Imitation Learning of Logical Program Policies for Multi-Agent Reinforcement Learning

Manuel Eberhardinger[1,*], Johannes Maucher[1] and Setareh Maghsudi[2]

[1]Hochschule der Medien Stuttgart, Germany
[2]University of Tuebingen, Germany

Abstract
This work shows that it is possible to learn Logical Program Policies (LPP) for the Multi-Agent Reinforcement Learning environment called Level-based Foraging. Policies are derived from decision trees that use a logical combination of small feature detector programs generated by a probabilistic context-free grammar. We show that these created programmatic policies allow to explain decisions of all agents in the environment and also can be used without retraining on all other environments. This is possible by creating not just one program for each agent, but several, by sampling learned programs and selecting the best programs according to the approximate reward in the specific environment the program was learned on. By using LPP to represent the policies for agents, the policies are interpretable and also extrapolate to unseen environments. Code: https://github.com/ManuelEberhardinger/LPP-MARL

Keywords
Program Synthesis, Multi-Agent Reinforcement Learning, Imitation Learning

1. Introduction

Programmatic reinforcement learning is an interesting research direction to make black-box deep reinforcement learning (DRL) policies interpretable. By using a synthesized program to control an agent, the behaviour of the agent is interpretable, as the generated program can be studied and verified by experts before using it in dangerous environments. Additionally, the program is controllable, as software developers can adjust the program to their own needs. Programs also tend to extrapolate to out-of-distribution data, as programs are not restricted to specific input sizes like neural network models.

Nevertheless, learning a programmatic policy directly, is very hard and most work use a form of Imitation Learning and a customised domain-specific language (DSL) to reduce the search space and make the synthesis of a program feasible [1, 2, 3].

We show that it is possible to learn programs for multiple agents by using a strong prior in the DSL so that the search for programs becomes feasible. Programs are learnt by using Imitation Learning after collecting data from a black-box neural network policy. In this work

Proceedings of the Second International Workshop on Explainable and Interpretable Machine Learning (XI-ML 2022) co-located with the 45rd German Conference on Artificial Intelligence (KI 2022), September 20, 2022, Trier (Virtual), Germany
[*]Corresponding author.

eberhardinger@hdm-stuttgart.de (M. Eberhardinger); maucher@hdm-stuttgart.de (J. Maucher); setareh.maghsudi@uni-tuebingen.de (S. Maghsudi)

we propose to use Logical Program Policies [4] to generate programs for the Multi-Agent Reinforcement Learning (MARL) environment Level-based Foraging (see Figure 2) [5]. Figure 1 shows a high-level overview of all necessary steps to create programs for a given MARL environment.

We show that the generated logical programs are interpretable and also extrapolate to unseen environments as the programs are not restricted by grid sizes, the number of players or the number of objects. This makes programs a good alternative for black-box neural network policies which must be retrained if one changes the details of the environment, even though synthesized programs do not achieve the same performance as the DRL policy which was trained until convergence and used for data collection.

2. Related Work

2.1. Program Synthesis and Programmatic Policies

Program synthesis has a long history in the artificial intelligence research community [6, 7]. In recent years, more and more research has been conducted on combining deep learning and program synthesis to make program search more feasible by reducing or guiding the search space [8, 9, 10]. Another promising method is to learn a library of functions from previously solved problems. These functions are then reusable in an updated DSL to solve more difficult problems [11, 12].

Existing methods for creating programmatic policies mostly use Imitation Learning. Verma et al. use program sketches and a neural network oracle to synthesise programmatic policies [1, 2]. Other work synthesised finite state machines to represent policies [3] or extract policies from decision trees [4, 13]. Recent promising work also shows that it is possible to synthesise programmatic policies directly [14, 15], since distilling policies into a program always leads to performance degradation on a given task.

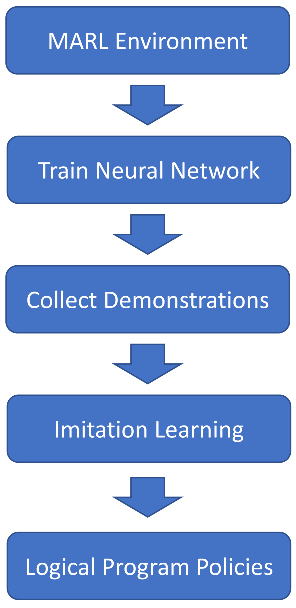

Figure 1: The high level overview of the necessary steps to create programs for a given MARL environment.

2.2. Multi-Agent Reinforcement Learning

The goal of MARL is to jointly learn multiple agents to solve a given task by learning how to behave in a shared environment. There exists a variety of paradigms how to learn MARL algorithms by adapting DRL methods for multiple agents. One approach is to learn the agents independently or with sharing of information [16]. The latter paradigm is also known as Centralised Training Decentralised Execution [17].

Program synthesis research for MARL was first conducted to derive programmatic communication rules by imitating a learned transformer communication policy [18]. To the best of our knowledge, there exists no work that tests the extrapolation qualities of learned policies on unseen environments.

2.3. Level-based Foraging

The implementation of the Level-based Foraging environment that is used in this work was introduced in [19] (see Figure 2). Each agent and fruit has a specific level. Possible actions are moving in each direction or picking up a fruit. The goal of the agents is to pick up all fruits, but agents can only pick up fruits if their level is greater or equal than the level of the fruit. If the level of the fruit is higher than the agents level, multiple agents need to cooperate to pick up the fruit. Therefore this is a mixed cooperative-competitive environment, as agents compete in picking up fruits but also need to cooperate for some fruits. The reward is divided by the agents that were involved in foraging of the fruit.

3. Method

This work adapts the Bayesian Imitation Learning method Logical Program Policies that calculates a posterior distribution of logical formulas h for MARL environments [4]. The logical formula consists of small feature detector programs, denoted as f, that are drawn from a probabilistic context-free grammar (PCFG) $\mathcal{G}$. The goal of the feature detectors is to parametrise the state-action pairs of the expert demonstrations D with boolean values if the given action was taken in the given state: $f(s,a): S \times A \rightarrow \{0,1\}$.

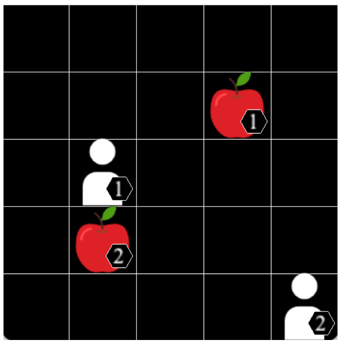

Figure 2: The environment Level-based foraging with two players and two fruits.

Actions taken in a given state represent positive examples. In the original method all actions not taken for a specific state were added as negative examples. However, this is not applicable for our purpose, since in a given state several actions are possible that lead to the same goal, namely to approach a fruit in order to pick it up. Therefore, we add as negative examples only actions that increase the distance of the agent from the fruit.

This newly created dataset can then be used to train a binary decision tree classifier. The logical formula is represented in a disjunctive normal form and can be extracted from this classifier as a combination of a finite subset of feature detector programs

$$h(s,a) \widehat{=} (f_{1,1}(s,a) \wedge \ldots \wedge f_{1,k_1}(s,a)) \vee \ldots \vee (f_{d,1}(s,a) \wedge \ldots \wedge f_{1,k_d}(s,a)), \tag{1}$$

where d is the number of the demonstrations and k is the number of the feature detector programs. Negation of f is also possible. These extracted formulas can be converted into a flow chart for making decisions of the agents interpretable for humans [20].

3.1. Approximating the posterior

The following equations (adapted for our purpose from [4]) show how to approximate the posterior distribution q from the prior $p(\pi)$ and the likelihood $p(D|\pi)$ where D are the expert demonstrations:

$$q(\pi) \approx p(\pi|D) \tag{2}$$

$$\forall\, 0 \leq w \leq 1 : p(\pi|D) = w * p(\pi) + p(D|\pi) - p(D) \tag{3}$$

$$p(\pi) = \text{the log probability of the generated feature detectors by } \mathcal{G} \tag{4}$$

$$p(D|\pi) = \ln(\frac{1}{N}\sum_{n}^{N}\sum_{i}^{Agents} r_i) \tag{5}$$

$$p(D) = \text{the log probability of the evidence, e.g. the provided demonstrations} \tag{6}$$

The prior of the distribution is defined as the weighted log probability of the generated feature detectors from the PCFG. Since w is defined to decrease the importance of the length of the programs, it must be in between $0 \leq w \leq 1$. We use $w = 0.1$ in our experiments as the environment is more complex than the grid games used in [4]. This decreases the importance of the length of the programs since very short feature detectors are not able to find good abstractions for our problem. The likelihood is calculated from the log of the average reward each agent received by following a given policy π for N runs. We set $N = 100$ to get a good approximation of how good the program is on the given environment. Equation 3 adds and subtracts the probabilities instead of multiplying and dividing them, since we work with log probabilities.

To better approximate the posterior, Silver et al. [4] use M logical formulas $h_1, ..., h_M$ in one LPP, as a weighted mixture of experts with each formula weighted equally. In our experiments we set $M = 25$. To provide a more diverse distribution of h to select, we train multiple decision trees with different random seeds. In our experiments we choose to train 5 decision trees. This improves the performance of LPP, since hardly one formula captures the complete structure of the state space. The selection of M formulas is implemented by sampling a subset of M programs from all generated decision trees and empirically evaluating which subset returned the highest average reward after N test runs of the environment. The more often M decision trees are sampled, the better the approximation for the best LPP. We run this 100 times in our experiments.

To use LPP in MARL, we store all combinations of evaluated M logical formulas in a list, and instead of returning only the best policy, the same number of policies as agents are returned.

Following equation derives the final policy $\pi_*(s)$ for using at test time:

$$\pi_*(s) = \underset{a\in A}{\arg\max}\, \mathbb{E}_q[\pi(a,s)] = \underset{a\in A}{\arg\max} \sum_{h\in q} q(h)\pi_h(a|s) \tag{7}$$

In words, each selected logical formula calculates for all actions the probability with which an action a should be taken in a given state s. The probabilities for the same action are added and then the action with the highest probability is returned.

3.2. Domain-specific Language for the feature detectors

The DSL was updated according to the requirements of the environment. We added more specific perceptual primitives that the feature detector programs can use to represent the state succinctly. These are the functions that can be used to determine in which direction the fruit is located from the agent (see Table 3 in the appendix). We also had to include the agent's position information as a parameter in the function so that the program would know which agent was being controlled.

3.3. The neural network oracle

For training of the neural network policy we choose to use the IQL algorithm [16] reported from [17] which received on both 8x8 grid size environments perfect reward. IQL is an adaptation of the Q-Learning algorithm [21] for multiple agents in which each agent has an independent state-action value function. The loss function is the default Q-learning loss, which is independently minimised for each agent's local past observations and actions [17]. We trained the model with the default parameters reported in the paper on the smallest environment on a grid size 5x5 with two players and two fruits. This model was then used to collect data for generation of the LPPs.

3.4. Summary of the algorithm

Algorithm 1 shows a high level overview of all necessary steps to generate programs with LPP for the level-based foraging environment. First, we need an oracle to create the positive expert demonstrations. Then we can create negative examples from the positive ones. The next step ist to generate all feature detector programs from $\mathcal{G}$ and calculate the corresponding prior. Then a dataset is created by running all feature detectors on all demonstrations. This dataset is used for training of multiple decision trees. The trained classifiers are evaluated to approximate the likelihood of the models using equation 5. After computing the posterior distribution with equation 3, we can select the best models, which are then converted into specific LPP polices by equation 7.

Algorithm 1: The complete algorithm for generating a program with LPP

```
Input  : neural network $\theta$, ensemble size $M$, number of demos $d$ and programs $k$
Output : the n best LPP policies
1  $w = 0.1$;                                        // the importance of the prior
2  $D$ = collect_demos_with_oracle($\theta$, $d$);
3  $\overline{D}$ = generate_anti_demos($D$);
4  $f$ = generate_feature_detectors_from_dsl($k$);
5  priors = calculate_priors($f$);
6  $X = \{(f_1(s,a), ..., f_k(s,a))^d : (s,a) \in D \cup \overline{D}\}$ ;   // generate data for learning decision trees
7  $Y = \{b_0, ..., b_d\}$ ;   // boolean truth values derived from $D \cup \overline{D}$ are the targets with $b \in \{0,1\}$
8  models = train_decision_tree_classifiers($X, Y, M$);
9  likelihoods = evaluate_likelihoods(models);
10 posterior = $w$·priors + likelihoods - ln(P(evidence));
11 LPPs = select_best_lpp_models(models, posterior);
12 return LPPs;
```

4. Experiments

In the experiments we want to compare the extrapolation capabilities of the neural network oracle and the created LPP policies. As extrapolation we define all larger environments than the

Table 1

Overview of the results. We always evaluated the average of 100 runs on the given environment. To use IQL for bigger environments we have to use a partial observability of 5x5 so that the input size will match.

Results		
Environment	LPP	IQL 5x5
Foraging-grid-5x5-2p-2f-v2	0.82	0.98
Foraging-grid-8x8-2p-4f-v2	0.71	0.27
Foraging-grid-10x10-2p-4f-v2	0.60	0.18
Foraging-grid-12x12-2p-4f-v2	0.63	0.09
Foraging-grid-14x14-2p-5f-v2	0.53	0.04
Foraging-grid-16x16-2p-6f-v2	0.54	0.05
Foraging-grid-18x18-2p-8f-v2	0.56	0.03

one which was used to train the neural network model. Training models on bigger grid sizes and then testing them on smaller ones, will not test the extrapolation capabilities as the smaller environments are just a subset of the larger environment. This would be interpolation of the state space and not extrapolation. This trained model is then used as an oracle and also as a comparison baseline.

As it is not easily possible to test the trained neural network model on other environments, we decided to use a partial observability for other grid sizes. Using partial observability makes it possible to run the neural network also on different grid sizes than the one it was trained on. We also evaluate the programmatic policies on different numbers of players. However, we cannot compare them to the trained neural network policies, since they cannot be easily adapted to different numbers of players.

Table 1 shows that we only loose performance on the smallest environment, that was used to generate the data from the oracle. The smallest environment is called `Foraging-grid-5x5-2p-2f-v2`, with a grid size of 5x5, the number before p are the players and the number before f are the amount of fruits. On all larger environments our approach outperforms the neural network policy by far.

Table 2 shows the results when using the created LPP policies for more players than the ones trained on. It is clearly visible that the programs can extrapolate to all other environments. With three players the performance drops, but with more players the performance increases again.

4.1. Number of demonstrations

In this experiment we want to analyse how important the provided number of demonstrations are. Therefore, we run the generation of LPP policies with different numbers of demonstrations. Figure 3 shows how the number of demonstrations affect the performance of the created policies. It is observable that providing too many demonstrations result in not finding a program at all. We choose to use 50 demonstrations as this results in the best performance. In addition, it takes as few as 15 demonstrations to achieve a performance almost as good as the best. However, we only evaluated the performance on the `Foraging-grid-5x5-2p-2f-v2` environment.

Table 2
Overview of the results on different number of players. We always evaluated the average of 100 runs on the given environment.

Results on different number of players			
Environment	LPP	Environment	LPP
Foraging-grid-5x5-3p-2f-v2	0.59	Foraging-grid-5x5-5p-2f-v2	0.91
Foraging-grid-8x8-3p-4f-v2	0.38	Foraging-grid-8x8-5p-4f-v2	0.68
Foraging-grid-10x10-3p-4f-v2	0.35	Foraging-grid-10x10-5p-4f-v2	0.58
Foraging-grid-12x12-3p-4f-v2	0.36	Foraging-grid-12x12-5p-4f-v2	0.60
Foraging-grid-14x14-3p-5f-v2	0.26	Foraging-grid-14x14-5p-5f-v2	0.54
Foraging-grid-16x16-3p-6f-v2	0.25	Foraging-grid-16x16-5p-6f-v2	0.51
Foraging-grid-18x18-3p-8f-v2	0.24	Foraging-grid-18x18-5p-8f-v2	0.51
Foraging-grid-5x5-4p-2f-v2	0.76	Foraging-grid-5x5-6p-2f-v2	0.91
Foraging-grid-8x8-4p-4f-v2	0.51	Foraging-grid-8x8-6p-4f-v2	0.79
Foraging-grid-10x10-4p-4f-v2	0.48	Foraging-grid-10x10-6p-4f-v2	0.74
Foraging-grid-12x12-4p-4f-v2	0.46	Foraging-grid-12x12-6p-4f-v2	0.70
Foraging-grid-14x14-4p-5f-v2	0.48	Foraging-grid-14x14-6p-5f-v2	0.65
Foraging-grid-16x16-4p-6f-v2	0.36	Foraging-grid-16x16-6p-6f-v2	0.60
Foraging-grid-18x18-4p-8f-v2	0.39	Foraging-grid-18x18-6p-8f-v2	0.58

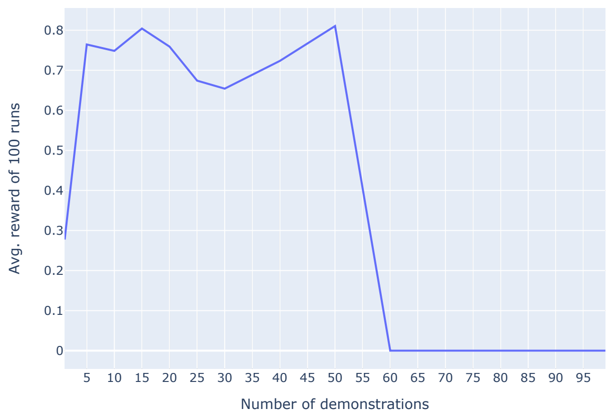

Figure 3: This plot shows how the number of demonstrations affect the performance of the LPP policy.

4.2. Using the same program for multiple agents

We also evaluated how the same LPP policy behaves if used for multiple agents. Figure 4 shows that the overall performance decreased on all environments. We think that the reason for this behaviour is, that the agents get more often stuck as they want to move to the same position or one agent blocks the way to the fruit from the other agent. We concluded that the program was not able to learn how to navigate around the other agent. This is also the case for the still existing gap between the average reward from the oracle and the created programs.

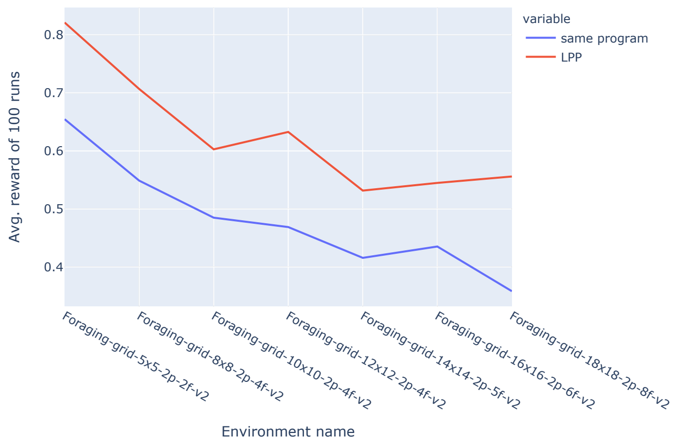

Figure 4: This plot shows the performance gap when using the same LPP policy for multiple agents instead of different ones.

4.3. Interpretation of the decision making process

Figure 5 shows an example code block why a given action was selected. The analysis of the program shows that the program search still needs improvement, as not the best programs are found and unnecessary parts are visible such as the three negated `action_is_east`, `action_is_west` and `action_is_south` checks. Theses checks will never be false in the formula as `action_is_load` already filters out other actions. As programs are only interpretable by someone who has a basic understanding about programming languages, further work can be invested in visualising a flowchart why a chosen action was taken for a single agent and state similar to [20].

5. Discussion and Conclusion

This work shows that it is possible to make decisions of Multi-Agent Systems interpretable by adapting the Logical Program Policies method. We show that generated programs on the

```
((action_is_load(cell_is_value( lbc.AGENT.value , action, state, pos) , action, state, pos)
  and not (action_is_east(fruit_is_east(state, pos) , action, state, pos))
  and not (action_is_west(fruit_is_west(state, pos) , action, state, pos))
  and not (action_is_south(fruit_is_south(state, pos) , action, state, pos))
  and not (action_is_load(fruit_is_west(state, pos) , action, state, pos))
  and not (action_is_load(fruit_is_east(state, pos) , action, state, pos))))
```

Figure 5: This code block shows the part of the formula why the action to pick up a fruit was selected. We have displayed only the part that was responsible for selecting the ‘Pick up a fruit‘ action. The other disjunctive parts of the logical formula were omitted for clarity as the program would get too large.

smallest environment can be used without retraining on other environments or players and are as well interpretable for any person if converted into a flowchart. This is only possible with using a strong prior in the DSL as otherwise programs are not interpretable and also do not generalise to other instances of the same environment.

Nevertheless, there is still a moderate performance gap between the neural network models and the created programs. One reason for this is, that distilling a policy into a program always results in loss of information [15]. Additionally, at the moment it is not possible for the agents to communicate with each other. Incorporating communication into the DSL could be a promising way to increase the performance of the programmatic policies, but only if the program also learns how to navigate around other agents.

An interesting way that agents could learn this behaviour is through the use of library learning as proposed in [12]. Right now, the DSL does not adequately represent how to navigate when there are objects between the agent and the selected target. Learning a library of functions would make it possible for the program to navigate around each agent when seen several times from the oracle. Another advantage is that one does not need to incorporate background knowledge of the fruit location in the DSL. The method should learn these functions itself by updating the DSL when the same structures are found in multiple programs. A disadvantage of learning a library is that much more demonstrations and computational power would be required to learn these functions.

References

[1] A. Verma, V. Murali, R. Singh, P. Kohli, S. Chaudhuri, Programmatically interpretable reinforcement learning, in: J. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, PMLR, 2018, pp. 5045–5054. URL: https://proceedings.mlr.press/v80/verma18a.html.

[2] A. Verma, H. M. Le, Y. Yue, S. Chaudhuri, Imitation-projected programmatic reinforcement learning, in: Proceedings of the 33rd International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, 2019.

[3] J. P. Inala, O. Bastani, Z. Tavares, A. Solar-Lezama, Synthesizing programmatic policies that inductively generalize, in: International Conference on Learning Representations, 2020. URL: https://openreview.net/forum?id=S1l8oANFDH.

[4] T. Silver, K. R. Allen, A. K. Lew, L. P. Kaelbling, J. Tenenbaum, Few-shot bayesian imitation learning with logical program policies, in: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, 2020, pp. 10251–10258. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6587.

[5] S. V. Albrecht, S. Ramamoorthy, A game-theoretic model and best-response learning method for ad hoc coordination in multiagent systems, in: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Richland, SC, 2013, p. 1155–1156.

[6] R. J. Waldinger, R. C. T. Lee, Prow: A step toward automatic program writing, in: IJCAI, 1969.

[7] Z. Manna, R. Waldinger, Knowledge and reasoning in program synthesis, in: Proceedings

of the 4th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'75, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1975, p. 288–295.

[8] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, Deepcoder: Learning to write programs, in: Proceedings International Conference on Learning Representations (ICLR), 2017. URL: https://openreview.net/pdf?id=rkE3y85ee.

[9] M. Nye, L. Hewitt, J. Tenenbaum, A. Solar-Lezama, Learning to infer program sketches, in: K. Chaudhuri, R. Salakhutdinov (Eds.), Proceedings of the 36th International Conference on Machine Learning, volume 97 of *Proceedings of Machine Learning Research*, PMLR, 2019, pp. 4861–4870. URL: https://proceedings.mlr.press/v97/nye19a.html.

[10] D. Ritchie, A. Thomas, P. Hanrahan, N. D. Goodman, Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks, in: Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16, Curran Associates Inc., Red Hook, NY, USA, 2016, p. 622–630.

[11] L. Hewitt, T. Anh Le, J. Tenenbaum, Learning to learn generative programs with memoised wake-sleep, in: J. Peters, D. Sontag (Eds.), Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI), volume 124, PMLR, 2020, pp. 1278–1287.

[12] K. Ellis, C. Wong, M. I. Nye, M. Sablé-Meyer, L. Morales, L. B. Hewitt, L. Cary, A. Solar-Lezama, J. B. Tenenbaum, Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning, in: PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, 2021, pp. 835–850. URL: https://doi.org/10.1145/3453483.3454080. doi:10.1145/3453483.3454080.

[13] O. Bastani, Y. Pu, A. Solar-Lezama, Verifiable reinforcement learning via policy extraction, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, 2018, p. 2499–2509.

[14] D. Trivedi, J. Zhang, S.-H. Sun, J. J. Lim, Learning to synthesize programs as interpretable and generalizable policies, in: A. Beygelzimer, Y. Dauphin, P. Liang, J. W. Vaughan (Eds.), Advances in Neural Information Processing Systems, 2021. URL: https://openreview.net/forum?id=wP9twkexC3V.

[15] W. Qiu, H. Zhu, Programmatic reinforcement learning without oracles, in: International Conference on Learning Representations, 2022. URL: https://openreview.net/forum?id=6Tk2noBdvxt.

[16] M. Tan, Multi-agent reinforcement learning: Independent vs. cooperative agents, in: In Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann, 1993, pp. 330–337.

[17] G. Papoudakis, F. Christianos, L. Schäfer, S. V. Albrecht, Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks, in: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS), 2021.

[18] J. P. Inala, Y. Yang, J. Paulos, Y. Pu, O. Bastani, V. Kumar, M. C. Rinard, A. Solar-Lezama, Neurosymbolic transformers for multi-agent communication, in: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.

[19] F. Christianos, L. Schäfer, S. V. Albrecht, Shared experience actor-critic for multi-agent reinforcement learning, in: Advances in Neural Information Processing Systems (NeurIPS),

2020.

[20] J. Skirzyński, F. Becker, F. Lieder, Automatic discovery of interpretable planning strategies, Mach. Learn. 110 (2021) 2641–2683. URL: https://doi.org/10.1007/s10994-021-05963-2. doi:10.1007/s10994-021-05963-2.

[21] C. J. C. H. Watkins, P. Dayan, Q-learning, Machine Learning 8 (1992) 279–292. URL: https://doi.org/10.1007/BF00992698. doi:10.1007/BF00992698.

A. Appendix

Table 3
The domain-specific language which is represented by a probabilistic context-free grammar. The start symbol is denoted by P. The prior for one generated program is the log of the multiplied probability of all used production rules. The filters for the correct actions are necessary, as each feature detector only applies to the action it was generated for.

Probabilistic Context Free Grammar		
Production Rule & Method	Probability	Description of the Method
Filter for correct actions		
S → action_is_west(C)	0.2	checks if the action is going to the west
S → action_is_east(C)	0.2	checks if the action is going to the east
S → action_is_south(C)	0.2	checks if the action is going to the south
S → action_is_north(C)	0.2	checks if the action is going to the north
S → action_is_load(C)	0.2	checks if the action is picking up the fruit
Conditions		
C → shifted(O, B)	0.5	shifts agents position and then checks a condition
C → B	0.5	
Base Conditions		
B → cell_is_value(V)	0.33	checks the value on the attended position
B → action_is_executable()	0.33	checks if selected action is executable
B → F	0.33	
Fruit Conditions		
F → fruit_is_east()	0.2	checks if fruits are east of the agent
F → fruit_is_south()	0.2	checks if fruits are south of the agent
F → fruit_is_west()	0.2	checks if fruits are west of the agent
F → fruit_is_north()	0.2	checks if fruits are north of the agent
F → fruit_is_pickable()	0.2	checks if agent can pickup a fruit
Directions		
O → (P, P)	0.25	Offsets for two positive numbers
O → (N, N)	0.25	Offsets for two negative numbers
O → (P, N)	0.25	Offsets for a positive and negative number
O → (N, P)	0.25	Offsets for a negative and positive number
Natural Numbers		
P → 0,...,4	each 0.2	Positive natural numbers
N → 0,-1,...,-4	each 0.2	Negative natural numbers
Values		
V → agent	0.5	the value that represents an agent
V → fruit	0.5	the value that represents a fruit