# Understanding the Slope Foraging Problem

Mostafa Rizk

September 9, 2020

## Contents



# 1 Executive Summary

# 2 Introduction

We're interested in evolving multi-agent cooperation. Use Slope Foraging as test bed. Insights gained from this relatively simple setup can hopefully be applied to other tasks that share attributes with this one. Want to place this problem in the context of known ones in the literature. How difficult is it? We use rwg benchmarking to compare it to known tasks. We then discuss Stag Hunt games.

# 3 Task Description

The task is modelled off the foraging behaviour of the Atta Leafcutter ant, as in Ferrante et al [1] and Pini et al [2, 3]. In nature, the ants cut leaves from a tree and take them to their nest. Sometimes the ants partition the task. When they do so, some ants are droppers, who cut leaves and let them fall, while other ants are collectors who collect fallen leaves from the base of the tree and take them to the nest. This partitioned approach is advantageous because gravity transports leaves faster than ants can. Rather than every ant climbing up the tree, cutting a leaf and bringing it to the nest, when they partition the ants are able to transport more leaves in the same time-span while consuming less energy.

We model this scenario similarly to Ferrante et al [1], with a slope in place of the tree trunk. Pini et al use a slightly different implementation where there resources can be transported via a long corridor or a booth with a wait time [2, 3]. Pini et al's approach may be useful for comparison at a later stage of the research. Much like the ants, agents transport resources from the source to the nest. An agent on a team can use a generalist strategy where it acts individually, going up and down the slope to retrieve resources. An agent can also use a specialist strategy. As a specialist it can either be a dropper, going up the slope once and dropping things from the nest, or it can be a collector and gather the resources that accumulate at the base of the slope, called the cache. Much like real robots that deplete their battery and ants that deplete their energy stores, there is a cost to moving and it is compounded when going up the slope. Teams that use complementary specialist strategies pay a smaller energy cost as well as time cost, since resources slide down faster than they can be carried. Preliminary analysis with hard-coded agents verifies that specialist teams gain higher overall reward than generalist teams.

More formally, you have a team of $n_{agents}$ agents. They are placed in a rectangular arena that is $l$ tiles long and $w$ tiles wide, illustrated in Figure 1 and Figure 2. The arena is divided into four sections $l = l_{nest} + l_{cache} + l_{slope} + l_{source}$. Each episode is composed of a number of finite time-steps $t = 0, ....T$. Since 3D physics is computationally expensive we use a 2D environment to expedite our experiments as the focus of our research is on the evolutionary process and team dynamics rather than the robotic element. We also use a discrete scenario, as opposed to a continuous one for further simplicity. To simulate the presence of gravity, resources move when on the slope, at a speed greater than the agents are capable of. The high sliding speed creates evolutionary

pressure for the team to specialise. Agents travel at speed $s_{agent}$ and resources slide when placed on the slope with a speed of $s_{resource}$ Additionally agents pay costs for moving. This simulates the presence of a battery, with energy expenditure varying depending on where the agent is moving. There is a base cost to moving $c$ paid by the agent for moving in any direction. The base cost is multiplied by different factors for moving up the slope ($f_{up}$) down the slope ($f_{down}$) and moving while carrying a resource ($f_{carry}$). An agent moving sideways on the slope pays the same cost as one moving on a non-slope area in any direction. An agent moving one tile up the slope at time step $t = 0$ while carrying a resource, for example, pays a cost of $C_0 = f_{up} \cdot f_{carry} \cdot c$ . There are $n_{resources}$ resources at the source, initially. Every time a resource is removed from the source, another one appears at the source so there are always at least $n_{resources}$. Each resource retrieved provides all team members a reward of $R$. The values we chose for these parameters can be found in the appendix

Fitness can be calculated for the team or for an individual depending on the level of selection. When calculating fitness for a team of agents, the fitness function is as follows:

$$F = \sum_{t=1}^{T} \sum_{i}^{n_{agents}} (R_{ti} - C_{ti})$$

That is, for each agent, at each time step, we calculate the reward it received at that time step (whether from retrieving a resource itself or from another agent retrieving a resource) and we subtract the cost it individually paid at that time step. We then take the summation of this calculation for all agents over all time steps in the simulation. The reward and cost for an agent $i$ at time step $t$ can be computed as shown here:

$$R_{ti} = R \cdot r_t$$

where $r_t$ is the total number of resources retrieved by all agents at time $t$

$$C_{ti} = \begin{cases} c & notonslopeormovedsidewaysonslope \\ c \cdot f_{up} & upslope \\ c \cdot f_{down} & downslope \\ c \cdot f_{carry} & notonslopeormovedsidewaysonslopewhilecarrying \\ c \cdot f_{carry} \cdot f_{up} & upslopewhilecarrying \\ c \cdot f_{carry} \cdot f_{down} & downslopewhilecarrying \end{cases}$$

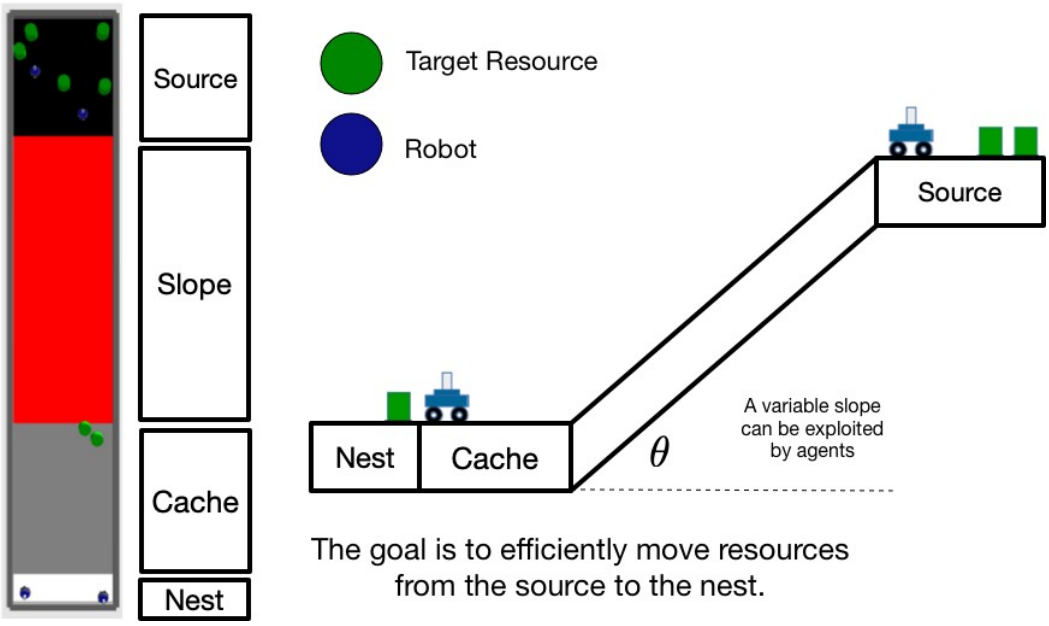


Figure 1: Arena Layout

When calculating fitness for an individual agent, the fitness function is as follows:

$F = \sum_{t=1}^{T}(R_{ti} - C_{ti})$

Example: Agent 1 incurs -200 retrieving resource. Agent 2 incurs -100 wandering around cache. Agent 1 score= 1000 - 200. Agent 2 score = 1000 - 100. Team selection: team score = agent 1 score + agent 2 score = 800 + 900 = 1700 Individual selection: agent 1 score = 800, agent 2 score = 900. In team selection, each team is compared to other teams. In individual selection, the highest scoring individual is selected.

## 3.1 Observations and Actions

Agents have a sensing range that indicates how many tiles around them they can observe. A sensing range of 0 means an agent can just observe the current tile it is on. A range of 1 means it can observe a square centred on its location that extends 1 tile in each direction (9 tiles total including current tile). A range of 2 means it can observe a square extending 2 tiles in each direction (25 tiles total). And so on. We assume our agent represents a robot

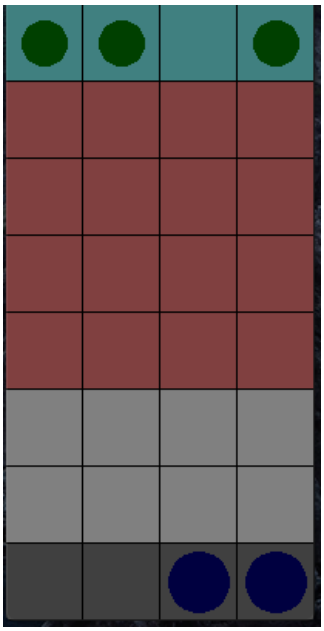

Figure 2: Arena Screenshot

with only local sensing capabilities and use a sensing range of 1, which has the added benefit of reducing computation. For each tile in its sensing range, an agent observes a onehotencoded 4-bit vector. The values it reads denote the following: Blank= 1000, Agent = 0100, Resource = 0010, Wall = 0001. Tiles are read row by row from top left to bottom right. The next part of an agent's observation is a 4-bit vector denoting which part of the arena it is on, similar to a real robot with a ground sensor that can detect the unique colour of each area. The values of this vector can be as follows: Nest = 1000, Cache = 0100, Slope = 0010, Source = 0001 The final part of an agent's observation is 1-bit for resource possession. The values can be as follows: Has resource = 1, Doesn't have resource = 0 The total length of the observation vector is 9x4 + 4 + 1= 41 bits.

An agent can perform 6 possible actions, represented by the following values: Forward = 0, Backward = 1, Left = 2, Right = 3, Pick-up = 4, Drop = 5. We use a recurrent neural network to choose actions based on the observed state. Since many of the positions in the environment will produce the same observation, a recurrent neural network gives the agent a simple form of memory, preventing it from getting "stuck" in infinite state transition loops. The default network has 41 inputs, 1 bias input and 6 recurrent inputs (one

for each of the 6 outputs). There is no hidden layer, just a 6-neuron output layer. This makes for a total of (41+1+6)x6 = 288 weights. The output layer uses a linear activation function.

# 4 Experimental Methods

We use the same setup as Oller et al [1]. We use 3 different architectures: a feedforward neural network (FFNN) with no hidden layers, one with 1 hidden layer composed of 4 hidden units and one with 2 hidden layers also composed of 4 hidden units. We ran experiments with and without a bias neuron and found no noticeable difference. We include the no- bias plots. All networks use tanh activation. For each architecture, we sample 10,000 genomes, where each genome contains weights for two neural networks, one for each agent on the team. For each sampled genome, we do 20 runs of the simulation (i.e. 20 episodes).

# 5 Results

Looking at Figure 4 and Figure 5 in comparison with the Gym benchmarks in Figure 3, we can immediately see that the SlopeForaging task is more difficult than all the standard Gym benchmarks. The mean score is below zero for all architectures and the distribution histogram for the architecture with two hidden layers shows that all episodes indeed receive negative scores. This holds whether or not there is a bias neuron. The same architectures that can successfully solve the Gym benchmarks appear to be insufficient to solve the SlopeForaging problem, indicating that this problem is more difficult than all of Gym's classic control benchmarks.

When we observe the failed controllers to see where they go wrong, we see them carrying out repetitive unproductive behaviours or oscillating between states. For example, an agent at the nest might choose to do a pickup action, despite there being no resource in range. Since there is no resource in range, the state won't change and the agent's observation will be the same at the next time step. Another agent might move forward when in the nest and move backward when in the cache, oscillating between two states until the end of the episode. Using a FFNN means that if an agent makes an observation, they will perform the same action every time they make that observation, meaning it is easy to get stuck like our agents do. A recurrent neural network (RNN)

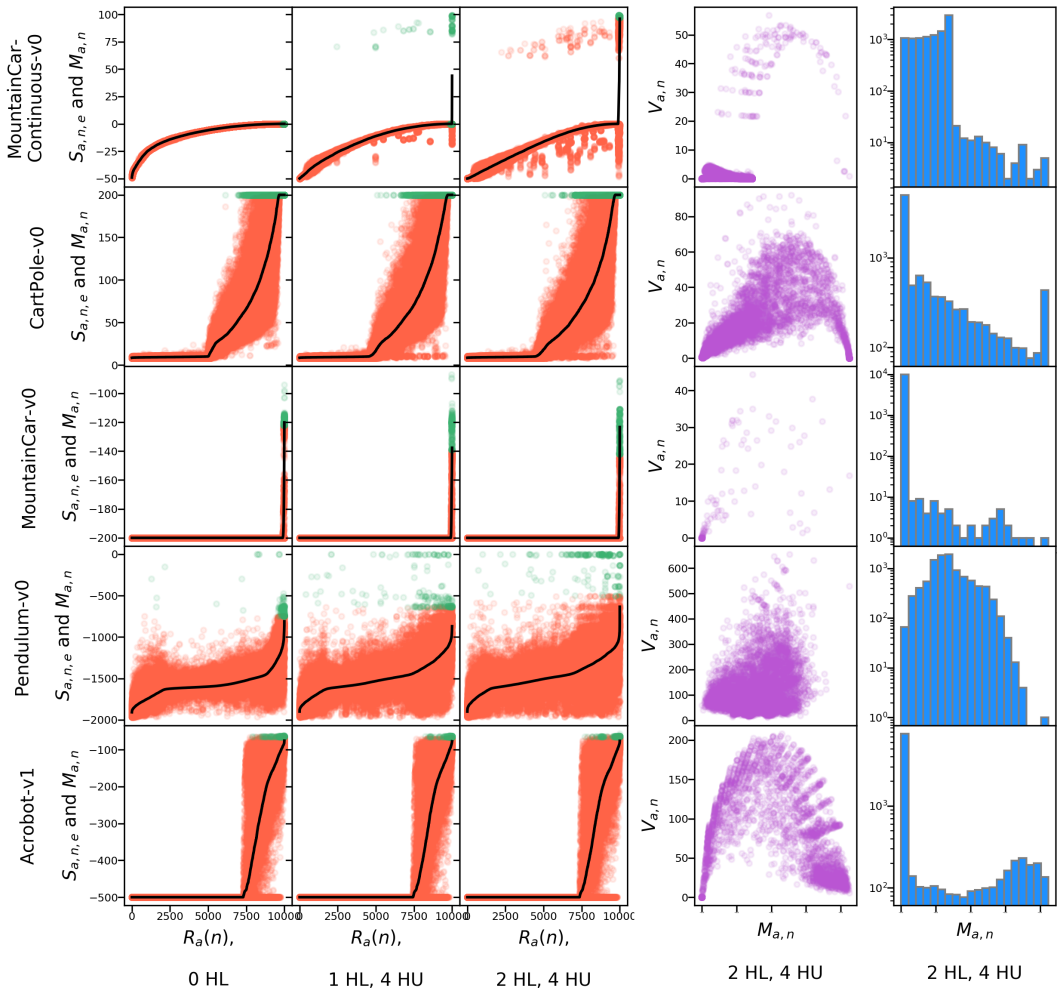


Figure 3: RWG Analysis of OpenAI Gym Benchmarks

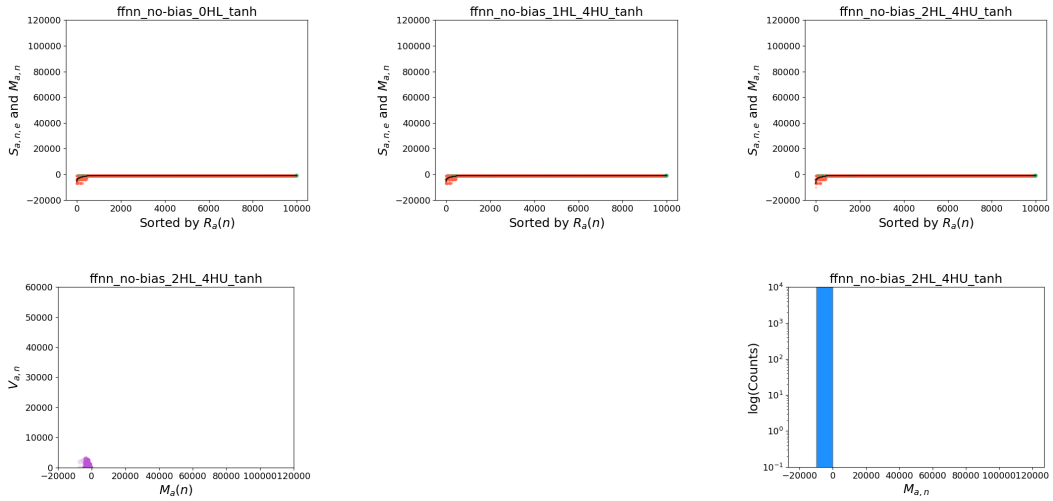


Figure 4: FFNN no bias

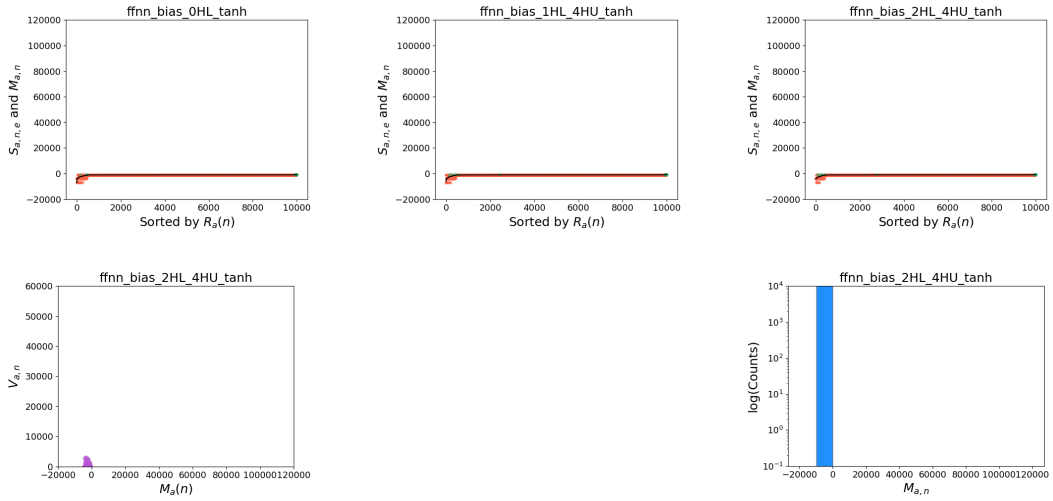


Figure 5: FFNN bias

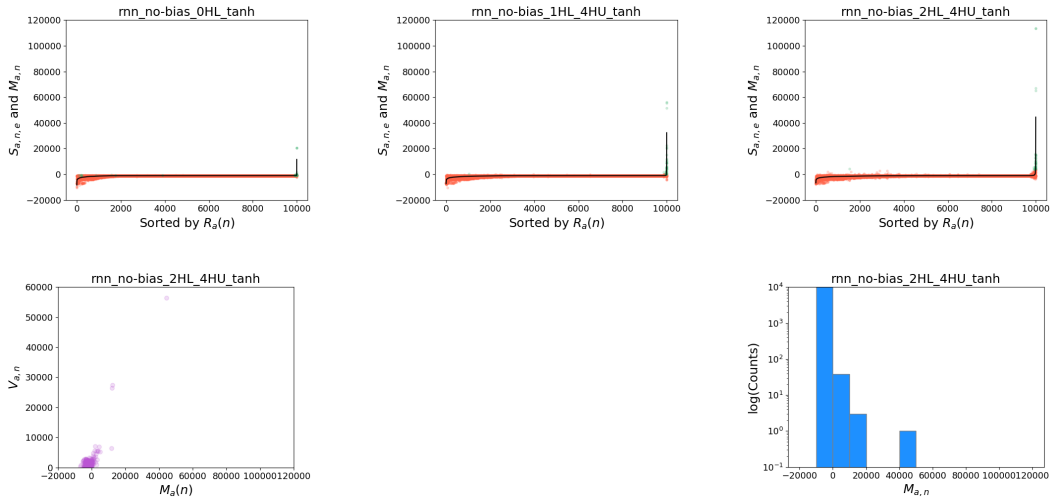


Figure 6: RNN no bias

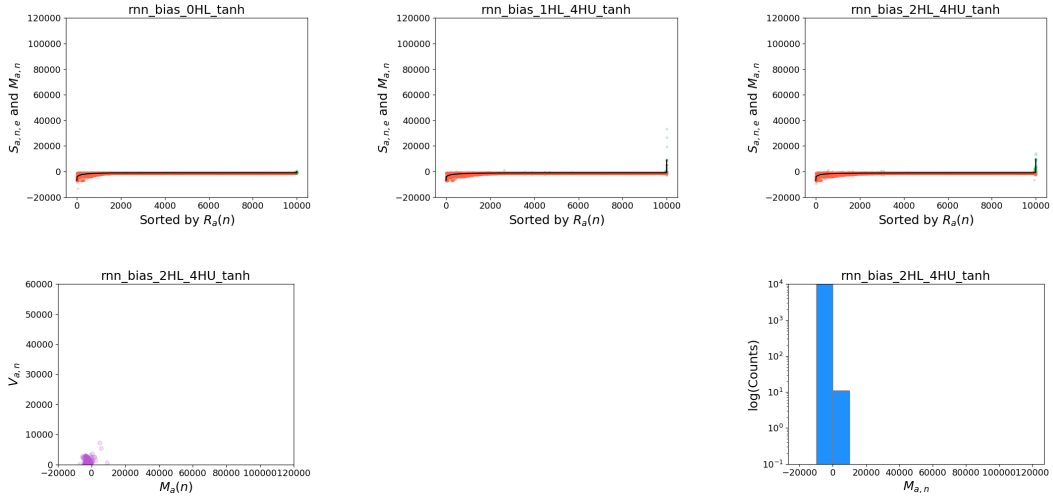


Figure 7: RNN bias

performs an action based on the observation as well as the last action, so if the observation is the same at time $t$ as it was at $t-1$ then an agent can, in principle, do a different action than the one it did at $t-1$ as that one obviously did not change the state. Consider the previously mentioned agent that does a pickup action with no resources nearby and receives the same observation at the following time step. An FFNN-based agent will, by necessity, do that same action again. An RNN-based agent might do a different action because while the observation is the same, the action it did at $t-1$ may not be. Of course, there is a chance the network will have the same output for that observation regardless of what the action at $t-1$ was, but that chance is 100% for an FFNN.

RNN comparison Difficult in a way that is similar to MountainCar and, to an extent, Acrobot Sparse reward Observation/action space (and weights) Partial observability Closeness to optimal

On a fundamental level, we learn from these experiments that the environment, independent of architecture, is challenging in a way that is similar to the MountainCar environment [**?**] investigated by Oller et al [**?**]. The plots are very similar with most solutions being equally poor and very few demonstrating a successful behaviour. We see this in Figure **??** where for both rows (the top row being a network with no bias and the bottom row being a network with bias) the mean curve is very flat with a spike at the end, indicating there are a small number of high performing solutions while the rest score below 0. This can also be seen in the distribution histogram, to the right, where the majority of solutions fall in the leftmost bucket for scores below 0. The variance plot, in the center, shows that variance is higher for higher means suggesting that genomes that score well do not do so consistently across all episodes.

The reason most solutions score very poorly is likely because both the SlopeForaging and MountainCar environments have a sparse reward. In the case of MountainCar, an agent must reach the top of the mountain to receive a score better than -200 and there are no rewards for partially successful intermediate behaviours. An agent that never moves is equivalent to one that stops just short of the goal. In SlopeForaging, an agent that travels up the slope, picks up a resource, carries it to the bottom of the slope and drops it just short of the nest is equivalent to an agent that oscillates back and forth between two tiles. A non-negative score can only be achieved if a resource is fully retrieved and there are no rewards for partially successful intermediate behaviours, so the task can not be learned incrementally.

| Environment Name | Observation Space | Action Space | Observability | Reward Density |
|---|---|---|---|---|
| MountainCarContinuous | Box(2) | Box(1) | Full | Sparse |
| CartPole | Box(4) | Discrete(2) | Full | Dense |
| MountainCar | Box(2) | Discrete(3) | Full | Sparse |
| Pendulum | Box(3) | Box(1) | Full | Dense |
| Acrobot | Box(4) | Discrete(3) | Full | Sparse |
| SlopeForaging | MultiBinary(14) | Discrete(6) | Partial | Sparse |

Table 1: Difficulty of each environment

In fact, the SlopeForaging task is actually more difficult for the following reasons:

- *The observation and action spaces are larger*- An observation consists of 14 binary inputs and there are 6 discrete actions. Even the simplest FFNN would have 14x6=84 weights per agent (168 total) compared to (2x4) + (4x4) + (4x3) = 36 for the most complex controller used by Oller et al to solve MountainCar.

- *The environment is partially observable*- Partial observability suggests that a more complex architecture, such as an RNN, may be required as stated by Oller et al [**?**].

- *The best solution isn't found even after 50,000 samples*- In MountainCar [**?**], the theoretical best score is close to 0 and the solutions generated by Oller et al approach this value, whereas in SlopeForaging, a team of specialists hardcoded with a cooperative strategy scores 157,711 but the best mean score of the solutions found by rwg is 39,879 and the best episode score is 52,642 (for a recurrent network). This is very far from the best known solution. It is even far from the best known generalist strategy, which scores 118,353. And all this is with 5 times more sampled solutions than Oller et al. All this is to say, the task is difficult to solve and, according to Oller et al's recommendations, may require an algorithm with a strong exploratory component.

**Main takeaway: The SlopeForaging task has similar attributes to MountainCar but is more difficult due to observation size, action space, partial observability and the cooperative nature of the task.**

| Environment Name | Minimum Number of Weights |
|---|---|
| MountainCarContinuous | 2 |
| CartPole | 8 |
| MountainCar | 6 |
| Pendulum | 3 |
| Acrobot | 12 |
| SlopeForaging | 168 |

Table 2: Number of weights required for the smallest possible network

| | Generalist | Dropper | Collector |
|---|---|---|---|
| Generalist | 43,649 | | |
| Dropper | | | 120,903 |
| Collector | | | |

Table 3: Hardcoded behaviour scores

# References


[1] Eliseo Ferrante, Ali Emre Turgut, Edgar Duéñez-Guzmán, Marco Dorigo, and Tom Wenseleers. Evolution of self-organized task specialization in robot swarms. *PLoS Computational Biology*, 11(8):e1004273, 2015.

[2] Giovanni Pini, Arne Brutschy, Gianpiero Francesca, Marco Dorigo, and Mauro Birattari. Multi-armed bandit formulation of the task partitioning problem in swarm robotics. In *International Conference on Swarm Intelligence*, pages 109–120. Springer, 2012.

[3] Giovanni Pini, Arne Brutschy, Marco Frison, Andrea Roli, Marco Dorigo, and Mauro Birattari. Task partitioning in swarms of robots: An adaptive method for strategy selection. *Swarm Intelligence*, 5(3-4):283–304, 2011.


# Appendices

## A Experimental Setup

In the following, we explain the current experimental setup for reference.

## A.1 Team Type and Reward Level

During the evolutionary process, it is possible to have four different combinations of team type and reward level that impact evolution. A team can be either homogeneous, with all agents having the same genome, or it can be heterogeneous, with agents having different genomes. In our case, a heterogeneous team has two different genomes, with half the team having one genome and the other half of the team having the other. During evolution, agents can be rewarded as a team or individually, meaning the resource retrieved by an agent can, respectively, count towards the fitness of its teammates or only its own fitness. In the latter case, an individual agent with a high reward and low cost can be selected while its team-mates are discarded. The four combinations are thus: heterogeneous team with team rewards (Het-Team), homogeneous team with team rewards (Hom-Team), heterogeneous team with individual rewards (Het-Ind) and homogeneous team with individual rewards (Hom-Ind).

For the purposes of this analysis, we only use the Het-Team configuration.