MAGNets: Micro-Architectured Group Neural Networks (Appendix)

Appendices

A Program Policy Generation

This section discusses how the policy derived through MAGNets can be converted into interpretable programs. Many recent works focus on converting neural policy into interpretable programs or ML models. In (Verma et al. 2018), the author proposes a framework that uses Neurally Directed Program Search (NDPS) to synthesize a program that takes the state as input and outputs actions that give the maximum reward. Whereas (Qiu and Zhu 2022) proposes a method to find a program policy without any oracle or expert policy. In comparison, our MAGNets policy can also be converted into terms of a program policy; however, our framework doesn't require specifying the program grammar. The authors in (Bastani, Pu, and Solar-Lezama 2018) propose the VIPER algorithm to construct an equivalent decision tree policy, and (Ghosh et al. 2021) uses an oracle to find an equivalent non-linear decision tree (NLDT) policy. In (Milani et al. 2023), the authors propose the MAVIPER algorithm, which is a modification of the VIPER (Bastani, Pu, and Solar-Lezama 2018) algorithm for multi-agent RL environments.

A.1 Interpretable/Programmatic Representation of MAGNet Policy

The neural networks used in our MAGNets framework are small and use simple feed-forward architecture. Therefore, each MAGNet can be represented using a conventional program block. We leverage that to expand the rules list into an equivalent program policy. We present the equivalent program policy representation of Cartpole and Inverted Pendulum's MAGNet policy. We use rounded values up to two decimal places for all the real values for readability.

A.1.1 Programmatic Representation of Cartpole MAGNet Policy

```
def get_action(state):
    ip0 = state[0] #Cart Position
    ip1 = state[1] #Cart Velocity
    ip2 = state[2] #Pole Angle
    ip3 = state[3] #Pole Angular Velocity
    action = 0 #0: Push cart to left, 1: Push cart to right

    #MAGNet_1
    x0 = (ip0*0.01)-(ip1*0.73)-(ip2*2)-(ip3 *1.49)+0.26
    x1 = -(ip0*0.05)+(ip1*0.26)+(ip2*0.14)+(ip3*0.2)-0.05
    x0 = (x0>0) ? x0: 0
    x1 = (x1>0) ? x1: 0
    x2 = -(x0 * 0.21) - (x1 * 0.14) - 0.29
    x3 = -(x0 * 2.15) - (x1 * 0.33) + 1.14
    x2 = (x2>0) ? x2: 0
    x3 = (x3>0) ? x3: 0
    x4 = (x2 * 0.28) + (x3 * 2.0) + 0.01
    MGN1 = (x4>0) ? x4: 0

    #MAGNet_2
    x5 = -(ip0*0.05)+(ip1*0.14)-(ip2*0.54)-(ip3*0.24)-0.15
    x6 = -(ip0*0.08)+(ip1*0.73)+(ip2*1.24)+(ip3*2.07)+0.61
    x5 = (x5>0) ? x5: 0
    x6 = (x6>0) ? x6: 0
    x7 = -(x5 * 0.21) - (x6 * 0.14) - 0.29
    x8 = -(x5 * 0.6) - (x6 * 2.09) + 2.01
    x7 = (x6>0) ? x6: 0
    x8 = (x7>0) ? x7: 0
    x9 = (x7 * 0.28) + (x8 * 2.79) + 0.02
    MGN2 = (x9>0) ? x9: 0

    x10 = -(MGN1 * 0.38) + (MGN2 * 0.48) - 0.27
    x11 = (MGN1 * 0.85) - (MGN2 * 0.58) + 0.65

    if (x10>=x11):
        action = 0
    else:
        action = 1

    return action
```

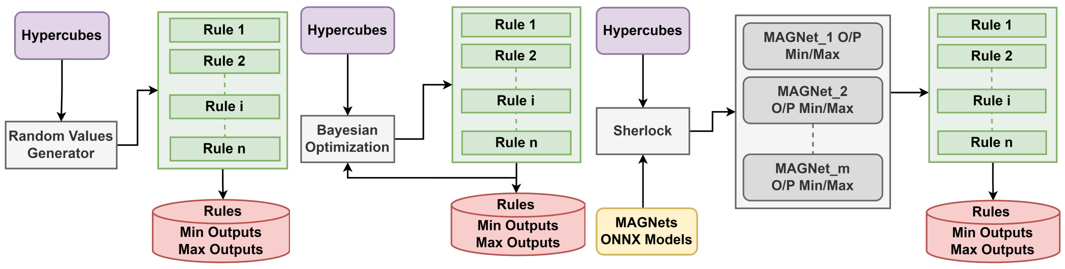
A.1.2 Programmatic Representation of Inverted Pendulum MAGNet Policy

```
def get_action(state):
    ip0 = state[0] #Cart Position
    ip1 = state[1] #Cart Velocity
    ip2 = state[2] #Pole Angle
    ip3 = state[3] #Pole Angular Velocity
    action = 0 #Force applied on the cart (-3 to 3)

    #MAGNet_0
    x0 = -(ip0*0.11)+(ip1*0.1)-(ip2*0.5)-(ip3*0.26)-0.15
    x1 = (ip0*1.1)+(ip1*2.76)+(ip2*0.24)+(ip3*0.96)+0.91
    x0 = (x0>0) ? x0: 0
    x1 = (x1>0) ? x1: 0
    x2 = -(x0 * 0.21) - (x1 * 0.14) - 0.29
    x3 = -(x0 * 0.59) - (x1 * 1.47) + 2.41
    x2 = (x2>0) ? x2: 0
    x3 = (x3>0) ? x3: 0
    x4 = (x2 * 0.28) + (x3 * 2.97) + 0.01
    MGN1 = (x4>0) ? x4: 0

    #MAGNet_2
    x5 = -(ip0*0.37)-(ip1*2.64)-(ip2*0.37)-(ip3*0.79)+1.03
    x6 = -(ip0*0.34)-(ip1*0.19)+(ip2*0.22)+(ip3*0.16)-0.12
    x5 = (x5>0) ? x5: 0
    x6 = (x6>0) ? x6: 0
    x7 = -(x5 * 0.21) - (x6 * 0.14) - 0.29
    x8 = -(x5 * 1.86) - (x6 * 0.28) + 2.29
    x7 = (x7>0) ? x7: 0
    x8 = (x8>0) ? x8: 0
    x9 = (x7 * 0.28) + (x8 * 2.83) + 0.4
    MGN3 = (x9>0) ? x9: 0

    MGN2 = 0 #MGN2 is a dead/zero neuron
    MGN4 = 0 #MGN4 is a dead/zero neuron

    x10 = -(MGN1*0.38) + MGN2 + (MGN3*0.54) + MGN4 - 0.39

    action = x10

    return action
```

B MAGNets Verification/Testing

The verification strategy employed using Sherlock is outlined in the main text. This section outlines two other testing strategies for MAGNets and provides additional results for Sherlock verification in different environments.

B.1 Verification/Testing Strategies



(a) Testing with Random I/P (b) Testing with BayesOpt (c) Verification with Sherlock

Figure 1: Magnets Testing/Verification Strategies

B.1.1 Testing with Random Input

Here a random value generator takes a hypercube as input and generates y number of random environment states within that hypercube, where y is a hyperparameter. After that, those randomly generated states are fed into the Rule List representation of the NNs. Each rule's maximum and minimum outputs are then captured and stored. Figure 1a depicts the overall process. Now, as we understand, with the larger value of y, the maximum and minimum output range will approach the rules' actual maximum and minimum values within that hypercube. However, in this way, finding the rules' actual maximum and minimum values within that hypercube can not be guaranteed. Also, blindly sampling the environment states might not be sample efficient to approximate the rules' actual maximum and minimum values within that hypercube. Therefore, we also use Bayes Optimization based technique which is discussed below.

B.1.2 Testing with BayesOpt

As we have discussed in the previous section, we use the Bayesian Optimization based sampling technique to achieve better sample efficiency than the randomized testing technique. The schematic representation of the Bayesian Optimization based testing technique is shown in Figure 1b. The Bayesian Optimization is a feedback-based sampling technique, as we see in Figure 1b where the output of the Rule List is given to the Bayesian Optimization as input. Bayesian Optimization uses this information to calculate the next environment state with a higher potential of being closer to the actual maximum or minimum output of the Rule List. In general, the Bayesian Optimization algorithm is designed to find the maximum output value of a black box function. However, we use it for finding both the maximum and minimum range of rules by negating the output while finding the minimum. The major disadvantage of this approach is that we have to run it twice, once to find the maximum and once to find the minimum, for each rule separately. Other limitations of this approach are it does not scale for a larger number of variables, its performance greatly depends on the choice of the kernel, and it does not work with variables having discrete domains.

B.1.3 Action Selection

By construction, the rules in Rule List output the logit value of respective actions (if the action space is discrete) or the mean value of that respective action (if the action space is continuous) for a given input. The various testing and verification strategies highlighted in our main draft as well as the appendix, helps us determine the output ranges for each rule given an input hypercube. These output ranges need to determine usable actions for the input region specified in the given hypercube.

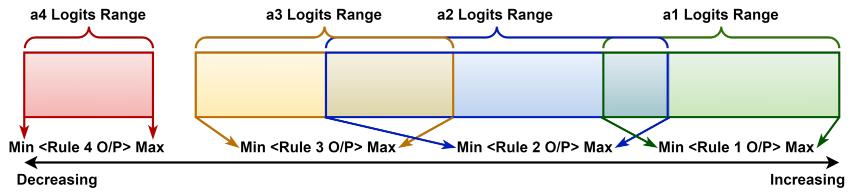
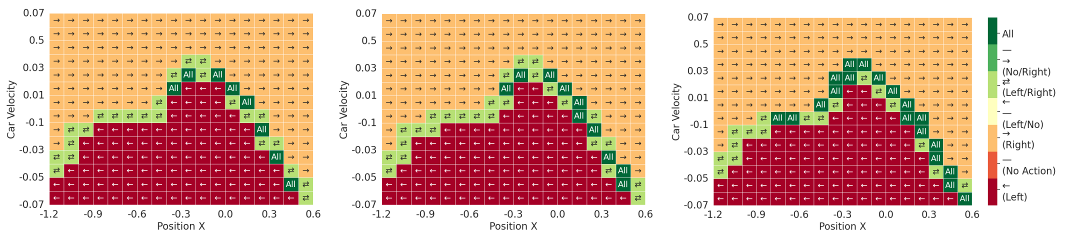


Figure 2: Action Selection (Discrete) based on the Rule List Output (O/P) ranges

For discrete action space, we consider the action with maximum value, i.e., the maximum value of the corresponding rule as suggested by the Rule List for that hypercube; we refer to it as Max Action. However, it could be possible that the Max Action output range is not mutually exclusive, which means it may overlap with other actions. That means the Max Action's minimum value is lesser than some other actions' maximum values, as shown in Figure 2. Then we consider all those overlapped actions along with the Max Action as the action suggested by the Rule List for that hypercube. For example, based on the scenario shown in Figure 2, we consider that the Rule List will suggest the action $a1$ and $a2$. However, as discussed previously, these situations can be resolved by reducing the granularity of state variables until the Max Action output range becomes mutually exclusive for all the hypercubes but at the cost of a larger verification time.

While for continuous action space, each rule's maximum and minimum values suggest the possible range of mean for the distribution of the corresponding action.

B.1.4 Verifying "MountainCar-v0" MAGNets Policy



(a) Testing with Random States (b) Testing with BayesOpt (c) Verification using Sherlock

Figure 3: MAGNets Testing and Verification Car Position-X vs. Car Velocity to Action Heatmap for "MountainCar-v0" environment.

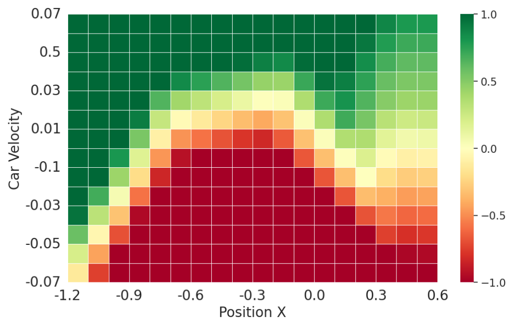
As we have seen in Table 1 of the main paper, the size of each MAGNet varies between 4-16. Also, as we have considered ReLU as the activation function for all the MAGNets' neurons, all the MAGNets can be easily verifiable using Sherlock.

We consider the "MountainCar-v0" environment, which has 16 neurons in each of its MAGNets, to show the verification results along with two other testing methods. "MountainCar-v0" has two observations of the position of the car along the x-axis (POS-X) with a domain range of (-1.2 to 0.6) and the velocity of the car (VEL) with a domain range of (-0.07 to 0.07). We consider a granularity of 0.1 for the axis POS-X and 0.01 for the axis VEL, which therefore divides the state space into (18×14) or 252 two-dimensional hypercubes. We use Sherlock along with Random-State-based testing and Bayesian optimization-based testing to determine the actions for each of these hypercubes. Figure 3 shows the actions determined by each of these methods for each of these hypercubes.

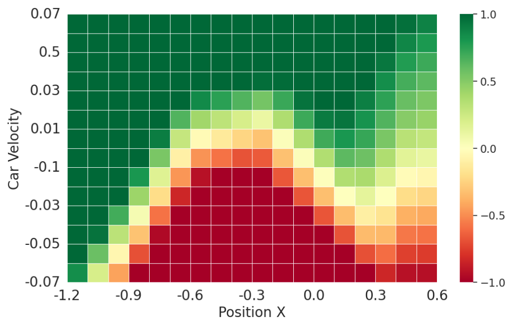
We can observe that even though the Bayesian Optimization-based Testing method is able to predict the possible actions for a given hypercube better than the Random State-based Testing method, it still lags behind the Sherlock-based verification method. Here, with a better prediction, we try to mean that the number of actions the testing or verification technique is able to identify from all the possible actions for a given hypercube. Here, we can see that for all the hypercubes, the identified action set by Random State-based testing is a subset of the identified action set by Bayesian Optimization-based testing, which is a subset of the identified action set by Sherlock-based verification. For example, consider the hypercube {Position X: [0.4,0.5], Car Velocity: [-0.05, -0.04]}. The Random States-based testing identifies the possible action set for that hypercube as {Left, Right}, whereas the BayesOpt-based testing and Sherlock-based verification identify the possible action set for that hypercube as {Left, Right, No Action}, which is the superset of the earlier action set. In contrast, when the hypercube {Position X: [0.5,0.6], Car Velocity: [-0.07, -0.06]}, is considered, Random States-based testing and BayesOpt-based testing identifies the possible action set for that hypercube as {Left, Right}, whereas Sherlock based verification identifies the possible action set for that hypercube as {Left, Right, No Action}, which is the superset of the former action set. This example also demonstrates the necessity of verification methods of neural networks over testing methods to describe the underlying policy accurately.

B.1.5 Verifying "MountainCarContinuous-v0" MAGNets Policy

"MountainCarContinuous-v0" is another OpenAI Gym having a similar state space as the "MountainCar-v0" environment but with one continuous action range between [-1.0 to 1.0]. Affinity to -1.0/1.0 indicates more forceful backward/forward movement of the car in this example. Therefore, we followed the same verification setup to find the actions for different hypercubes. We have plotted our findings in Figure 4. We can see observe a similarity with its discrete counterpart in Figure 3c.



(a) Action Lower Bounds identified by Sherlock



(b) Action Upper Bounds identified by Sherlock

Figure 4: Car Position-X vs. Car Velocity to Action Heatmap for "MountainCarContinouous-v0" environment.

B.1.6 Verifying "CartPole-v1" MAGNets Policy

The "CartPole-v1" environment has two discrete actions, Left and Right, and the state space has four continuous-valued attributes "Cart Position" (or "Position X"), "Cart Velocity," "Pole Angle," and "Pole Angular Velocity." We consider the following bounds and granularity for each attribute, respectively. In Figure 5, we plotted the actions Sherlock found for different hypercubes. The inner dimensions of the figure have Position X or Cart Position as the X-axis and Cart Velocity as the Y-axis, whereas the outer dimensions have Pole Angle as the X-axis and Pole Angular Velocity as the Y-axis. We can observe that Pole Angular Velocity and Cart Velocity are the dominant state space attributes for deciding the action. If (0 $<$ Pole Angular Velocity $<$ 1.0 and Cart Velocity $>$ 0.5) and (1.0 $<$ Pole Angular Velocity $<$ 2.0), then the MAGNet policy suggests action Right, and if (0 $>$ Pole Angular Velocity $>$ -1.0 and Cart Velocity $<$ -0.5) and (-1.0 $>$ Pole Angular Velocity $>$ 2.0), then the MAGNet policy suggests action Left. If (-1 $<$ Pole Angular Velocity 1 and -0.5 $<$ Cart Velocity $<$ 0.5), then MAGNet policy may suggest between Left or Right.

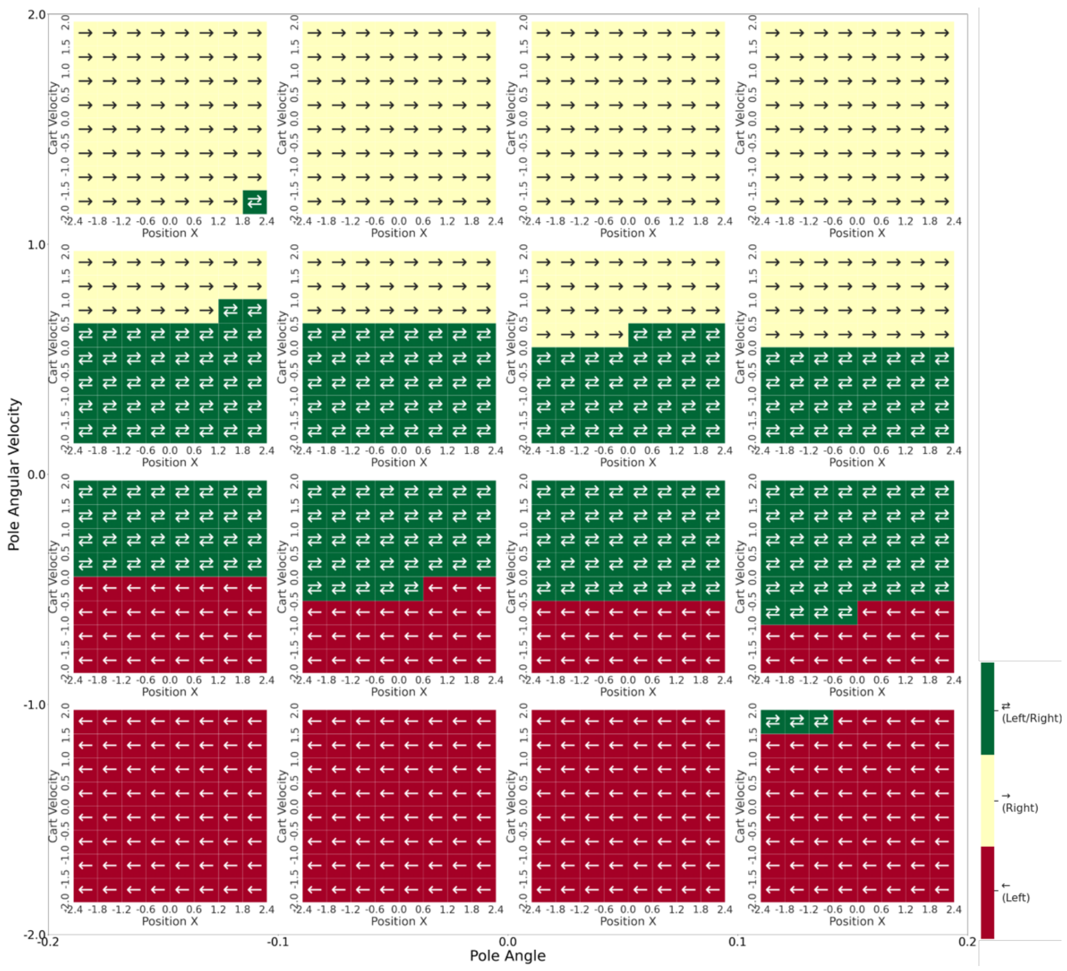


Figure 5: Actions identified by Sherlock for "CartPole-v1" environment MAGNets policy.

State Property	Min Bound	Max Bound	Granularity
Cart Position	-2.4	2.4	0.6
Cart Velocity	-2.0	2.0	0.5
Pole Angle	-0.2	0.2	0.1
Pole Angular Velocity	-2.0	2.0	1.0

Table 1: "CartPole-v1" environments hyperparameters for verification

B.1.7 Verifying "InvertedPendulum-v2" MAGNets Policy

"InvertedPendulum-v2" is an Open-AI Gym Mujoco environment having similar state space as CartPole but has continuous action space [-3.0, +3.0]. Therefore, we use similar attribute bounds as the "CartPole-v1" environment as given in Table 1. However, as the environment has continuous action space, we plotted two graphs, one for the Sherlock-identified action's lower bound (see Figure6) and another for the higher bound (see Figure 7). For both the figures, the inner dimensions have Position X or Cart Position as the X-axis and Cart Velocity as the Y-axis, whereas the outer dimensions have Pole Angle as the X-axis and Pole Angular Velocity as the Y-axis. From Figure6 and Figure7, We can observe that Pole Angular Velocity and Cart Velocity are the deciding state space attributes for choosing action. If ($0 <$ Pole Angular Velocity < 2.0 and Cart Velocity > 0.5), then most of the time, the MAGNet policy suggests action Right, and if ($0 >$ Pole Angular Velocity > -2.0 and Cart Velocity < -0.5) and ($-1.0 >$ Pole Angular Velocity > 2.0), then most of the time, the MAGNet policy suggests action Left.

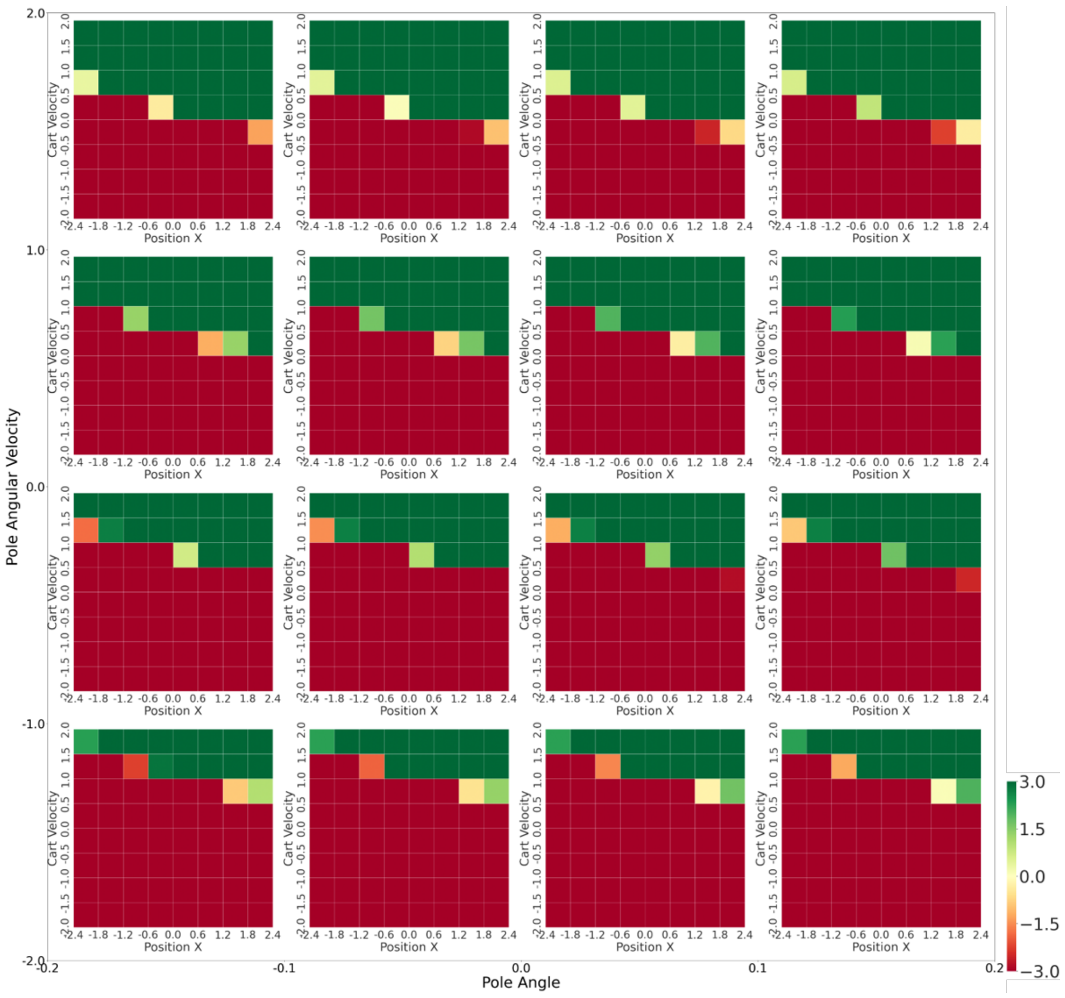


Figure 6: Action Lower Bounds identified by Sherlock for "InvertedPendulum-v2" environment MAGNets policy.

C Additional Baseline

We compare our MAGNets framework with two additional baselines, PIRL (Verma et al. 2018) and VIPER (Bastani, Pu, and Solar-Lezama 2018). PIRL proposes a framework to construct a conventional program based on some defined program grammar and following a DRL-based Oracle. On the other hand, VIPER proposes a framework to convert DRL policy learning into a Decision Tree.

We could not find any official implementation details for these baselines; therefore, we consider the openly available unofficial implementations for both PIRL [1] and VIPER[2]. However, the unofficial implementations lack support, like the VIPER implementation doesn't work for continuous action space. Therefore, we were able to run this algorithm for only four available OpenAI Gym environments with discrete action space. For PIRL, we were able to produce results for only three environments as we were able to create/access the required program grammar for those three environments only. We have given the comparison result in Table 2 average rewards over 100 episodes.

[1]https://github.com/VAIBHAV-2303/PiRL.git

[2]https://github.com/quantumiracle/Cascading-Decision-Tree.git

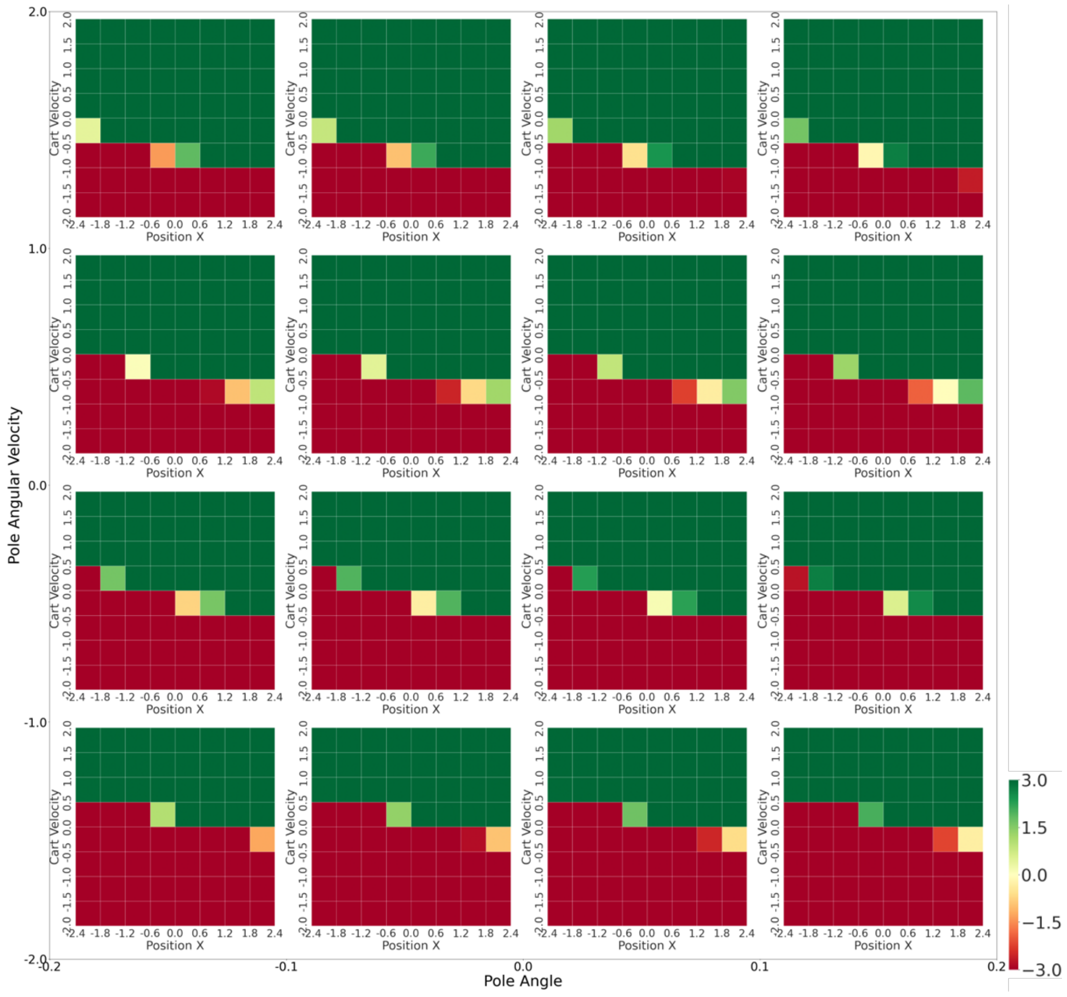


Figure 7: Action Upper Bounds identified by Sherlock for "InvertedPendulum-v2" environment MAGNets policy.

We can see that for the CartPole environment, PIRL and MAGNets policies are able to achieve the perfect reward, while the VIPER (or decision tree-based policy) collected only a reward of 370.41. For the MountainCarContinuous environment, the MAGNets policy performs a little better than the PIRL policy, while for the BipedalWalker environment, the MAGNets policy performs far better than the PIRL policy. For the Acrobot environment, the MAGNets policy performs a little better than the VIPER policy, while for the LunarLander environment, the MAGNets policy performs a little better than the VIPER policy. Overall, the MAGNets policy performs much better than VIPER and PIRL-based policies in terms of average reward.

Environments	**CartPole**	**MountainCarCont**	**BipedalWalker**	**Acrobot**	**LunarLander**
PIRL	500.0	84.71	-61.76	-	-
VIPER	370.41	-	-	-78.14	-382.83
MAGNets	500.0	89.2	280.74	-76.76	259.98

Table 2: Performance Comparison with respect to average rewards

D Compute Configurations and Hyper-parameter choices

We run all our experiments on a Ubuntu 22.04 system with 64 GB of RAM, an i7-8th Gen Processor, and a 24 GB Nvidia RTX-2080 graphics card. We use Python 3.8.13, Pytorch 1.8.0+cu111, Numpy 1.21.3, and OpenAI Gym 0.15.7 on the software end. We have also specified the values of the hyperparameters used in the experiments in Table 3 and Table 4.

Training	Hyperparameters	Value
MAGNets	Learning Rate	1.00E-04
	Train Epochs	250
	Batch Size	4000
	Activation Function	ReLU

Table 3: Hyper-parameters values used for MAGNets Training

Training	Hyperparameters	Value
Original Policy and Modified Policy	Target KL	0.01
	Clip Ratio (ϵ)	0.2
	Gamma (γ)	0.99
	Lambda (λ)	0.97
	(π)-Learning Rate	3.00E-04
	(V)-Learning Rate	1.00E-03
	(π)-Train Iterations	80
	(V)Train Iterations	80
	Train Epochs	1000
	Batch Size	4000
	Activation Function	ReLU

Table 4: Hyper-parameters values used for PPO Training

E Additional Results

E.1 Safety Gym Environments

To test the framework, we have conducted a few experiments on Safety Gym environments (Achiam and Amodei 2019). Our findings are shown in Table 5. Unlike the other experiments, we consider PPO-Lagrangian (Achiam and Amodei 2019) as our baseline, a modified version of the PPO that learns policy considering rewards and costs (safety constraints). We can see that for both of these environments, the MAGNets policy collects nearly the same rewards and cost as the PPO-Lagrangian agent while having only a total of 3 and 1 MAGNets of size (10×10), respectively, instead of the 512 neurons-based PPO-Lagrangian agent.

Environment Name	PPO Size	Layer-M Size	MAGNets Size	PPO-Lag Rewards	MAGNets Rewards	PPO-Lag Costs	MAGNets Costs
Safexp-PointGoal1-v0	256x256	3	10x10	25.13	21.76	28	26.72
Safexp-CarGoal1-v0	256x256	1	10x10	17.57	19.28	25.2	20.79

Table 5: Safety Gym

E.2 PyTorch Profiling

In the result section of the main paper, we have shown the Average Time and Memory taken by the Original PPO Policy and MAGNets Policy for each iteration during testing in "LunarLanderContinuous-v2" and "MountainCarContinuous-v0" environments in edge devices like Arduino Uno R3 and Nvidia Jetson Nano. In Table 6, we have shown similar statistics from the PyTorch profiler. From the results, it is clear that the MAGNets-based policies have a far better advantage in execution time and a much lesser memory footprint (10-20 times faster).

Environment Name	**PPO Time (CPU+GPU)**	**MAGNets Time (CPU+GPU)**	**PPO Memory**	**MAGNets Memory**
MountainCarContinuous-v0	(906+1191) us	(120+118) us	2017.28 Kb	109.38 Kb
LunarLanderContinuous-v2	(993+1325) us	(189+316) us	2037.76 Kb	230.47 Kb

Table 6: PyTorch Profiling

E.3 Effect of varying Layer-M Size while MAGNets Size is Fixed

We have conducted a few experiments to test the sensitivity of our framework for the Layer-M Size hyperparameter. Table 7 varies the size of Layer-M while keeping the MAGNets size fixed. Performance comparison between PPO Policy and MAGNets Policy while varying the size of Layer-M, keeping the size of MAGNets fixed. As the Layer-M size grows, the MAGNets policy earns more and more rewards for "Ant-v2" and "Hopper-v2", but the gains seem to be at a plateau for "HalfCheetah-v2", indicating a similar trend as in varying MAGNet size.

Env Name	**Layer-M Size**	**Magnet Size**	**Reward PPO**	**Reward MAGNets**
	4	4x4	4482.52	2286.3
Ant-v2	10	4x4	4482.52	3966.27
	14	4x4	4482.52	4636.13
	2	8x8	1286	199.46
Hopper-v2	3	8x8	1286	274
	4	8x8	12864	1273.8
	1	4x4	4876.75	825.03
HalfCheetah-v2	2	4x4	4876.75	4419.72
	4	4x4	4876.75	4614.9

Table 7: Effect of varying Layer-M Size while MAGNets Size is Fixed

E.4 Effect of varying MAGNets Size while Layer-M Size is Fixed

We have conducted a few experiments to test the sensitivity of our framework for the MAGNets Size hyperparameter. Table 8 varies the size of MAGNets while keeping the Layer-M size fixed. Performance comparison between PPO Policy and MAGNets Policy while varying the size of MAGNets, keeping the size of Layer-M fixed. As the size of MAGNets grows, the policy earns more and more rewards for "Acrobot-v1" and "LunarLander-v2", but the gains for "BipedalWalker-v3" are marginal. This shows the benefits of increasing the size of MAGNets plateaus at some time.

Table 7 and Table 8 show the tradeoff between the dimensions of the network and the rewards obtained. PPO-like rewards are attained With larger Layer-M or larger MAGNets size, even though the networks are still much smaller. Choosing the suitable Layer-M and MAGNets sizes requires tuning to achieve satisfactory rewards on a target platform. This is enabled by the proposed method.

Env Name	Layer-M Size	MAGNets Size	Reward PPO	Reward MAGNets
Acrobot-v1	1	6x6	-61.74	-78.86
	1	8x8	-61.74	-76.82
	1	10x10	-61.74	-70.34
BipedalWalker-v3	4	4x4	298.85	222.04
	4	6x6	298.85	276.23
	4	8x8	298.85	280
LunarLander-v2	4	2x2	276.14	170.59
	4	3x3	276.14	180.84
	4	4x4	276.14	260

Table 8: Effect of varying MAGNets Size while Layer-M Size is Fixed

E.5 Average Episodic Test Reward Graphs

We have plotted the average episodic rewards and standard deviation in Figure 8 (over 100 episodes) collected using four different seeds for the baselines and MAGNets policies for various OpenAI Gym Environments. The tabular results and corresponding analysis for this visualization have already been provided in the Result Section of the main paper.

References

Achiam, J.; and Amodei, D. 2019. Benchmarking Safe Exploration in Deep Reinforcement Learning.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Neural Information Processing Systems*.

Ghosh, A.; Dhebar, Y.; Guha, R.; Deb, K.; Nageshrao, S.; Zhu, L.; Tseng, E.; and Filev, D. 2021. Interpretable AI Agent Through Nonlinear Decision Trees for Lane Change Problem. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, 01–08.

Milani, S.; Zhang, Z.; Topin, N.; Shi, Z. R.; Kamhoua, C.; Papalexakis, E. E.; and Fang, F. 2023. MAVIPER: Learning Decision Tree Policies for Interpretable Multi-Agent Reinforcement Learning. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2022, Grenoble, France, September 19–23, 2022, Proceedings, Part IV*, 251–266. Springer.

Qiu, W.; and Zhu, H. 2022. Programmatic Reinforcement Learning without Oracles. In *International Conference on Learning Representations*.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 5045–5054. PMLR.

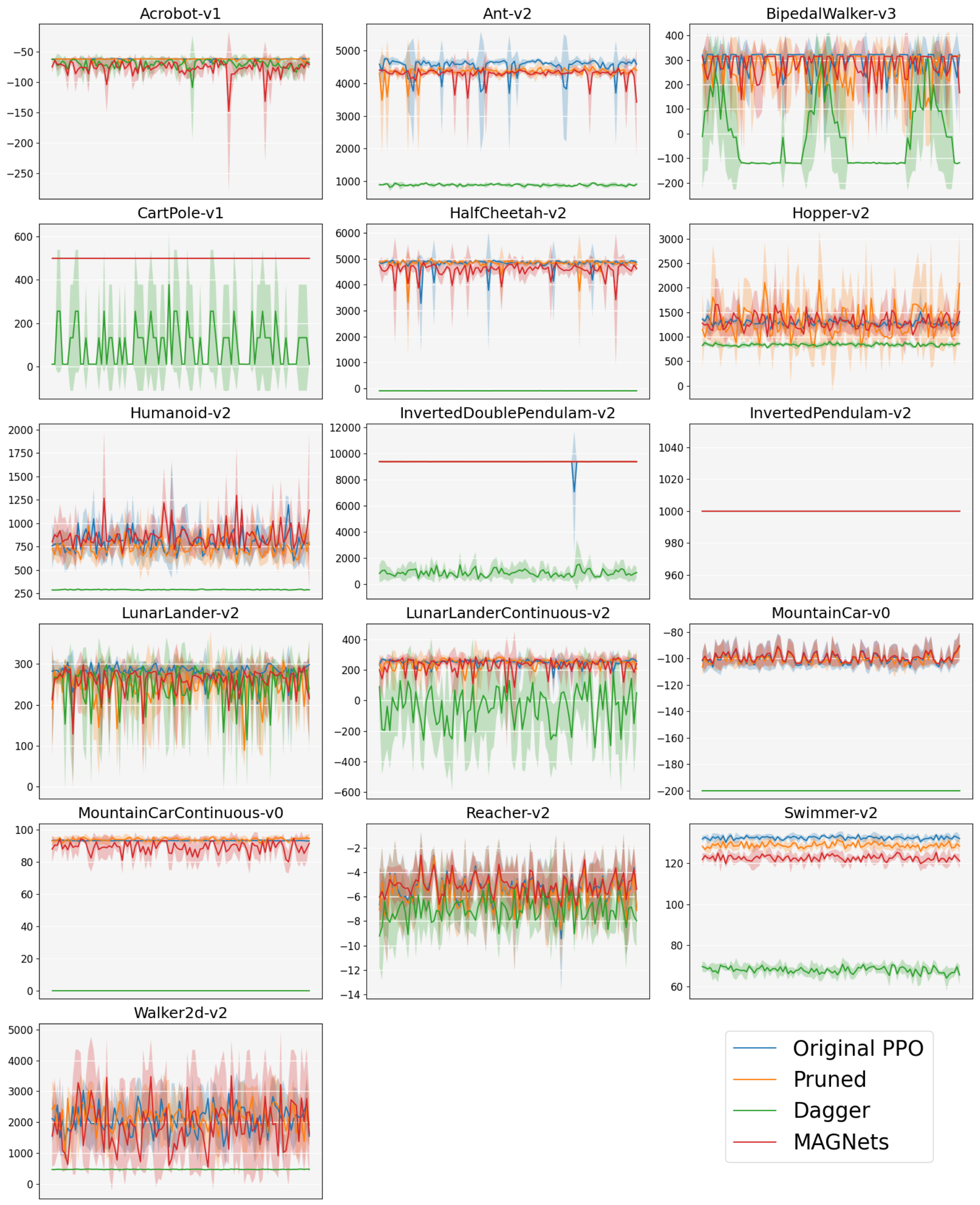


Figure 8: Average Rewards collected by baseline policies and MAGNets policy over 100 episodes and four different seeds for various OpenAI Gym Environments.