2.63×10^9	10
SHES (48 cores)	0 ± 0	4.4×10^9	10
SHES (100 cores)	0 ± 0	7.5×10^9	10
SHES (240 cores)	0 ± 0	7.5^9	9.3
SHES (600)	0 ± 0	7.5^9	**4.8**
SHES-TL (48 cores)	0 ± 0	4.59^9	10
SHES-NS (48 cores)	0 ± 0	4.24^9	10
HIRO (ours)	0.99 ± 0.02	10^7	12
HIRO (Nachum et al.)	$\mathbf{0.99 \pm 0.1}$	$\mathbf{10^7}$	*Unknown*
FuN	0.16 ± 0.33	10^7	*Unknown*
SNN4HRL	0 ± 0	10^7	*Unknown*
VIME	0 ± 0	10^7	*Unknown*

Table 4: Task performance of SHES, SHES variants and gradient-based methods on **Ant Maze**. Performance is the average of 10 randomly seeded trials with standard error. Performance of all methods that are not SHES is taken from Nachum et al. [2018].

Method	Test reward	Steps	Time (h)
SHES (24 cores)	0.7 ± 0.48	2.12×10^9	10
SHES (48 cores)	0.8 ± 0.42	3.63×10^9	10
SHES (120 cores)	0.8 ± 0.45	6.53×10^9	10
SHES (240 cores)	1 ± 0	7.5^9	10
SHES (600)	1 ± 0	7.5^9	**5.54**
SHES-TL (48 cores)	1 ± 0	3.65^9	10
SHES-NS (48 cores)	0 ± 0	4.07^9	10
HIRO (ours)	0.52 ± 0.36	10^7	12
HIRO (Nachum et al.)	0.92 ± 0.04	$\mathbf{10^7}$	*Unknown*
FuN	0.56 ± 0.39	10^7	*Unknown*
SNN4HRL	0.02 ± 0.01	10^7	*Unknown*
VIME	0.02 ± 0.02	10^7	*Unknown*

Table 5: Task performance of SHES, SHES variants and gradient-based methods on **Ant Push**. Performance is the average of 10 randomly seeded trials with standard error. Performance of all methods that are not SHES is taken from Nachum et al. [2018].

Method	Test reward	Steps	Time (h)
SHES (24 cores)	0 ± 0	2.11×10^9	10
SHES (48 cores)	0 ± 0	3.63×10^9	10
SHES (120 cores)	0 ± 0	7.5×10^9	10
SHES (240 cores)	0 ± 0	7.23^9	10
SHES (600)	0 ± 0	7.5^9	**6.3**
SHES-TL (48 cores)	0.6 ± 0.52	3.67^9	10
SHES-NS (48 cores)	0 ± 0	3.19^9	10
HIRO (ours)	0.15 ± 0.3	10^7	15
HIRO (Nachum et al.)	$\mathbf{0.66 \pm 0.07}$	$\mathbf{10^7}$	*Unknown*
FuN	$0.0.07 \pm 0.22$	10^7	*Unknown*
SNN4HRL	0 ± 0	10^7	*Unknown*
VIME	0 ± 0	10^7	*Unknown*

Table 6: Task performance of SHES, SHES variants and gradient-based methods on **Ant Fall**. Performance is the average of 10 randomly seeded trials with standard error. Performance of all methods that are not SHES is taken from Nachum et al. [2018].

Hyper-parameter	**SHES**	**HIRO**
Controller Interval	25	10
Controller distance	4	10
Learning rate	0.01	0.001
Sigma	0.02	*n/a*
Episodes per policy	5	*n/a*
Policy per generation	1000	*n/a*
Time horizon (Agent lifetime)	500	500
Experiment parameter	**SHES**	**HIRO**
Runs	10	10
Generations	3000	3000
Run-time Limit	10 hrs	12 hrs
CPU Cores	48, 240, 600	24 + Nvidia V100 GPU

Table 7: Experiment and method (SHES, HIRO) parameters running tuning experiments for SHES (section 4.2.1).

4.3.3 Graph Description

4.3.3.1 Ant Gather For ant gather HIRO achieves its mean test reward of 3.02 in 10 million steps, however, figure 24 (a) shows our experiments on HIRO did not

match this performance, only reaching a test reward of 1.22 after 10 million steps and 12 hours. This mismatch is unexpected as we used the official repository with no modifications. Whereas SHES can achieve a mean test reward of 3.76 in 860 million steps, which is 86x the number of samples used by HIRO. Importantly on 600 cores, SHES can achieve HIROs test reward of 3.02 in 1 hour and 5 minutes as opposed to the 12 hours it takes to fully train HIRO for 10 million steps.

Figures 26 (a) and 25 (a) show that an increase in CPU core count leads to an increase in learning speed per wall clock time. SHES on 48 cores reaches a test score of 2.5 in 4 hours and 20 minutes, while on 120 cores it takes 2 hours and 40 minutes to reach the same test score, on 240 cores it takes 1 hour and 50 minutes and on 600 cores it takes 40 minutes.

Figure 27 (a) show that SHES-TL plateaus after reaching a training reward of 1.6 in 1 hour. SHES-TL was trained on 48 cores and up to its plateau, it matches the speed of SHES trained on 240 cores, which reaches the same training reward in 50 minutes. Figure 28 (a) shows a reason for this speed, SHES-TL's pretrained primitive reward rises very sharply to its maximum as opposed to the slower rise of SHES' primitive. Figure 27 (a) displays SHES-NS' poor performance as it is unable to achieve much above a 0 train reward, whereas figure 28 (a) shows the impressive, but highly variable performance of the primitive.

Figure 29 shows that SHES-onehot does improve SHES' overall performance, however, it is not significant ($p = 0.24$). It achieves a maximum accuracy of 4.18 compared to SHES' maximum accuracy of 3.7. Even though the performance increase is not significant figure 29 clearly shows that SHES-onehot's learning curve rises faster than SHES' learning curve on the same number of CPU cores.

4.3.3.2 Ant Maze Figure 24 (b) shows SHES' poor performance on ant maze, where neither the 48 core nor 600 core could achieve over 0 test reward. In this environment, the results of our HIRO experiments match the results from Nachum et al. where it achieves a maximum reward of 0.99 in 12 hours and 10 million steps [Nachum et al., 2018].

Figure 25 (b) is flat because SHES was unable to achieve a test reward of more than 0, however, figure 26 (b) shows a more informative graph with the same experiments plotted using training reward. One can see that to reach a training reward of -5000: SHES on 600 cores takes 30 minutes, SHES on 240 cores takes 2 and 30 minutes, finally SHES on 48 cores takes 5 hours.

Figures 27 (b) and 28 (b) shows the impressive performance of SHES-TL and the poor performance of SHES-NS. SHES-TL's train reward rises faster than SHES' train reward and obtains a significantly higher reward ($p < 0.0001$) in contrast to

SHES-NS which fails to learn and is significantly less performant than SHES ($p < 0.0001$). Interestingly SHES-NS' primitive achieves the highest task performance and is significantly better than both SHES-TL ($p = 0.0007$) and SHES ($p < 0.0001$), but with high variance.

4.3.3.3 Ant Push Figure 24 (c) shows that SHES on 48 cores performance lies between the original HIRO's performance and our HIRO experiments performance. SHES 48 core achieves a test reward of 0.8 which is significantly better than our HIRO experiments test reward of 0.52 ($p = 0.01$), but worse than HIRO's original test reward of 0.92. SHES on 48 cores matches the speed of our HIRO experiments up until 4 hours where SHES' performance improves past HIRO's maximum. While SHES on 48 cores is unable to match the original performance of HIRO, SHES on 240 and 600 cores is able to surpass it's performance achieving the maximum reward of 1.

Figures 25 (c) and 26 (c) again show SHES' scalability across multiple cores. SHES on 600 cores rises the fastest, with SHES on 240 cores rising faster than SHES on 48 cores, all in both test and train reward. Final training rewards are quite similar at around -6000, however, SHES on 600 cores and 240 cores achieve the maximum test reward of 1, while SHES on 48 cores achieves a test reward of 0.8.

Figure 27 (c) shows that SHES and SHES-TL display very similar maximum training rewards with SHES-TL rising faster than SHES, while SHES-NS fails to learn, being significantly worse than SHES (0.0007). Figure 28 (c) again shows very similar primitive performance between SHES and SHES-TL, with SHES-NS having higher primitive performance than both SHES ($p = 0.25$) and SHES-TL ($p < 0.0001$).

4.3.3.4 Ant Fall Figure 24 (d) shows that SHES on 48 cores is unable to achieve more than 0 test reward. It also shows that HIRO (ours) is unable to match the performance of HIRO (Nachum et al.) only achieving 0.15 versus the 0.66 of HIRO (Nachum et al.).

Figure 25 (d) shows that no amount of cores can allow SHES to learn for long enough to achieve a greater than 0 reward. However, figure 26 (d) shows that SHES on all number of cores achieves similar test reward, however, higher core counts allow for higher learning speed.

Figure 27 (d) shows the impressive performance of SHES-TL and the poor performance of SHES-NS. SHES-TL can significantly improve on the performance of SHES ($p < 0.0001$) and as can be seen in table 6 it can achieve a test reward of 0.6, while SHES is not able to achieve a greater than 0 test reward. Figure 28 (d) shows that SHES-NS has the highest rewarded primitive being significantly better than SHES

($p < 0.0001$) and SHES-TL ($p < 0.0001$).

4.3.3.5 Other graphs Figure 31 shows the factor by which you can expect SHES' run-time to decrease when using different numbers of cores. As a reference, the perfect linear speedup is plotted on the same graph along with S-ES speed. Figure 30 shows the fitness objective weighting of a randomly selected SHES-NS run on the ant maze environment. The novelty objective weighting is calculated as $1 - fitness\_weighting$.

5 Discussion

In this section we analyse the results of our experiments to determine the learning speed (run-time) and performance of SHES and its extensions. We make specific reference to the research goals outlined in section 1.1.

5.1 Speedup

Figure 31 presents the SHES method's computational speedup as a function of the number of cores plotted versus perfect linear speedup. SHES has sub-linear speedup, but with a linear trend. This is not unexpected given the strictly serial parts of SHES, namely fitness shaping, ranking and the gradient calculation. SHES offers approximately a one-sixth speedup for each core added, which was tested up to 600 cores. Interestingly SHES is able to achieve more speedup per core than S-ES, this is likely due to the higher proportion of parallel work it needs to perform, because of its extra policy. This clearly shows that SHES has retained the speedup of S-ES, which is a key factor in the usefulness of this algorithm. To further illustrate, consider that SHES (600 cores) was able to match the test score of HRL with Off-Policy Correction (HIRO Nachum et al.) in under an hour, on both the ant gather and ant push tasks. Replication of HIRO on these tasks (running for 10 million training steps [Nachum et al., 2018]), took over 12 hours to achieve the same test score when executed on an Nvidia V100 GPU. Thus SHES offers at least a 12× learning speedup over the HIRO method in these environments.

Given that SHES has unavoidably serial sections, one may assume that due to Ahmdals law [Gustafson, 1988] there is a point where adding more cores does not decrease the runtime. However, when given enough compute to saturate the parallel portion of this algorithm one can simply increase the amount of parallel work needing to be done without having a major impact on performance. This is done by increasing the evaluated policies in a single generation, this not only provides more available parallel work but can also increase the accuracy of the gradient update, especially for policies with many parameters. Increasing the policies-per-generation and episodes per policy parameters does have diminishing returns in terms of increasing performance (see figures 20 and 21), but allows for the use of many more CPU cores and larger models.

The scalability and learning speed of SHES relative to state-of-the-art gradient-based methods satisfy research goal one (section 1.1) as it is clear that SHES can scale to multiple CPUs and is faster than gradient-based methods in terms of wall-clock time when given adequate compute.

5.2 SHES performance

Figures 24 (a), (b), (c) and (d) show that SHES is competitive with the previous state-of-the-art (HIRO) and with other gradient-based methods as can be seen in tables 3, 4, 5 and 6. In ant gather it improves upon the previous state-of-the-art's mean test reward by a factor of 1.24 (see table 3) and significantly improves on our HIRO experiments (our HIRO experiments use their official repository) ($p = 0.008$, Mann–Whitney U test [Flannery et al., 1986]). As for ant push, SHES is statistically better ($p = 0.0007$) than our HIRO experiments and SHES on 25 nodes is 1.09 times better than the results originally reported HIRO (table 5). On ant maze and fall, while SHES is able to learn a strategy it is not performant enough to achieve above a 0 test reward and thus is statistically worse than our HIRO's performance ($p < 0.0001$).

These results show that SHES has bettered HIRO on half of the environments tested, however, this does come at the cost of sample efficiency. SHES requires approximately 100× more samples than HIRO in order to obtain this performance (tables 3, 4, 5 and 6), while this is a large difference it is not unexpected given that gradient-free optimization is often less sample efficient than gradient-based optimization [Sigaud and Stulp, 2019]. Specifically, the original S-ES paper saw sample efficiency up to approximately 8 times worse than TRPO on simple 2D locomotion environments [Salimans et al., 2017], meaning one would expect even more sample inefficiency on harder 3D locomotion and navigation tasks. SHES' sample inefficiency is emphasized by the fact that it is compared to HIRO, an off-policy HRL method specifically made to be sample efficient.

While SHES shows unsurprisingly poor sample efficiency it is able to make up for this by being faster than HIRO in terms of wall-clock time when given enough compute. This can be most clearly seen in figure 24 (c) where performance is quite similar between HIRO and SHES. SHES on 600 cores rises faster than HIRO when plotted against time, indicating greater learning speed and showing the benefit of its easy scalability. While on ant gather SHES on 600 cores is able to match HIRO's test reward of 3.02 performance in under an hour. Given SHES' poor sample efficiency, but impressive learning speed it would be most suitable for situations where samples are computationally cheap to produce and wall time is critical.

While SHES performed poorly on ant maze and fall in terms of test reward, figures 26 (b) and (d) show that it does still learn, as its training reward clearly increases over time. However, it is unable to perform well enough to obtain an above zero test reward. While this performance is poor, table 4 and 6 shows that SNN4HRL and VIME both achieve the same test reward. Interestingly SHES on 24 cores is able to achieve a test reward of 0.2 (table 4) on ant maze, showing that it is able to solve

the environment. It is unlikely that this performance is a result of the number of cores, but rather a *lucky* parameter initialization for the experiments done on SHES 24 core.

It is likely the poor task performance is due to the short time horizon (500 steps), where an agent can barely reach the end goal in the allotted time, thus increasing the task difficulty. When viewing video replays[5] of the best performing SHES agents on ant maze we found that they were able to reach goals about 75% of the way through the maze, but could not reach all the way to the end goal within the short time horizon. Couple this with the fact that when given a longer time horizon (1000 steps) SHES is able to learn to reach the end goal consistently and one can conclude that SHES struggled to solve this problem because of its relatively short time horizon. We did not use a longer time horizon as it would not have allowed fair comparison with previous works. As for ant fall when viewing video replays of SHES agents, it is clear that it was unable to learn the optimal strategy of pushing the block into the chasm and walking over it to reach the target. Instead, it gets trapped in the local minima of falling down the chasm and being unable to make it close enough to the target on the other side. It is possible that this is also due to the same short time horizon (500 steps) used in ant maze as longer time horizon tests were also able to solve this environment.

Another possible explanation for poor task performance is that controllers are ranked against each other, but depend heavily on their primitives performance. Thus this ranking is not perfectly fair, this could hinder exploration as a well performing controller may be paired with a low performing primitive and therefore receive a low ranking. Addressing this is left to future work and discussed more in section 6.1.

SHES performance on ant gather also shows the benefit of S-ES's indifference to delayed rewards in HRL problems. Ant gather is the environment with the most sparse reward and the environment on which SHES outperforms HIRO by the highest margin. This seems to show the applicability of SHES to sparse reward problems and the benefit of using S-ES over gradient-based methods.

Given SHES' performance on ant gather, maze, push and fall it is clearly competitive with HIRO seeing as it is able to outperform HIRO on ant gather and push. While it is not able to achieve an above zero test reward on ant maze and fall one can see from figures 26 (b) and (d) that it is still able to learn in these environments as the training reward increases over time. Thus, this satisfies research goal two, which is to create an HRL framework that is competitive with state-of-the-art gradient-based methods (tables 3, 4, 5 and 6) and general enough to learn across multiple tasks.

[5] `https://github.com/sash-a/HrlMuJoCoEnvs.jl/blob/master/README.md`

5.3 Extension performance

This section discusses the performance of the transfer learning, novelty search and one-hot extensions to SHES.

5.3.1 Transfer learning

Transfer learning seems to have both benefits and drawbacks for SHES. For all environments it is able to improve learning speed as can be seen by the steeper rise of the SHES-TL curves in figures 27 (a), (b), (c) and (d). However, it falls significantly short of SHES' maximum train reward ($p = 0.0003$) in the ant gather environment, while it improves on SHES' max train reward for ant push the improvement is not significant ($p = 0.26$). SHES-TL is able to significantly improve on SHES' maximum training reward for ant maze ($p < 0.0001$) and fall ($p = 0.005$) where it is able to achieve a positive test reward of 0.6 unlike SHES, which was unable to achieve a test reward over 0 for the ant fall environment (table 6). When comparing SHES-TL to SNN4HRL [Florensa et al., 2017], which is a method that also uses pretraining (see section 2.2.2.4) it is clear that SHES-TL displays greater task performance on all environments except ant maze, where it matches the performance of SNN4HRL (tables 3, 4, 5 and 6).

The graphs showing the learning speed improvement (figures 27) do not include the time taken to pretrain the primitive, which took approximately 10 hours on 48 cores. When only considering these four tasks one could add roughly three hours to the training time of SHES-TL, which would put it on par with SHES in terms of learning speed. However, seeing as the same primitive could be used in multiple other similar task settings it is difficult to say exactly what the time impact is of pretraining a primitive.

In all cases, SHES-TL has the most consistently high performing (low variance) primitive as can be seen from figures 28 (a), (b), (c) and (d). This indicates that the pretraining phase was successful and that the primitive learned a general strategy for walking towards a goal, as such the pretrained primitive clearly was a contributing factor in SHES-TL's increased learning speed in all environments and task performance in ant maze, push and fall.

On ant gather SHES-TL shows an initial period of rapid learning when compared to SHES, which can be seen in figure 27 (a) followed by a plateau in the training reward. The initial period of rapid learning is expected, given that the primitive is already trained and SHES-TL is able to learn rapidly in other environments, however, the plateau is surprising. There are two possible hypotheses for the cause of this plateau: first, similar to what Nachum et al. observed, the plateau could be due to the lack of specialization of the primitive to its controller [Nachum et al., 2018]. Second, the

plateau could be caused by the frozen weights of the primitive, which contribute to SHES' exploration. If these hypotheses were the reason for the plateau one would expect to see a similar graph for ant maze, push and fall, however, this is not the case, in fact figures 27 (b), (c) and (d) show that SHES-TL both learns faster and achieves a higher reward than SHES on ant maze, push and fall. Given that ant maze, push and fall offer a different reward to ant gather. In ant gather reward is only provided when the agent makes contact with a food or poison item, while in the other environments reward is given as distance to the target at every time-step. Consequently, one could conclude that SHES-TL has difficulty learning in sparse reward environments wherein one should rather utilize SHES.

In order to address SHES-TL's poor performance in sparse reward environments, one must address the two hypotheses. The first hypothesis, the fact that the primitive is unable to specialize to its controller, is caused by the frozen weights of the primitive and it leads to a less than optimal primitive per task. The second hypothesis: SHES-TL lacks the robust exploration of SHES, is also caused by the frozen weights of the primitive. During training in SHES, the primitive's behaviour is constantly changing, this changing behaviour allows SHES to explore different areas of the search space, thus implicitly improving its exploration. Both of these drawbacks are caused by the frozen weights of SHES-TL's primitive, however unfreezing these weights leads to notably worse performance, which is caused by the primitive quickly specializing to the ineffectual goals set by early controllers, thus losing all benefits of pretraining, without removing the non-stationary problem (a problem frozen weights solve). Thus a possible solution to both of these problems that address the drawbacks of a non-frozen primitive is to have an initial period of learning with a frozen primitive, until the desired performance is reached and then unfreeze the weights of the primitive, which would allow it to specialize to its controller and contribute to the overall exploration of the system, without losing all the benefits of pretraining.

These results show transfer learning's applicability can come both in the form of learning speed and task performance, however, it also shows that SHES is able to perform competitively against a transfer learning method, which has innate benefits over SHES (discussed in sections 3.4 and 4.1.2), thus complementing SHES' already competitive performance with state-of-the-art. SHES-TL satisfies half of the third research goal in that transfer learning was investigated as a means to improve exploration, task performance and sample efficiency. It does improve learning speed across all environments, outperforming SHES in ant maze and push, but unable to match SHES' performance in ant gather.

5.3.2 Novelty search

Overall the novelty search extension performed poorly, it was intended to aid in controller exploration, which would have been most beneficial to the ant maze, push and fall environments. However, it seems to only have been a detriment to performance as can be seen in figures 27 (a), (b), (c) and (d). In all environments, it fails to achieve meaningful rewards and is statistically worse than SHES ($p < 0.0001$).

This is most likely caused by novelty search guiding the controller policy to areas of low reward in parameter space. This leaves SHES in a bad cycle of novelty becoming highly weighted, SHES moving parameters to areas of low reward, which increases novelty's weighting even further and so on. Unlike what was observed by Conti et al., when the novelty objective has a high weighting SHES policies were not able to escape the local minima, thus the novelty weighting stayed low, an example of this can be seen in figure 30.

However, SHES-NS' poor performance could also be caused by a more fundamental issue with how controller novelty is calculated. In this work, behaviour is characterised by the ending position of the agent, which is the same characterization Lehman and Stanley and Conti et al. used and is intended to encourage the controller to recommend novel areas for the primitive to explore. The issue that arises is the controller does not have direct control over where the agent ends up, that is the role of the primitive, which leads to this being a sub-optimal behaviour characterization for this specific use-case and would likely need some future research to be improved. Combining novelty search with an HRL method for controller exploration has yet to be performed successfully and is an interesting avenue to investigate, as novelty search has the potential to greatly aid in exploration for hard exploration problems, especially harder problems than the ones tested in this work, such as more intricate mazes. As it stands this specific area requires more research.

An interesting side-effect of poor controller performance is the high primitive performance. In all environments, the SHES-NS primitive achieved the highest reward (figures 28 (a), (b), (c), (d)). It is hypothesized that this is due to the controller recommending *easy* goals to the primitive, as this explains the controllers poor performance since easy goals do not necessarily provide high environmental reward and it explains the high performance of the primitive where it easily attains these goals, thus gaining a high reward.

This satisfies half of the third research goal, where the aim was to investigate the impact of novelty search on SHES in terms of exploration, task performance and sample efficiency. Novelty search did not have the desired impact on performance or exploration as it was only a detriment to performance and its exploration provided no benefit in the deceptive ant maze, push and fall tasks.

5.3.3 One-hot performance

Figure 29 shows that SHES-onehot does improve SHES' overall performance, however, it is not significant ($p = 0.24$) it achieves a maximum test reward of 4.18 compared to SHES' maximum test reward of 3.7. Even though the performance increase is not significant figure 29 clearly shows that SHES-onehot is able to learn faster than SHES. The results show that one-hot encoding is a beneficial addition to SHES, however, its applicability is limited, since it can only be applied to environments where the robot has a sensor, making its performance benefit highly task dependant.

Importantly this extension is not limited to SHES and can be applied to other methods that train in suitable environments, thus increasing the applicability of the extensions. It should be noted that this method of one-hot encoding is untested on other methods, but SHES-onehot's performance indicates that it could be a positive addition to existing methods.

5.4 Hyperparameter Robustness

The reason for hyperparameter tuning is twofold, it was used both to obtain optimal parameters for SHES and to show that SHES is an HRL algorithm that is robust to hyperparameter changes. As can be seen in figures 18, 20 and 21 the target distance, policies per generation and episodes per policy parameters show minimal performance difference across all values tested, with most maximum training rewards being insignificantly different from each other and within a single standard deviation. This speaks to the robustness of SHES to hyperparameter changes as it is able to perform well, across all environments, regardless of the value of these parameters. The controller interval parameter seems to be an exception to this rule as it shows that a value of 100 is significantly better for the ant maze, push and fall environments, whereas this value performs poorly on the gather environment. This shows how critical the controller interval parameter is to the performance of SHES.

The hyperparameters learning rate and sigma show the most deviation between values, however the most performant hyperparameter values found by the hyperparameter search of 0.01 for learning rate and 0.02 for sigma are the same values used by Salimans et al. in the original S-ES implementation. This shows the lack of tuning required for SHES as it is likely that optimal, or close to optimal, values have already been found for these parameters for a diverse set of environments including Atari games, basic MuJoCo walking environments and the complex navigation environments tested in this work. This shows that it is mostly unnecessary to tune SHES for a range of problems, this is in stark contrast to current gradient-based HRL methods, which are brittle hyperparameter changes [Paine et al., 2020], thus

often needing to be tuned.

6 Conclusion

The main contribution of this work is a new evolutionary HRL method: Scalable Hierarchical Evolution Strategies (SHES). Across all hard HRL tasks tested, SHES achieved significant computational speedup and showed that it was scalable up to at least 600 CPU cores, allowing it to solve problems twelve times faster than the current state-of-the-art off-policy HRL method: HIRO. However, this increased learning speed does come at the cost of sample efficiency where SHES was notably worse than HIRO. Importantly, SHES outperforms HIRO in terms of task performance on two out of the four tasks tested and due to S-ES' in-variance to delayed rewards SHES performs especially well on sparse reward RL tasks.

We extend SHES with transfer learning, this shows the benefit of a single reusable primitive, which can speed up learning for all tasks and improve task performance (training reward) in most tasks. Thus, speaking to the performance of SHES as it is comparable to a similar method with a pretrained primitive. Another extension to SHES was tested in the form of novelty search with the intention of improving SHES' exploration characteristics, however, this performed poorly having significantly worse task performance in all environments.

Thus, we provide a novel evolutionary HRL method, which addresses a current need for computationally expedient HRL methods that yield high task performance across a range of hard HRL (and more generally RL) tasks. This method is most appropriate when samples are computationally cheap to produce and wall-clock time is critical.

6.1 Future Work

6.1.1 Unfair Controller Rankings

In theory, a better way to alleviate unfair controller comparisons caused by low performing primitives is to weight controller rewards by their primitive's performance, such that if both a primitive and controller perform poorly one ignores the controller in the optimization step or re-evaluates it using a new primitive. However, this was not done in this work as it is difficult to tell if a primitive is performing poorly. One would need to design a threshold that defines the line between a poorly and well-performing primitive throughout the training process (as the primitives performance is increasing) as such this is left for future work.

6.1.2 Improving Sample Efficiency

SHES biggest disadvantage is its sample inefficiency, this is caused by the inefficiency of S-ES (the method used to train the policies). Work has been done by Liu et al.

that shows that S-ES can be made much more sample efficient through the use of trust regions [Liu et al., 2019]. A future avenue that could be investigated is using similar methods that were used by Liu et al. to improve the sample efficiency of SHES, in the process solving its biggest disadvantage.

References

Shane Acton, Sasha Abramowitz, Liron Toledo, and Geoff Nitschke. Efficiently coevolving deep neural networks and data augmentations. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2543–2550. IEEE, 2020.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *International Conference on Machine Learning*, pages 507–517. PMLR, 2020.

Aijun Bai, Feng Wu, and Xiaoping Chen. Online planning for large Markov decision processes with hierarchical decomposition. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(4):45:1–45:28, Jul 2015.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – a comprehensive introduction. *Natural Computing*, 1:3–52, 2002. ISSN 1567-7818. doi: 10.1023/A:1015059928466.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671.

Shweta Bhatt. 5 things you need to know about reinforcement learning, 2018. URL `https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html`.

Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7 (1):1–41, 2013.

Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V Arnold, and Tim Hohm. Mirrored sampling and sequential selection for evolution strategies. In *International Conference on Parallel Problem Solving from Nature*, pages 11–21. Springer, 2010.

Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. *Advances in Neural Information Processing Systems*, 2018-December:5027–5038, 12 2017.

Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15 (49):5, 2013.

Cheng Cui, Zhi Ye, Yangxi Li, Xinjian Li, Min Yang, Kai Wei, Bing Dai, Yanmei Zhao, Zhongji Liu, and Rong Pang. Semi-supervised recognition under a noisy and fine-grained dataset, 2020.

Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

Sélim Datoussaïd, Guy Guerlement, and Thierry Descamps. Applications of evolutionary strategies in optimal design of mechanical systems. *Foundation of Civil and Environmental Engineering*, 7:35–51, 2006.

Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1993.

Bhuwan Dhingra, Qiao Jin, Zhilin Yang, William Cohen, and Ruslan Salakhutdinov. Neural models for reasoning over multiple mentions using coreference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 42–48, 2018.

Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Miguel Duarte, Jorge Gomes, Sancho Moura Oliveira, and Anders Lyhne Christensen. Evorbc: Evolutionary repertoire-based control for robots with arbitrary locomotion complexity. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 93–100, 2016.

A. Eiben and J. Smith. From Evolutionary Computation to the Evolution of Things. *Nature*, 521(1):476–482, 2015a.

A. Eiben and J. Smith. *Introduction to Evolutionary Computing (2nd edition)*. Springer, Berlin, Germany, 2015b.

B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, UK, 1986.

Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1704.03012*, 2017.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

Iris Ginzburg and Haim Sompolinsky. Theory of correlations in stochastic neural networks. *Physical review E*, 50(4):3171, 1994.

Jorge Gomes and Anders Lyhne Christensen. Comparing approaches for evolving high-level robot control based on behaviour repertoires. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–6. IEEE, 2018.

Jorge Gomes, Sancho Moura Oliveira, and Anders Lyhne Christensen. An approach to evolve and exploit repertoires of general robot behaviours. *Swarm and Evolutionary Computation*, 43:265–283, 2018.

Santiago Gonzalez and Risto Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *International conference on artificial neural networks*, pages 799–804. Springer, 2005.

William H. Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The minerl competition on sample efficient reinforcement learning using human priors. *NeurIPS Competition Track*, 2019a.

William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019b. URL `http://minerl.io`.

John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5): 532–533, 1988.

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.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

Majd Hawasly and Subramanian Ramamoorthy. Lifelong transfer learning with an option hierarchy. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1341–1346. IEEE, 2013.

Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers, 2016.

Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. *Advances in Neural Information Processing Systems*, 29:1109–1117, 2016.

Andrej Karpathy, Tim Salimans, Jonathan Ho, Peter Chen, Ilya Sutskever, John Schulman, Greg Brockman, and Szymon Sidor. Evolution strategies as a scalable alternative to reinforcement learning, Mar 2017. URL `https://openai.com/blog/evolution-strategies/`.

Adam Katona, Daniel W Franks, and James Alfred Walker. Quality evolvability es: Evolving individuals with a distribution of well performing and diverse offspring. *arXiv preprint arXiv:2103.10790*, 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347, 1997.

Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*, pages 491–507. Springer, 2020.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, 2012.

Heinrich Kuttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktaschel. The nethack learning environment, 2020.

Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011a.

Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218, 2011b.

Guoqing Liu, Li Zhao, Feidiao Yang, Jiang Bian, Tao Qin, Nenghai Yu, and Tie Yan Liu. Trust region evolution strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4352–4359. AAAI Press, 7 2019. ISBN 9781577358091. doi: 10.1609/aaai.v33i01.33014352.

Gong Mao-Guo, Jiao Li-Cheng, Yang Dong-Dong, and Ma Wen-Ping. Evolutionary multi-objective optimization algorithms. 2009.

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):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

J-B Mouret and Stéphane Doncieux. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 20(1):91–133, 2012.

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 31:3303–3313, 2018.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Near-optimal representation learning for hierarchical reinforcement learning. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1emus0qF7`.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

Tom Le Paine, Cosmin Paduraru, Andrea Michi, Caglar Gulcehre, Konrad Zolna, Alexander Novikov, Ziyu Wang, and Nando de Freitas. Hyperparameter selection for offline reinforcement learning. *arXiv preprint arXiv:2007.09055*, 2020.

Xue Bin Peng, Michael Chang, Grace Zhang, Pieter Abbeel, and Sergey Levine. Mcp: Learning composable hierarchical control with multiplicative compositional policies. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3681–3692. Curran Associates, Inc., 2019. URL `http://papers.nips.cc/paper/8626-mcp-learning-composable-hierarchical-control-with-multiplicative-composition.pdf`.

Kedar Potdar, Taher S Pardawala, and Chinmay D Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International journal of computer applications*, 175(4):7–9, 2017.

Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016. ISSN 2296-9144. doi: 10.3389/frobt.2016.00040. URL `https://www.frontiersin.org/article/10.3389/frobt.2016.00040`.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc Le. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pages 1–10. AAAI, 2019.

Ingo Rechenberg. Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment Library Translation 1122*, 1965.

Hongyu Ren, Shengjia Zhao, and Stefano Ermon. Adaptive antithetic sampling for variance reduction. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5420–5428. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/ren19b.html`.

Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29:2234–2242, 2016.

Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017. URL `http://arxiv.org/abs/1703.03864`.

Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

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.

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

Hans Schumann, Gregg Willcox, Louis Rosenberg, and Niccolo Pescetelli. "human swarming" amplifies accuracy and roi when forecasting financial markets. In *2019 IEEE International Conference on Humanized Computing and Communication (HCC)*, pages 77–82, 2019. doi: 10.1109/HCC46620.2019.00019.

Olivier Sigaud and Freek Stulp. Policy search in continuous action domains: an overview. *Neural Networks*, 113:28–40, 2019.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Jorge Sola and Joaquin Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on nuclear science*, 44(3):1464–1468, 1997.

Kenneth Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

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

Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999a.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999b.

Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training, 2021.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549. PMLR, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Michael D Vose. *The simple genetic algorithm: foundations and theory*. MIT press, 1999.

Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15:949–980, 2014.

Wenhao Yu, Greg Turk, and C Karen Liu. Learning symmetric and low-energy locomotion. *ACM Transactions on Graphics (TOG)*, 37(4):1–12, 2018.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. *arXiv preprint arXiv:2106.04560*, 2021.

Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*, 2020.

A Appendix

A.1 Code Structure

This work uses three packages developed in the Julia language [Bezanson et al., 2017] namely HrlMuJoCoEnvs.jl[6] that manages environments, ScalableES.jl[7] that replicates the work of Salimans et al. in Julia and ScalableHrlEs.jl[8] that builds on ScalableES.jl using Julia's multiple dispatch system in order to modify the functionality to work with multiple policies. ScalableES.jl implements methods such as `step_es` and `approxgrad` which ScalableHrlEs.jl can override using its own data structures in order for it run with two policies. This approach allows for minimal code duplication between the two packages and allows for the low level MPI commands to be mostly ignored when extending ScalableES.jl.

A.2 Loading Checkpoint Models

Checkpoints of models and raw logs have been saved in the in an online repository[9]. One can run these checkpoints by first installing HrlMuJoCoEnvs.jl, ScalableES.jl and ScalableHrlEs.jl in the same folder, then using the checkpoints inside of `msviz.jl/checkpoints` one can run the following:

```
cd ScalableHrlEs.jl
julia --project scripts/runsaved.jl
    ../msviz.jl/checkpoints/AntPush 3000 AntPush
```

One can also view videos of the checkpointed models instead in the HrlMuJoCoEnvs.jl readme.

[6] https://github.com/sash-a/HrlMuJoCoEnvs.jl
[7] https://github.com/sash-a/ScalableES.jl
[8] https://github.com/sash-a/ScalableHrlEs.jl
[9] https://github.com/sash-a/msviz