Understanding the Challenges of Evolving Multi-Agent Cooperation Using RWG Benchmarking

Mostafa Rizk

August 7, 2020

Contents

1 Executive Summary

We conducted an analysis of the SlopeForaging problem, described in Section A of the appendix, using Oller et al's rwg benchmarking approach [3] to understand the types of challenges it poses and why it is difficult to evolve cooperative behaviour. We focused primarily on the role of the neural network architecture. Our main insights were as follows:

Primary:

- The SlopeForaging task has similar attributes to MountainCar, meaning a highly exploratory algorithm may be more successful.
- SlopeForaging is more difficult than MountainCar due to observation size, action space, partial observability and the cooperative nature of the task.
- An RNN architecture appears to be required for good performance but bias neurons don't make a difference.
- Networks with non-linear activation are not as successful as those with linear activation but get better with additional layers. This is most pronounced for tanh activation.
- Deeper networks skew slightly towards better solutions, though not significantly. Good solutions may be more prevalent for deeper networks but harder to find due to the larger solution space.
- In order for a homogeneous team to exhibit specialisation, it might need to be able to switch between policies and remember how long it has been doing each one. A network with long term memory may be necessary.

Secondary:

- Sliding speed doesn't significantly impact the success of generalist solutions because changing sliding speed doesn't make agents slower or increase the cost of moving up the slope.

- The absence of a cost for moving and carrying (i.e. a battery) does not make it harder to find generalist solutions.
- The implementation contains some minor logical errors with regards to reward allocation and movement costs that should be modified for future experiments.

2 Analysis

We have faced difficulty evolving cooperation in the SlopeForaging multi-agent setting (described in Section A). We conducted an analysis using Oller et al's rwg benchmarking approach [3] to understand the nature of the difficulties posed by this environment. We hope this will provide insight into what changes need to be made to our approach in order to produce the desired behaviour. All experiments used a homogeneous team, rewarded at the team level. Heterogeneous teams and individual rewards are not considered yet.

In each experiment, we sampled 50,000 genomes from a normal distribution with a mean of 0 and unit variance. Each genome was evaluated for 5 episodes. In any instance where hidden layers were used, we used 4 hidden units. We plot a) all the trials in sorted order by score, b) the variance of all trials and c) the distribution on a log scale.

We started our experiments with the simplest possible architecture, a FFNN with no bias, no hidden layers and linear activation (i.e. the identity function). We added up to two hidden layers, then repeated with bias, making for a total of 6 experiments. We then repeated the 6 experiments with a RNN.

For the FFNN, the scores are overwhelmingly poor in comparison to RNNs. For brevity we put the FFNN plots and analysis in the appendix in Section B.1. We also learned, after the experiments, that according to Theorem 1.5.1 in [1], that hidden layers do not add computational power if the activation function is linear. We therefore chose to exclude those plots and repeated these experiments in Section 2.3 with various non-linear activations.

The organisation of this report is as follows: Section 2.1 discusses the fundamental difficulties of the task using Oller et al's analysis of MountainCar [4] as a reference. Section 2.2 talks about why a recurrent network may be

necessary. Section 2.3 explores the impact of non-linear activation functions on performance. Section 2.4 does a preliminary analysis of the benefit of additional layers. Section 2.5 uses additional experiments to examine what may be the minimum requirements of a neural network used in this task. Finally Section 2.6 summarises our findings and proposes our next steps.

2.1 Task Difficulty

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 [4] investigated by Oller et al [3]. The plots are very similar with most solutions being equally poor and very few demonstrating a successful behaviour. We see this in Figure 1 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.

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

- *The observation and action spaces are larger*- An observation consists of 41 binary inputs and there are 6 discrete actions. Even the simplest

FFNN would have 41x6=246 weights 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 [3].

- *The best solution isn't found even after 50,000 samples*- In MountainCar [4], 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.

2.2 Recurrent Networks and Memory

As stated at the beginning of the section, RNNs were found to outperform FFNNs. This makes sense because the environment is partially observable; an agent can only see 1 tile in any direction. A recurrent architecture means an agent can act on its observation while also knowing the action it took in the previous time step. This gives it a form of memory, meaning it is effectively able to act on sequences of observations rather than individual observations, giving it an advantage over an agent with no memory. Presumably, if the agent's sensing radius was expanded to include the entire arena then the FFNN would perform similarly to the RNN. Interestingly, though, we find that adding a bias neuron does not make a difference to the results, with the plots in both the first and second row of Figure 1 being identical.

Main takeaway: Memory appears to be required for good performance but bias doesn't make a difference.

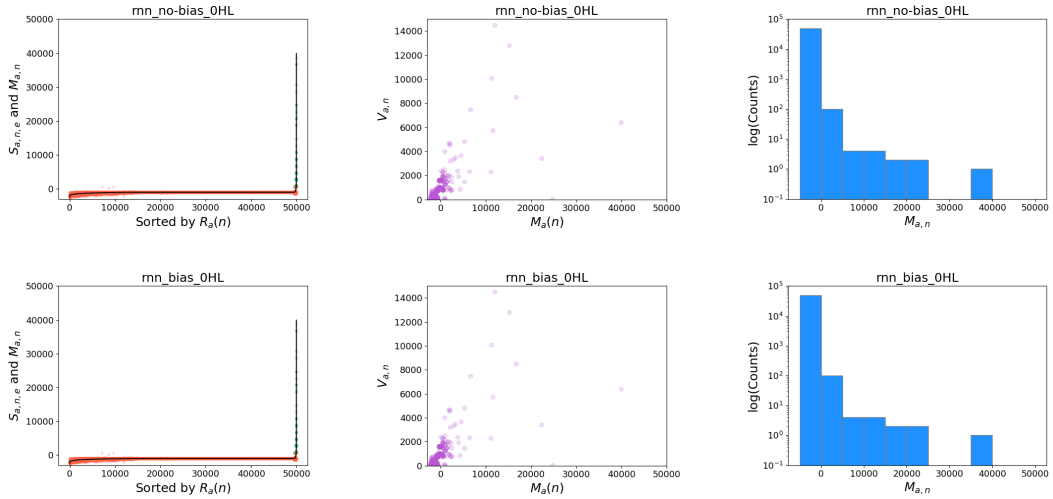

Figure 1: RNN with linear activation

2.3 Non-linear Activation

Following the results from Section 2.2 we conducted the same experiments with non-linear activations, specifically tanh, ReLU and sigmoid functions. We primarily looked at RNNs because previous experiments indicated that an RNN architecture may be necessary for successful behaviour, but we included a FFNN with tanh activation for due diligence in Section C of the appendix. For the three activations, as before, we did six experiments, three with bias and three without, and of those three we tested 0, 1 and 2 hidden layers, with 4 hidden units. Again we found no difference between networks with and without bias so we moved the bias plots to Section D of the appendix.

For all three activations, the highest scores are lower than they are for linear activation. This is surprising as one would expect non-linear activation to provide more representational power to the neural network. Interestingly, though, for tanh and ReLU (Figures 2 and 3 respectively) the instances of high scores (and the top score) increase as hidden layers are added. This suggests that while a network with linear activation is more successful with no hidden layers than its counterparts, the networks with tanh and ReLU activation gain representational power and their performance improves with additional layers. In the histograms in Figure 3, the difference between layers is not as visible as it is for Figure 2 but can be seen in the top scores in Table 1. This suggests that the added gain of each new layer is greater for networks using tanh than

networks using ReLU.

In both cases, though, it is unclear how many layers this trend continues for, or if varying the number of hidden units would accelerate the improvement or if there is another activation function for which the improvement is more pronounced. Additional rwg runs with these deeper architectures can help answer these questions but may also reach a bottleneck as additional layers means more weights, increasing the size of the solution space. Some form of topological search [7] is another way of discovering which architecture is most capable of representing problem solutions.

Sigmoid activation was the exception to the trend, with the distribution of RNN solutions in Figure 4 being similar to those of FFNNs with linear activation in Figure 8. Networks with sigmoid activation appear to not have the representational power required to solve the problem, but it is unclear why. The same holds true for FFNNs with tanh activation (Figure 12). For all networks, though, bias neurons did not make a notable difference.

Main takeaway: Networks with non-linear activation are not as successful as those with linear activation but get better with additional layers. This is most pronounced for tanh activation.

2.4 Adding Layers

As noted in the previous section, networks with tanh activation seem to yield more high performing solutions as the number of hidden layers increases. To investigate this further, we ran additional experiments, using tanh activation, no bias and 3, 4, 5, 6 and 7 hidden layers. We found that while the quality of solutions appeared to increase between 0, 1 and 2 hidden layers, this trend does continue but only very slightly. We can see in the histograms in Figure 5 that the almost all solutions fall into the first 2 buckets, with only 1 falling into the third bucket in the network with 4 hidden layers (in the second row of plots). Higher top scores are thus not found, which we also see in Table 2 where the highest score fluctuates, suggesting that the distribution does not shift significantly, though this does not mean the distribution does not shift at all.

The height of the second bucket in the histogram plots approaches and reaches 10^2 as the number of layers increases. The corresponding plot for the 2 layer setup in Figure 2 is not as high. Further increasing the number of

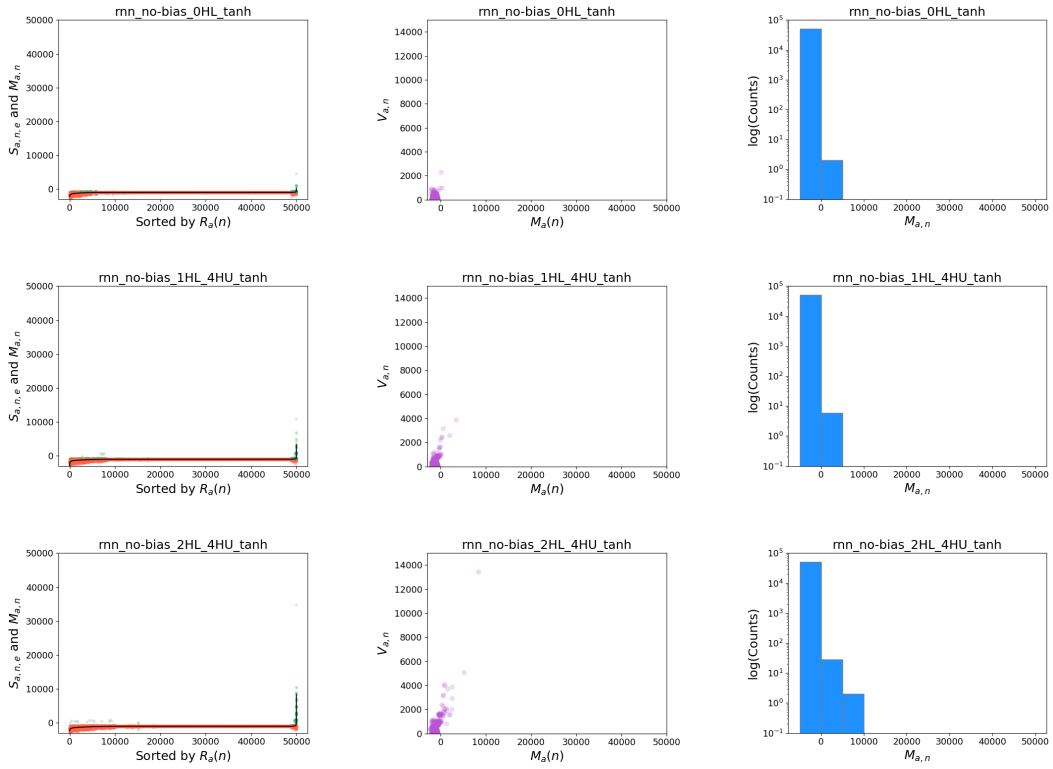

Figure 2: RNN with tanh activation

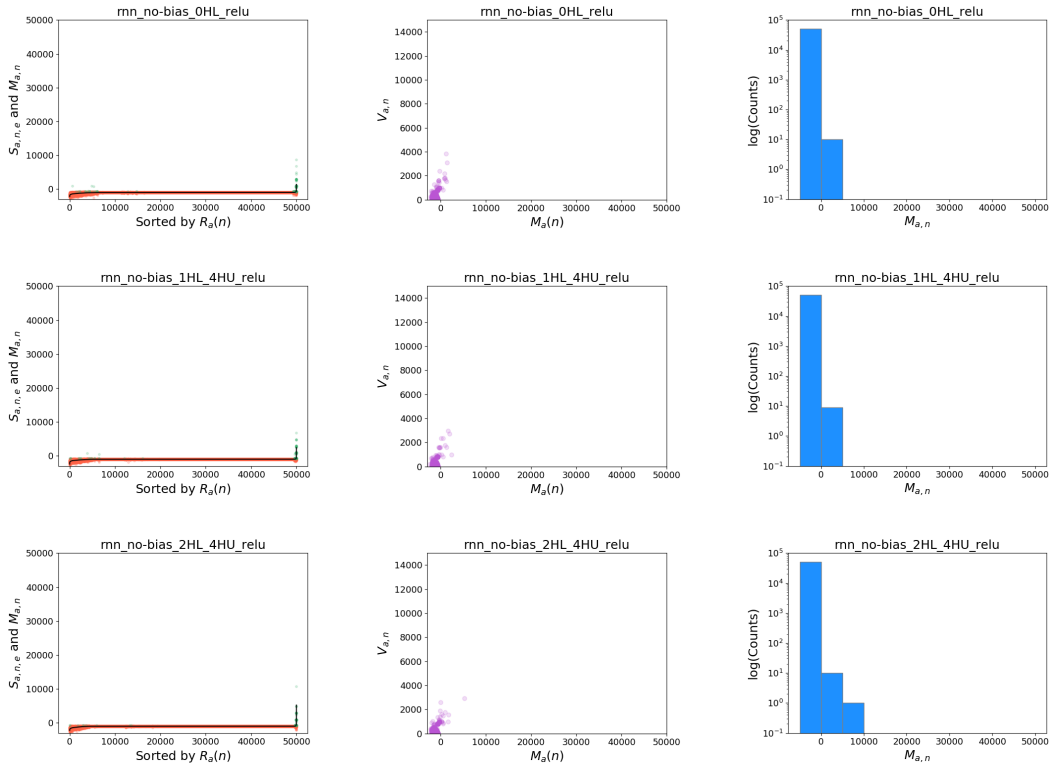

Figure 3: RNN with ReLU activation

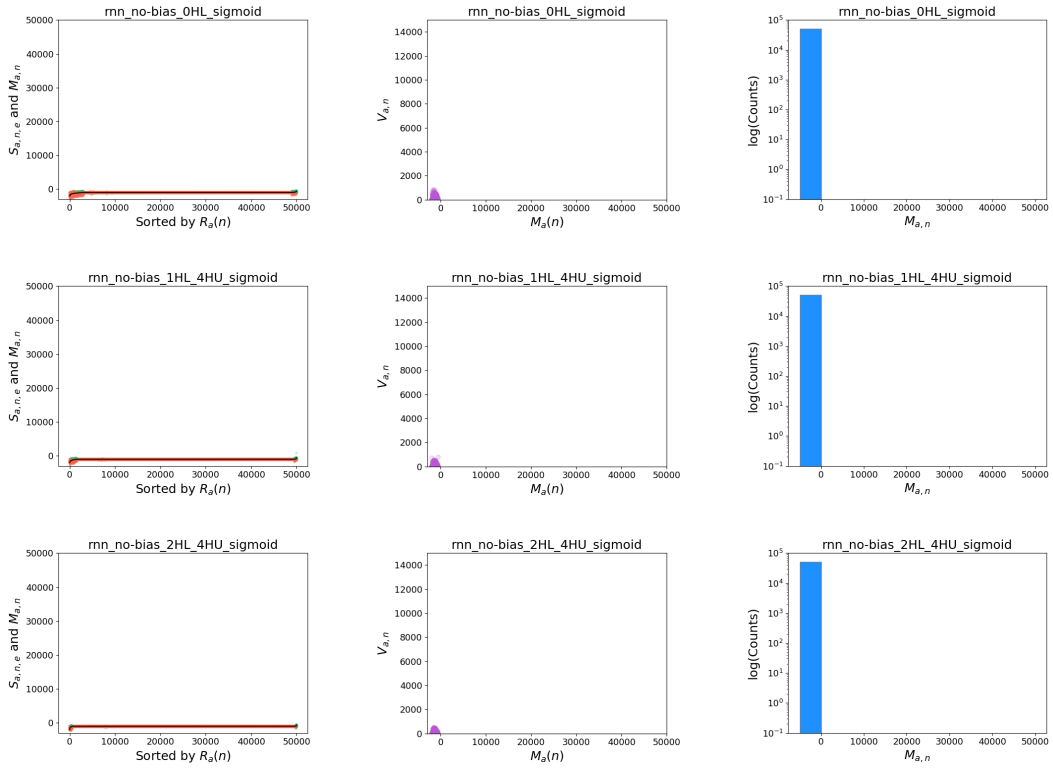

Figure 4: RNN with sigmoid activation

Network Type	Layers	Bias	Activation	Top Score
FFNN	0	N	Linear	-620
FFNN	0	Y	Linear	-620
RNN	0	N	Linear	39879
RNN	0	Y	Linear	39879
RNN	0	N	tanh	297
RNN	1	N	tanh	3361
RNN	2	N	tanh	8376
RNN	0	Y	tanh	297
RNN	1	Y	tanh	3361
RNN	2	Y	tanh	8376
RNN	0	N	ReLU	1375
RNN	1	N	ReLU	2402
RNN	2	N	ReLU	5224
RNN	0	Y	ReLU	1375
RNN	1	Y	ReLU	2402
RNN	2	Y	ReLU	5224
RNN	0	N	sigmoid	-610
RNN	1	N	sigmoid	-603
RNN	2	N	sigmoid	-620
RNN	0	Y	sigmoid	-610
RNN	1	Y	sigmoid	-603
RNN	2	Y	sigmoid	-620
FFNN	0	N	tanh	-620
FFNN	1	N	tanh	-609
FFNN	2	N	tanh	-620
FFNN	0	Y	tanh	-620
FFNN	1	Y	tanh	-609
FFNN	2	Y	tanh	-620

Table 1: Top scores with different architectures

Network Type	Layers	Bias	Activation	Top Score
RNN	0	N	tanh	297
RNN	1	N	tanh	3361
RNN	2	N	tanh	8376
RNN	3	N	tanh	3965
RNN	4	N	tanh	5197
RNN	5	N	tanh	4064
RNN	6	N	tanh	4853
RNN	7	N	tanh	5087

Table 2: Top scores with tanh activation and different numbers of layers

layers does in fact allow us to find more solutions of higher quality, however the number is only on the order of tens. However, it is important to view this in the context of the larger solution space (due to the larger genome size). A network with 2 hidden layers has 260 weights, whereas one with 7 hidden layers has 420. RWG was able to find tens more solutions that fit in the second bucket, but there are also many more possible solutions to sample. It is possible that in the solution space for a 7-hidden layer network there is a larger number of successful architectures but they are harder to find by sampling. The RWG approach may be limited in its ability to analyse task difficulty for applications with larger neural networks. Further analysis may require alternative techniques.

Main takeaway: Deeper networks may be more adept at solving the task but it is difficult to tell because the solution space is much larger.

2.5 Minimum Architecture Complexity

A question that arises in performing all the above analysis of various architectures is "What is the minimal architecture needed to represent a successful solution to this problem?". Our results so far seem to indicate that more network complexity may be required. RNNs are more successful than FFNNs, indicating the need for memory and multiple hidden layers with non-linear activation show a trend towards better solutions. Though none of these have been able to represent a specialist team yet, which raises the question of how much is needed? Insight into this question can be gained by looking at a be-

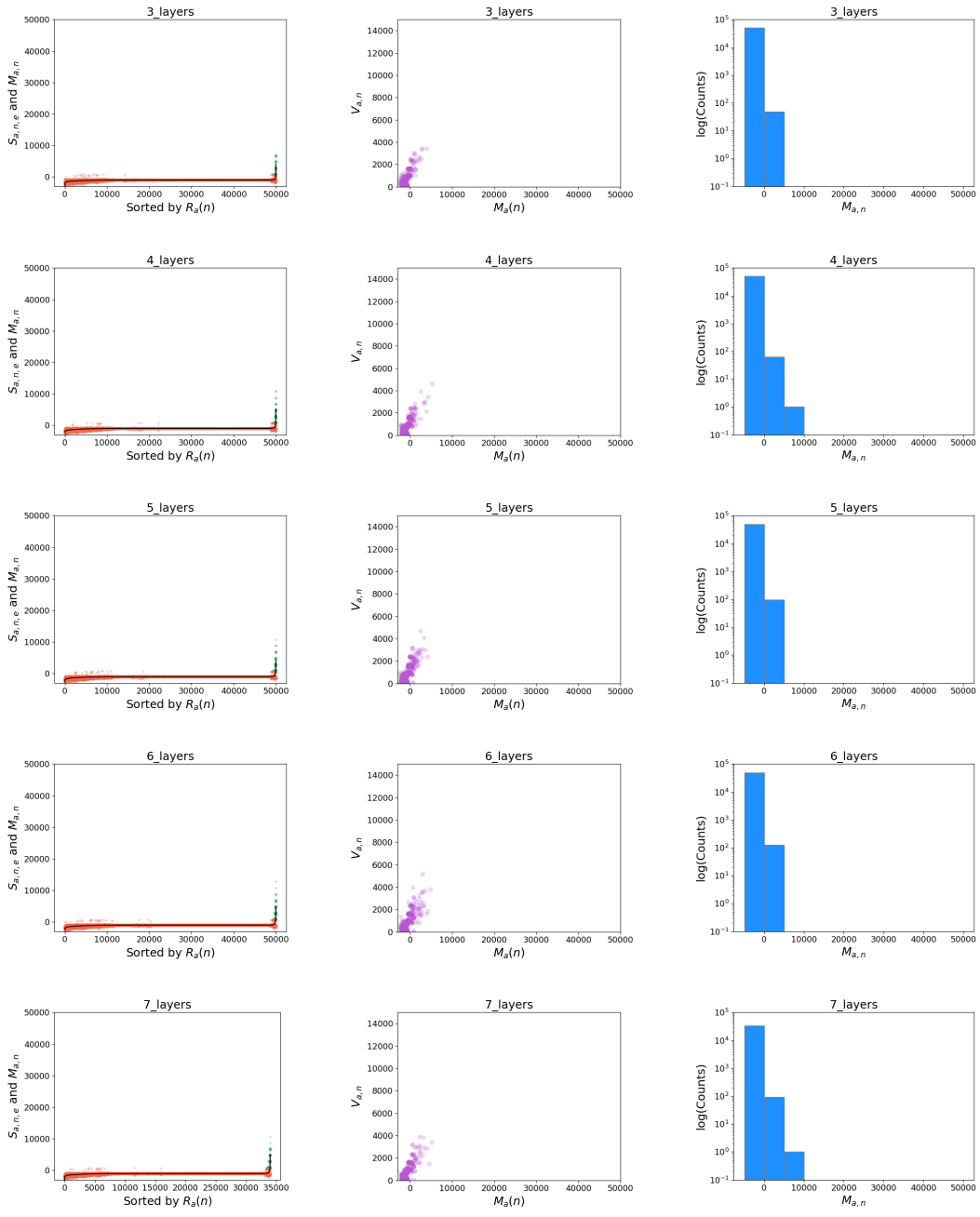

Figure 5: RNN with tanh activation and multiple layers

haviour we call a "lazy generalist".

A lazy generalist is a strategy where an agent goes up the slope and drops resources for a while before switching activities, going to the cache and moving the resources it dropped into the nest. This agent is "lazy" because it does not want to pay the cost of carrying a resource down the slope. By dropping the resource, it saves itself time and energy. In the case of our setup, it loses less points by letting resources slide. If two agents perform this behaviour simultaneously, they exhibit what looks like specialisation. Both agents begin as droppers but at some point, one of them drops resources while the other collects what was dropped, effectively collecting the resources the other agent intended to collect itself. We confirm this by hardcoding a lazy generalist behaviour and observing two of them working in the same environment.

Intuitively, in order for a homogeneous team to perform specialisation, each individual agent must be capable of both dropping and collecting and switch between the two behaviours. In order for specialisation to emerge, evolution must find the lazy generalist behaviour and the neural network must be capable of representing it. RWG does not appear to find this behaviour, which could suggest it is rare. However, further optimising with CMA does not find it either, which suggests that the neural networks we are using are not able to represent it.

In order for a neural network to represent the lazy generalist behaviour, it needs to be able to observe the same state and do two different actions depending on whether it is a dropper or a collector. This could mean that it randomly chooses whether to do one or the other based on a certain probability, or it could mean that it remembers what it has been doing for several time steps. For example, an agent that is empty-handed at the top of the slope should know to go forward (to collect more resources) unless it has been in that region and dropping for several time steps, then it should know to go backward towards the nest to collect. This suggests the need for a network capable of 'remembering' several time steps i.e. that is able to recognise sequences over many time steps. A standard RNN may not be sufficient for this. A more sophisticated recurrent network such as an LSTM may be needed or an RNN with several layers.

Main takeaway: In order for a homogeneous team to exhibit specialisation, it needs to be able to switch between policies and remember how long it has been doing each one. A network with

long term memory may be necessary.

2.6 Conclusion and Next Steps

SlopeForaging is a non-trivial task to solve and the above analysis confirms it at several junctions. Our challenges in finding a solution may be influenced by the choice of algorithm but the results seem to point towards the need for a more expressive architecture. Throughout the study, as we have complexified the network by adding recurrence, non-linear activation and multiple hidden layers, we have seen signs that these features give the network more representational power. Based on these findings we believe it is highly likely that a more sophisticated network is needed to solve this problem. We believe that the next logical course of action is to investigate the techniques available for architecture search such as those in [7] and its successors. It may be worthwhile to first try an exploratory search algorithm and if that algorithm fails to find the desired solution, that would further support the hypothesis that a more expressive architecture is needed.

References

[1] Charu C Aggarwal. *Neural networks and deep learning*. Springer, 2018.

[2] 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.

[3] Declan Oller, Tobias Glasmachers, and Giuseppe Cuccu. Analyzing reinforcement learning benchmarks with random weight guessing. *arXiv preprint arXiv:2004.07707*, 2020.

[4] OpenAI. Mountaincar description. `https://github.com/openai/gym/wiki/MountainCar-v0`.

[5] 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.

[6] 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.

[7] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Appendices

A Experimental Setup

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

A.1 The Foraging Task

We use a foraging task as the testbed for the evolution of specialisation. The task is modelled off the foraging behaviour of the Atta Leafcutter ant, as in Ferrante et al [2] and Pini et al [5, 6]. 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 [2], 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 [5, 6]. 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 6 and Figure 7. 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

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

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.

A.2 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 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

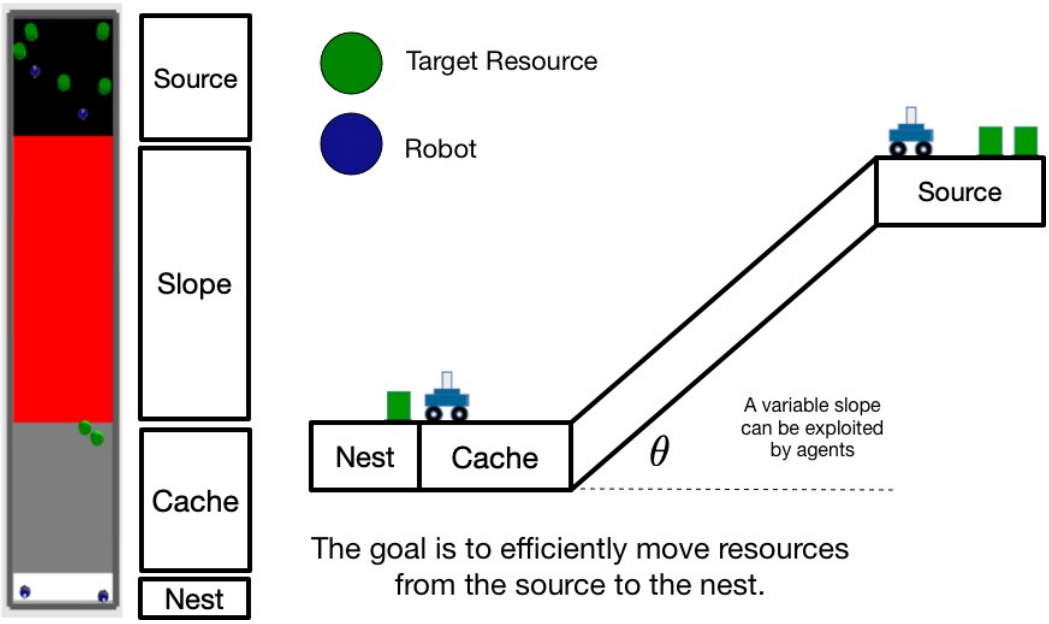

Figure 6: Arena Layout

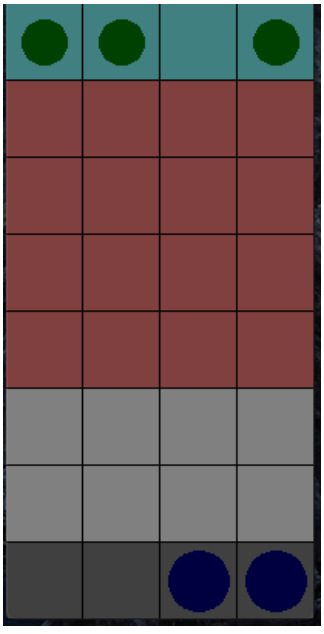

Figure 7: Arena Screenshot

= 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.

A.3 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 Hom-Team configuration.

B Additional Experiments

In the following, we show experimental results that were not included in the main report.

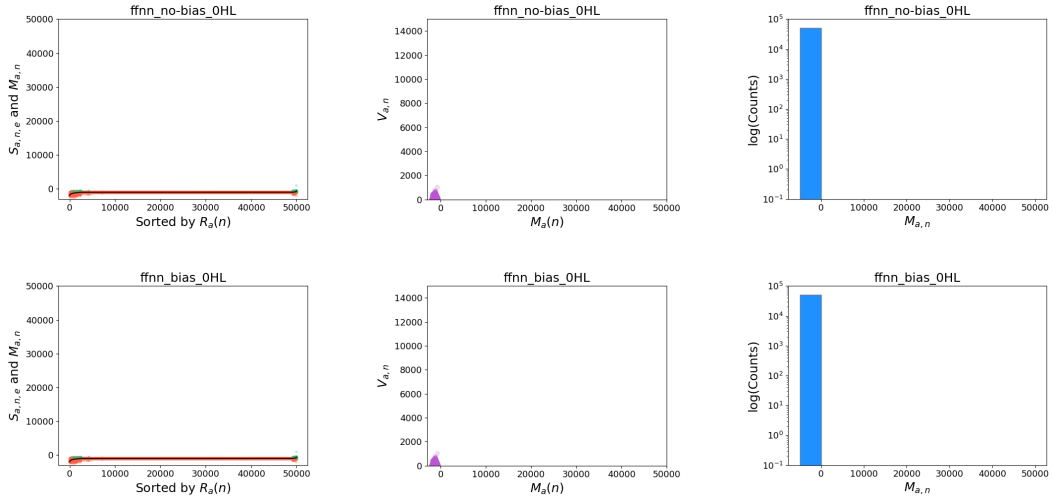

Figure 8: FFNN with linear activation

B.1 FFNN

The scores for FFNNs all fall into the leftmost bucket of the distribution plot as we see in Figure 8. We also see that the mean curve is very flat, with very little variance. FFNN solutions are consistently poor, indicating, as mentioned in the main text of the report, that RNNs might be necessary to solve the problem.

B.2 Modifying Environmental Factors

The initial experiments gave us an understanding of why the task is difficult, but they used a specific configuration of the environment. It was important to observe how changing these parameters changed the task difficulty. We varied a) the sliding speed, an approximation of slope angle for a 2D discrete environment, b) the presence of a battery i.e. the cost associated with performing actions, and c) the multi-agent nature.

B.2.1 Sliding Speed

By default, the environment has sliding speed = 4. This means that when a resource is dropped, it slides 4 tiles per time step. This simulates a slope. The presence of a slope is the environmental factor that makes specialised teams outperform generalist teams. In an environment with no slope, a specialist

strategy adds no benefit over a generalist one and may even be worse. Conversely, in an environment with a steeper slope, a specialist strategy could provide an even larger advantage because the slope can get resources to the bottom even faster. We ran experiments with varying sliding speed to observe how it impacts the solutions found. For brevity, we only use one architecture: an RNN with bias, no hidden layers and linear activation.

From the plots in Figure 9 we find that the sliding speed doesn't make much of a difference to the quality of solutions. It is important to note that while sliding speed is an approximation of slope angle it is not a direct analogue. Slope angle affects the speed at which a resource slides down the slope and the speed at which an agent goes up, but sliding speed only affects the speed of the resource. Since the only successful solutions rwg finds are generalists, the speed at which resources slide doesn't factor into their scores because they never drop resources. Any solutions that do drop resources, don't succeed at the task, so the speed at which those resources slid did not affect their score.

We also spotted a minor logical error in the code that causes the cost of moving up/down the slope to be the same regardless of the sliding speed. This does not change the stag hunt dynamic of the environment, however it would more accurately represent a slope if the cost of moving changed with the sliding speed.

Main takeaway: Sliding speed doesn't significantly impact the success of generalist solutions because changing sliding speed doesn't make agents slower or increase the cost of moving up the slope.

B.2.2 No "battery"

In our default setup, agents pay a cost for moving in any direction. The cost is higher going up the slope and lower coming down the slope. It is also higher if carrying a resource. This simulates the presence of a battery. An agent is thus, penalised for wasting time and for carrying things on the slope, making specialisation more advantageous and more accurately representing a real robot. A battery is not used by Ferrante et al. [2] and simply counting the number of resources is sufficient to see specialisation emerge, however, Ferrante et al also had the benefit of 3D physics to create delays in robot and resource movement. It is important to understand how the penalisation of movement impacts the scores of solutions.

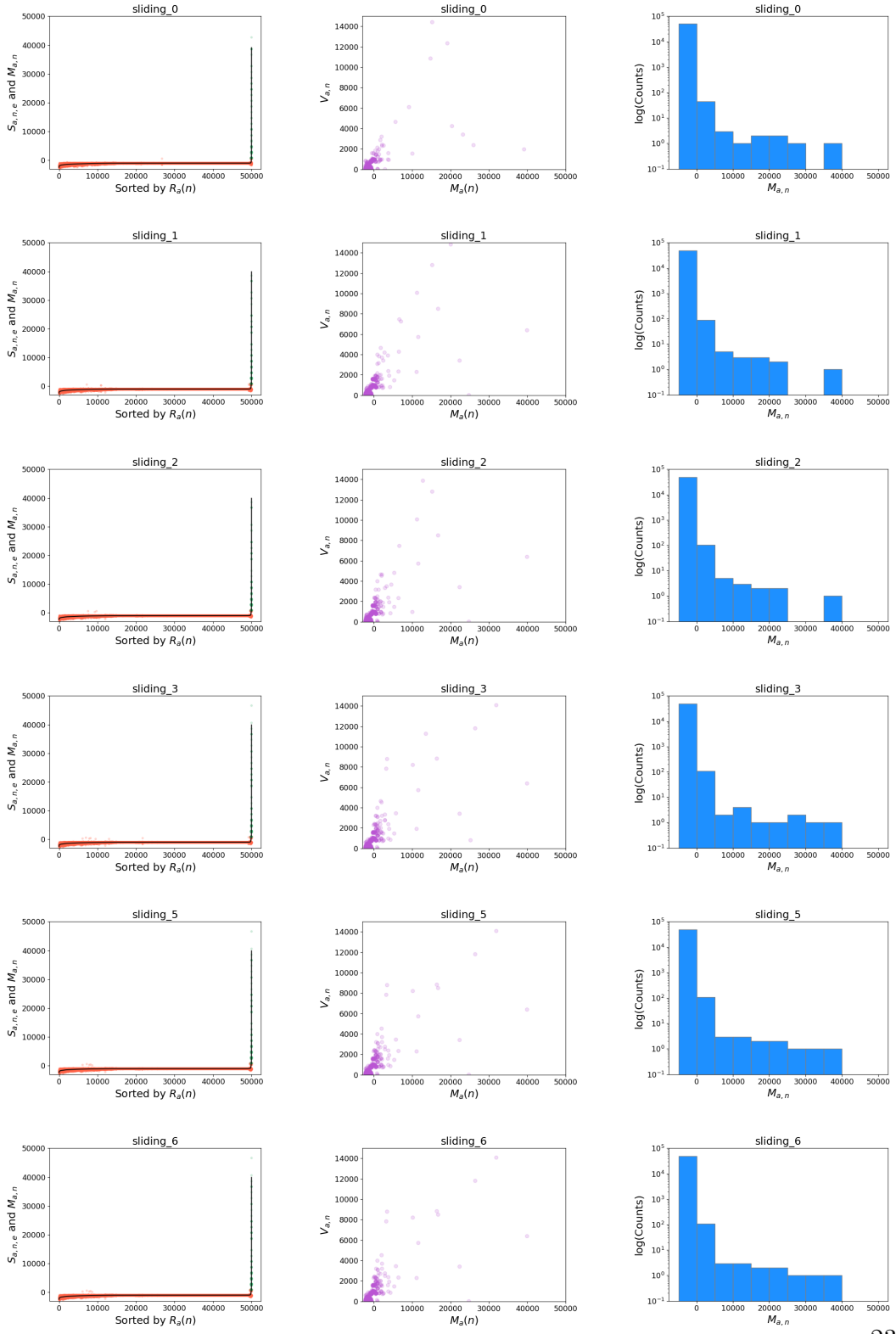

Figure 9: Varying sliding speed

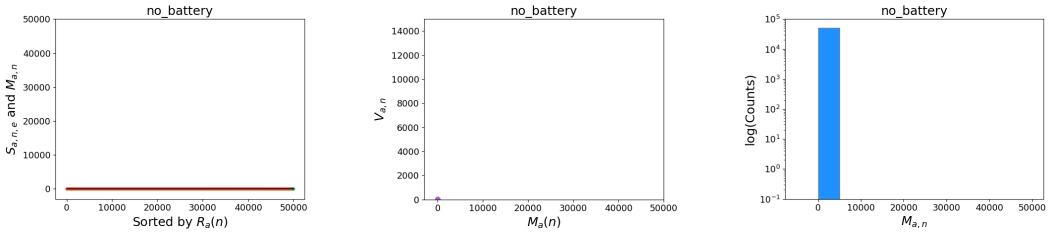

Figure 10: No battery

In the no battery experiments, movement has no cost and agents are rewarded one point for retrieving a resource rather than 1000. We scale down the mean and variance axes in our plots by 1000 to account for this difference. Once again we see in Figure 10 that the majority of solutions are poor, with most retrieving 0 resources. And again we see that variance is higher for the better solutions indicating they are not always successful. There would appear to be more solutions that fall into larger buckets but this is likely only because the scale of the y-axis is smaller, so it is easier for those buckets to be filled. The experiment may need to be repeated with a 1000 score for each retrieved resource.

Overall, the absence of a "battery" or cost to movement does not appear to make any difference to the difficulty of the task which makes sense as the main source of difficulty is the sparseness of the reward. We also know from hardcoded controllers that in the absence of a battery, a specialist team still outperforms a generalist team (scoring 158.4 as opposed to 114.4). The presence or absence of movement and carrying costs does not appear to affect task difficulty.

Main takeaway: The presence or absence of a "battery" does not appear to affect task difficulty.

B.2.3 Single Agent

In the single agent experiment, we again used an RNN with bias, no hidden layers and linear activation. Here the results are significantly lower, with the top score reaching only 6153 (Table 3), compared to a 2-agent team which achieves a top score of 39,082. This can be explained by the reward function. When an agent retrieves a resource, it receives +1000 and its teammates do as well. The team score is the sum of all individual scores. This mean that

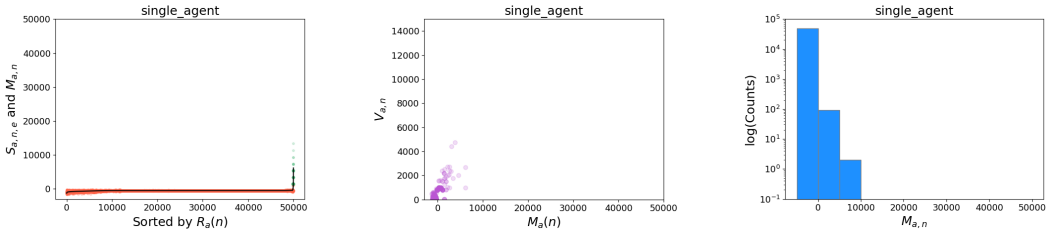

Figure 11: Single agent

Environment	Top Score
Sliding speed = 0	39082
Sliding speed = 1	39879
Sliding speed = 2	39879
Sliding speed = 3	39879
Sliding speed = 4	39879
Sliding speed = 5	39879
Sliding speed = 6	39879
No battery	41.2
Single agent	6153

Table 3: Top scores in different environments

if there are two agents, one retrieved resource counts as 2000 and with two agents acting on the environment, they are likely to gather twice as many resources for four times the score. This is a logical error that was spotted as a result of these experiments.

In terms of task difficulty, having only one agent does not impact the difficulty. Figure 11 shows similar patterns to previous plots, but this may be a factor if there are enough agents in the arena to experience collisions. Sequential position updates are one way of solving that problem.

Main takeaway: Rewards should be divided not multiplied

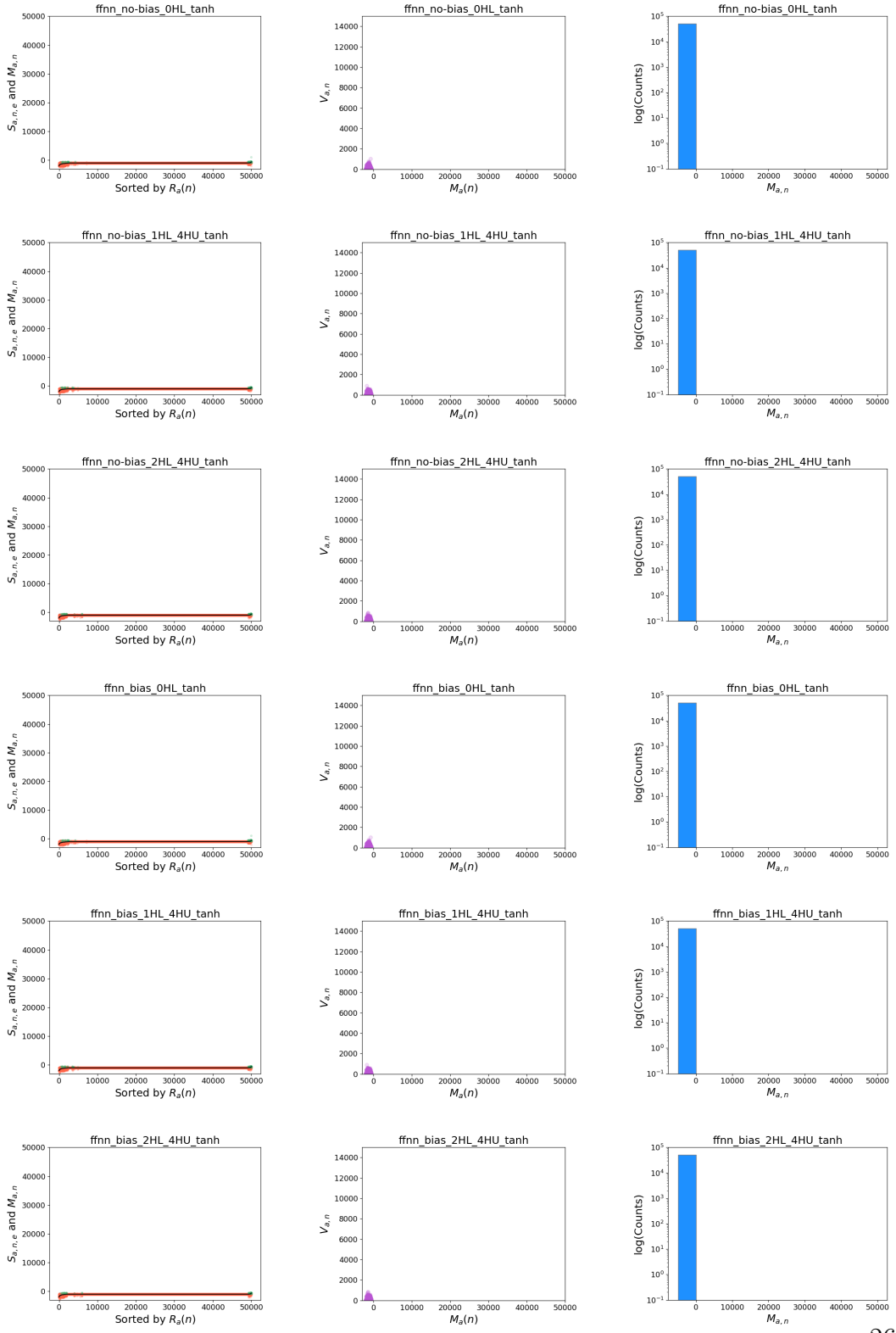

Figure 12: FFNN with tanh activation

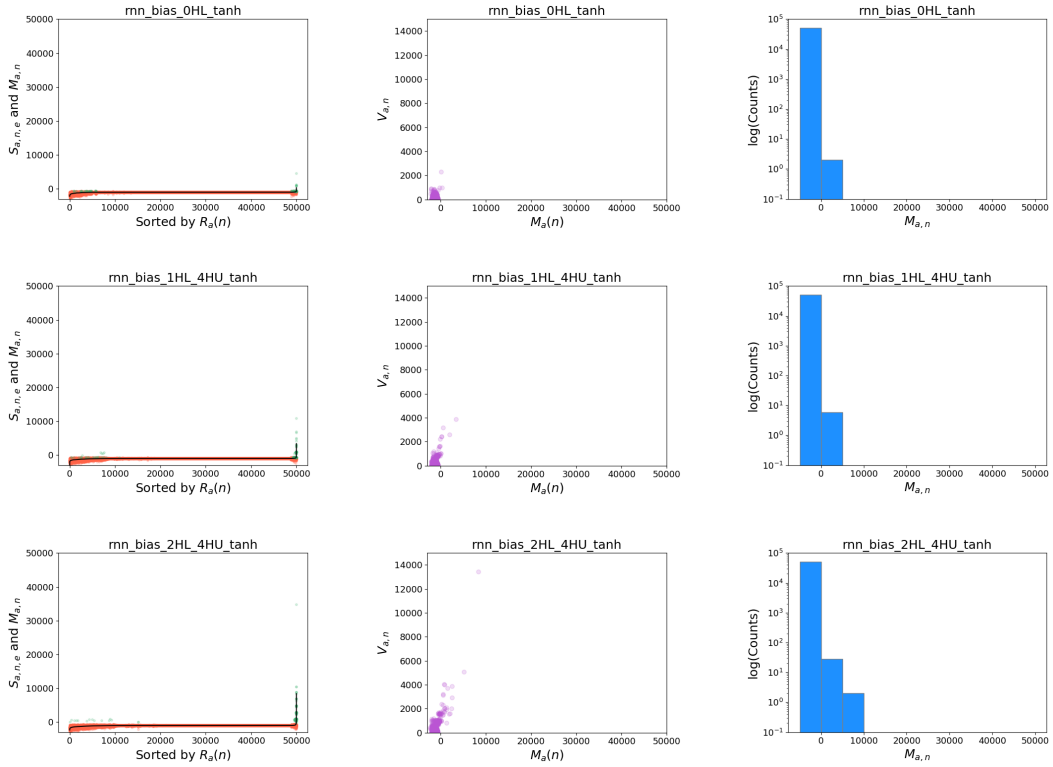

Figure 13: RNN with tanh activation and bias

C FFNN w/ Non-linear Activation

D RNN with Non-linear Activation and Bias

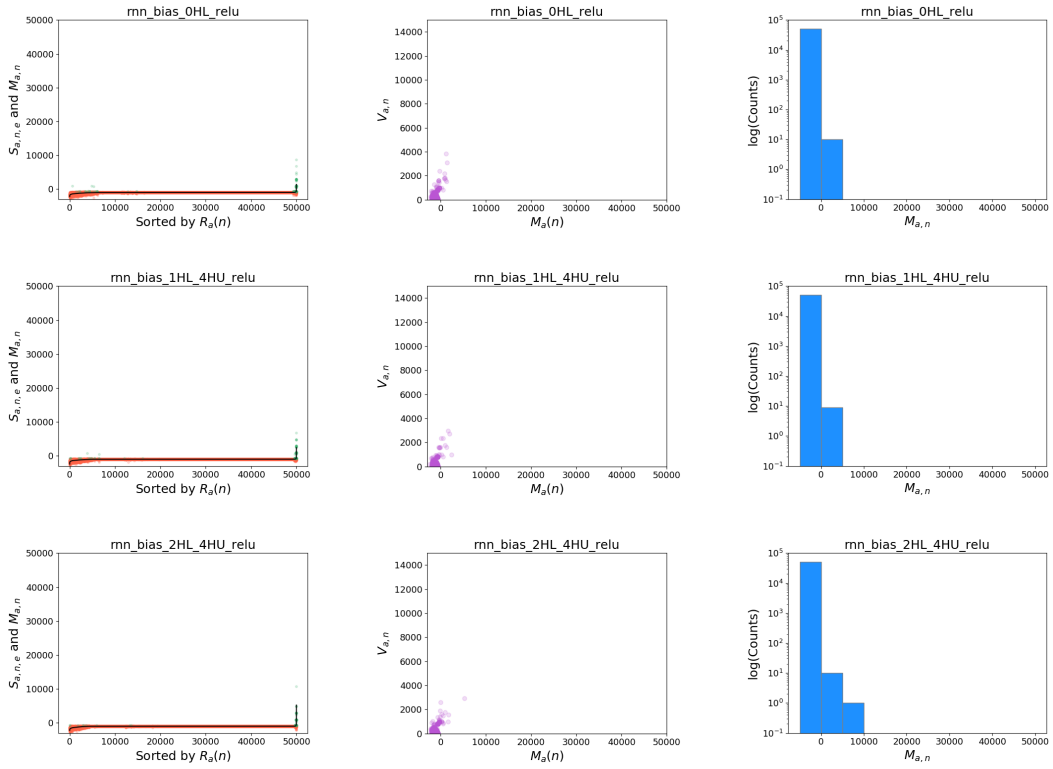

Figure 14: RNN with ReLU activation and bias

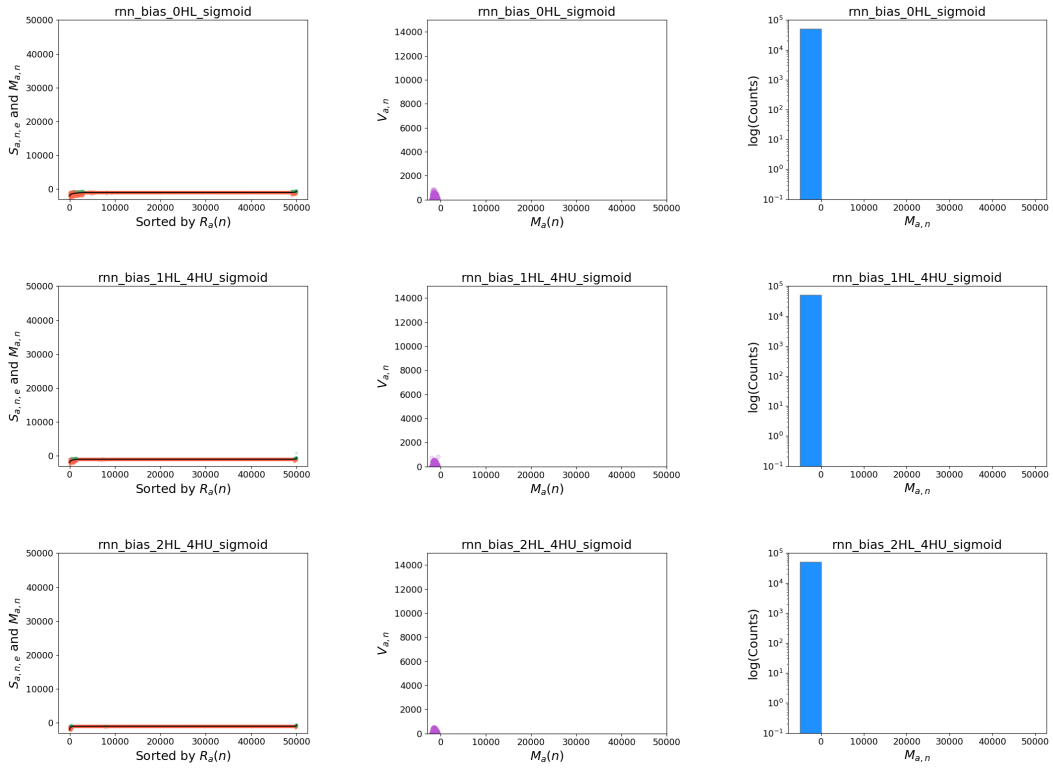

Figure 15: RNN with sigmoid activation and bias