Ideas for a Reinforcement Learning Algorithm that Learns Programs

Susumu Katayama(✉)

University of Miyazaki, 1-1 W. Gakuenkibanadai,
Miyazaki, Miyazaki 889-2192, Japan
skata@cs.miyazaki-u.ac.jp

Abstract. Conventional reinforcement learning algorithms such as Q-learning are not good at learning complicated procedures or programs because they are not designed to do that. AIXI, which is a general framework for reinforcement learning, can learn programs as the environment model, but it is not computable.AIXI has a computable and computationally tractable approximation, MC-AIXI(FAC-CTW), but it models the environment not as programs but as a trie, and still has not resolved the trade-off between exploration and exploitation within a realistic amount of computation.

This paper presents our research idea for realizing an efficient reinforcement learning algorithm that retains the property of modeling the environment as programs. It also models the policy as programs and has the ability to imitate other agents in the environment.

The design policy of the algorithm has two points: (1) the ability to program is indispensable for human-level intelligence, and (2) a realistic solution to the exploration/exploitation trade-off is teaching via imitation.

Keywords: AIXI · Inductive programming · Imitation

1 Introduction

Artificial General Intelligence (AGI) is, unlike the conventional *Narrow AI* that is implemented by experts to behave intelligently for specific purposes, machine intelligence that can deal with various unknown problems and unexpected situations in the same way as human beings.

Some researchers of conventional reinforcement learning (RL) algorithms used to insist this kind of intelligence as an advantage of RL. However, conventional RL algorithms have not actually been able to deal with unexpected situations in the same way as humans, because they require generalizations to be designed for each problem specifically in order to learn efficiently enough for practical purposes. In general, dealing with various problems as intelligently as humans requires the ability to program (often repetitive or recursive) procedures for problem solving. Nevertheless, RL research from the viewpoint of **"learning to obtain procedures or programs"** had not been focused on for decades.

At the beginning of this century, Marcus Hutter (e.g. [2]) devised AIXI which is a theoretical, general framework for RL. AIXI involves conventional RL algorithms, and at the same time learns the environment model as Turing machines or programs. Its drawback is that AIXI is not computable, and its approximation AIXI*tl* requires prohibitively enormous computation. MC-AIXI(FAC-CTW) [10], which is another AIXI approximation requiring less computation, models the environment not as Turing machines but as Context Tree Weighting (CTW) [11] that is essentially a tabular representation using a PATRICIA tree which is a kind of trie. In addition, MC-AIXI(FAC-CTW) has not resolved the trade-off between exploration and exploitation within practical computation.

In the real environment, it is not realistic to expect single agents to thoroughly explore, because they must avoid danger that can cause their death. Under totally unknown situations, real agents must avoid danger, sometimes based on prior knowledge (or "instinct"), and sometimes by behaving in the same way as other surviving agents. We should also note that search methods by population such as genetic algorithms, which let each individual exchange information, are known to be effective for finding globally optimal solutions.

This paper presents our idea of an RL algorithm that (1) models the policy, as well as the environment, as programs, and (2) can imitate other agents (and maybe other teachers), or learn behavior from the environment model in order to explore by population.

The rest of this paper is organized in the following way. Section 2 introduces AIXI and its existing approximations, along with our speculations. Section 3 describes the RL algorithm we are going to implement in details. Section 4 discusses how to evaluate the implemented system. Section 5 concludes this paper.

2 AIXI and its Approximations

This section explains AIXI and its existing approximations, along with our speculations.

2.1 AIXI

AIXI [2] is a general framework for RL and a theoretical tool for discussing the limitation of intelligent agents. It is designed to be as general as possible, and it can be considered to cover almost all abilities of artificial (and possibly natural, i.e. human) intelligence.

AIXI is based on the following ideas:

– modeling the environment as Turing machines;
– retaining all of an infinite number of possible environment candidates along with their plausibilities;
– respecting the idea of Occam's razor: simpler environments, or environments that can be described using shorter programs, should be more plausible.

More concretely, an AIXI agent decides the action $\dot{a}_k$ at time k from the *history* h_k at time k, or the sequence of the actual actions, rewards, and observations $\dot{a}\dot{r}\dot{o}_{1:k-1}$ until time $k-1$, by using the following equation:

$$\dot{a}_k = \arg\max_{a_k \in \mathcal{A}} \sum_{r_k o_k \in \mathcal{R} \times \mathcal{O}} \dots \max_{a_{m_k} \in \mathcal{A}} \sum_{r_{m_k} o_{m_k} \in \mathcal{R} \times \mathcal{O}} (r_k + \dots + r_{m_k}) \cdot \xi(\dot{a}\dot{r}\dot{o}_{1:k-1} a\underline{ro}_{k:m_k}) \tag{1}$$

where $\mathcal{A}$ denotes the set of actions, $\mathcal{R}$ denotes the numerical set of rewards, and $\mathcal{O}$ denotes the set of observations. $m_k (\geq k)$ represents the extended lifetime: at time k only the sum of rewards $r_k + \dots + r_{m_k}$ until m_k is considered, and rewards farther in the future are ignored. $a \cdot b$ represents usual scalar multiplication of a and b, and juxtaposition represents tuples and sequences. $aro_{t:u} = a_t r_t o_t a_{t+1} r_{t+1} o_{t+1} \dots a_u r_u o_u$ denotes the time sequence of actions, rewards, and observations from time t until u. $\xi(\dot{a}\dot{r}\dot{o}_{1:k-1} a\underline{ro}_{k:m_k})$ is the probability of observing the sequence of rewards and observations $ro_{k:m_k}$, given the history $h_k = \dot{a}\dot{r}\dot{o}_{1:k-1}$ at time k and assuming to take actions $a_k \dots a_{m_k}$ from time k to m_k.

$$\xi(aro_{1:k-1} a\underline{ro}_{k:m_k}) = \xi(a\underline{ro}_{1:m_k}) / \xi(a\underline{ro}_{1:k-1}) \tag{2}$$

holds, where the denominator of the rhs at time k is constant and thus can be ignored. For ξ, AIXI uses the universal prior

$$\xi(a\underline{ro}_{1:t}) = \sum_{q \in \{q' | q'(a_{1:t}) = ro_{1:t}\}} 2^{-l(q)} \tag{3}$$

where q is the Turing machine computing the behavior of the environment, and $l(q)$ denotes the description length of q when represented in the binary form. In other words, $\xi(a\underline{ro}_{1:t})$ is the probability that q returns $ro_{1:t}$ for the input of $a_{1:t}$ when the Turing machine q is selected as a random sequence of bits.

Because (3) requires infinite computation, AIXI is not exactly computable. It is not an algorithm, but rather a theoretical framework for discussing the ideal intelligence.

AIXI is proved to be self-optimizing if there exist self-optimizing policies, i.e., if there exist policies that make the expected average reward converge to the optimal one for $m \to \infty$ for all environments, AIXI agents' average reward also converges to it in the same limit. [1] On the other hand, [8] shows that AIXI (and other greedy algorithms using fixed priors) do not converge to the optimal policy if the environment is programmed to suddenly change the reward for some action as the history grows. Although Hutter's self-optimizing theorem suggests that there is no deterministic self-optimizing policy in such a case, [8] argues that such environment is plausible in the real world, and suggests the need for non-greedy exploration strategies.

2.2 AIXI Approximations

AIXItl is a computable AIXI approximation that picks up the best program within length l and computable within time t. Its main focus is on being computable, and it still requires unrealistic computational complexity.

MC-AIXI(FAC-CTW)[10] is another, more efficient approximation of AIXI. It selects actions using the UCT algorithm [6] which is a kind of Monte-Carlo Tree Search, and uses a generalization of Context Tree Weighting (CTW) [11] for the environment model.

Veness et al. [10] applied MC-AIXI(FAC-CTW) to several toy problems and report better results than other related algorithms. They also applied it to *partially observable Pacman* (a.k.a. *Pocman*) which is much more challenging, and found some facts: MC-AIXI(FAC-CTW) reportedly learns to avoid walls, get food, and run away from ghosts; on the other hand, it does not learn to aggressively chase down ghosts after eating a power pill.

Veness et al. [10] point out that solving the exploration/exploitation trade-off is computationally intractable for MC-AIXI(FAC-CTW). However, learning to capture a ghost only from experience without any teaching would be difficult for *any* learning algorithm, because the ghosts flee and thus it should take time until the agent happens to capture one by chance for the first time. Without seeing other players play, being explained the rule of Pacman, or guessing from the paleness of ghosts, even human players would find difficulty in finding effectiveness of this behavior for themselves.

That being said, if the agent's policy is stochastically implemented as a distribution over programs, and if the universal prior is used to assign higher probability to short programs, the agent may try the compound action of "simply chasing pale ghosts" as exploration.

Another limitation of MC-AIXI(FAC-CTW) which is not mentioned by [10] is that it cannot generalize over sequences of observations because it uses CTW instead of learning Turing machines, unlike AIXI. CTW is essentially a finite map from finite sequences to frequencies implemented compactly using a PATRICIA tree, which means that it uses a tabular representation for modeling the environment.

Again, MC-AIXI(FAC-CTW) does not learn to chase down ghosts *aggressively* after eating the power pill. If it generalized over sequences of observations, it would flee from ghosts, even after eating the pill.

3 Our Idea: Reinforcement Learning Algorithm that Learns to Program

This section describes the RL algorithm we plan to implement in details. The algorithm is based on the following reflections on AIXI and MC-AIXI(FAC-CTW) mentioned in Sect. 2:

- since generalization is necessary in order to solve real world problems, the environment should be modeled as a mixture of programs in a similar (but efficient) way as AIXI;

- the agents should be able to be taught (rules of games, etc.); one way is by words, and another way is by letting them imitate what others do; we consider the latter in this paper because the former looks difficult;
- the exploration/exploitation trade-off could be dealt with by explicitly enabling exploratory behavior; further, designing the policy stochastically as a distribution over multiple programs may enable exploration by compound actions such as chasing pale ghosts in Pacman.

3.1 Environment Model

Since generalization over observations is necessary, we model the environment as a mixture of programs, like AIXI and unlike MC-AIXI(FAC-CTW). However, AIXI is not computable and AIXI*tl* requires a large amount of computation.

Our idea is to let MagicHaskeller [3,4][1] enumerate programs within some fixed length and use them as the candidates for the environment. Also, in order to preserve the possibility of generating longer programs, the *component function library* $\mathcal{L}$ that is used to generate programs should be learned incrementally [5].

MagicHaskeller is an inductive functional programming system based on generate-and-test. Inductive functional programming is automated functional programming from ambiguous specifications such as input-output examples and properties that the resulting program should satisfy.

In fact MagicHaskeller's program generator is similar to the environment model of AIXI in that MagicHaskeller generates stream of programs exhaustively from the shortest increasing the length, and tests them against the given examples. On the other hand, they differ in the following ways:

- MagicHaskeller generates Haskell programs, or typed λ expressions, instead of Turing machines, and
- it keeps those programs in its memorization table, and it saves the size of the table by not generating expressions that are obviously semantically equivalent to already-generated expressions and by shelving generation of expressions that may be equivalent until its difference from other expressions is proved.

Learning More and More Complicated Environment. Now we discuss how to update the component library $\mathcal{L}$.

If we apply the idea of AIXI straightforwardly, the true environment is one big (but modularized) program that is consistent with all of the input/output history. An AIXI agent holds (infinitely) many candidate programs with different plausibilities as theories explaining the true environment. This means that although there are many programs, incremental learning by adding expressions often appearing there (considering that they are useful) to the component library $\mathcal{L}$ may make only a little sense, if ever, because only one program out of them is supposed to explain the truth.

[1] http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html.

However, we need not stick to the AIXI way strictly, but we may relax the above consistency rule. Actually, since there are many programs, there can be Program A that explains the state transition in situation x but cannot explain that in situation y, and Program B that explains that in y but cannot explain that in x at the same time. We think that the plausibility of each program depends on the situation. By admitting this and permitting partially satisfying programs, we can think of incremental learning that adds useful functions in some situations, which can result in a program which is plausible in many situations by conditioning on them.

$\mathcal{L}$ can be updated by adding expressions frequently appearing in plausible environment programs. More exactly, $\mathcal{L}$ should be chosen to make the total length of environment programs weighted with the plausibility as short as possible.

3.2 Action Selection

In (1) that defines AIXI, the other part that needs to be approximated for complexity reasons is the expectimax operations that select the actions at each time step. Also, as written at the beginning of Sect. 3, we want to enable exploratory compound actions by implementing the policy as a distribution over multiple programs. Using exploratory compound actions can have an advantage over stepwise exploration strategies such as ϵ-greedy which can mess up everything by inconsistent exploratory actions.

A simple way to implement it as a set of programs is to use $\arg\max_{p\in\mathcal{P}}$ instead of the expectimax operation in (1) by using some set of programs $\mathcal{P}$:

$$p_k = \arg\max_{p\in\mathcal{P}} ... \tag{4}$$

$$\dot{a}_k = p_k(\dot{a}\dot{r}\dot{o}_{1:k-1}) \tag{5}$$

However, this is not very desirable, because

- this requires computation of $\arg\max_p$ at every time step, and
- $\arg\max_p$ is too arbitrary, because p takes the history as the argument, and nothing but $\mathcal{P}$ restricts the return values for not experienced histories.

Thus, we will use a common policy p not depending on the time step. p should be updated from time to time, which means we assume episodic (or factorizable) environment. More concretely, $p \in \mathcal{P}$ such that for all history $h \in (\mathcal{A}\times\mathcal{R}\times\mathcal{O})*$

$$V^{p\xi}(h) = \max_{p'\in\mathcal{P}} V^{p'\xi}(h) \tag{6}$$

holds should asymptotically be found, where $V^{p\xi}(h)$ denotes the expected sum of the total reward after history h from time k to time m_k assuming that the agent will follow the policy p under the environment ξ, and k is the next time after history h finishes. We conjecture such V exists for any set of computable functions $\mathcal{P}$.

One way to find such V asymptotically is to keep the L^2 norm between the best estimation of V and the current estimation of V

$$\sum_h (n_h(\max_{p'} \hat{V}^{p'}(h) - \hat{V}^p(h))^2)$$

for each p, and choose the minimal p either deterministically or stochastically allowing exploration by using, e.g. the roulette selection, where $\hat{V}$ denotes the estimation of V, and n_h denotes the number of visitations to h.

When learning a value function asymptotically, it is also important what values should be assigned to those arguments that have not been used. In the field of RL, simply assigning big values for those unvisited arguments in order to encourage exploration is known to be effective (e.g. Optimistic Initial Values [9] and UCT [6]). However, this will not work for infinite or very big finite $\mathcal{P}$, which has (infinitely) many actions to be tried.

Even when selecting p, programs with less Kolmogorov complexity should be prioritized more based on the idea of Occam's razor in the same way as the environment case. Then, the universal probability (or something similar) should be used as the prior. (In practice, the selection may not necessarily be stochastic, and can be something like "When the number of program selections already done exceeds the reciprocal amount of the sum of the priors of programs that have not been tried, try the shortest one among them.") A good news compared to the environment case is less computational complexity, because the algorithm need not compute the expectation but only choose one program.

Now the question is what should this $\mathcal{P}$ be like. Our idea is to let MagicHaskeller enumerate $\mathcal{P}$, too. This means $\mathcal{P}$ is the enumeration of all the programs within some given length, using functions in the component library $\mathcal{L}'$. AIXI requires to find the maximum of $|\mathcal{A}|^{m_k}$ cases, but by using $\arg\max_{p\in\mathcal{P}}$ our algorithm will need to find the maximum of $|\mathcal{P}|$ cases.

Imitation. How should $\mathcal{L}'$ be learned? Although $\mathcal{L}$ could be learned to find the minimal description of the environment, this way cannot be applied to the case of learning $\mathcal{L}'$ that is the library for defining policies, because policy learning is unsupervised.

Our solution to this question is to let $\mathcal{L}' = \mathcal{L} + \mathcal{C}$ where $\mathcal{C}$ is a constant set of primitive actions. This means, "construct your behavior using what you see, i.e., by imitation", because $\mathcal{L}$ should be filled with functions for describing the behavior of the environment in short. In this way, we think, the question of how to implement imitation can be solved at the same time. On the other hand, we should note that with this approach it is difficult for an agent to imitate instantly what it has just seen, because the agent has to wait until it is added to the component library $\mathcal{L}$, and we think it is difficult for this to happen within the same episode.

One question is whether we should consider the amount of reward when registering functions to $\mathcal{L}$, or whether we should give functions used in reward-earning policies more chance to be registered to $\mathcal{L}$. We think that this need not to be done, because the reward-earning policies and resulting reward-earning states will be tried many times anyway, and if otherwise, it is not clear whether the reward is really due to the policy or not.

4 Evaluation and Applications

Most of our ideas are inspired from the experience of playing Pocman in [10]. In order to make sure that newly introduced functionalities are doing what they are supposed to do, evaluation by Pocman is necessary.

Also, in order to make sure that we are not going backward, several other easier problems shown in [10] should also be tried.

Pocman is a partially observable version of Pacman, where cells far from Pacman are invisible. Also, observations in Pocman are made relative to the position of Pacman. Although partially observable problems are more difficult than fully observable ones in general, those changes may have made the game easier, by hiding unimportant information. Because real people play the original Pacman, it would also be interesting to evaluate the agent under the original one.

A more challenging task may be multiplayer games such as Doom, where the effect of introducing imitation can be evaluated. Our algorithm will use MagicHaskeller for the policy and the environment, and because MagicHaskeller generates functional programs by combining functions in the component library, by including complicated functions implementing Narrow AI in the component library the functionality will be made available. This way, our algorithm can focus on decision making, while using existing technology for e.g. image recognition.

Although Deep Q Network [7] that is a monolithic algorithm dealing with decision making and vision processing at the same time holds the spotlight in recent years, human brains make decisions and process perceptions separately. This modular approach may be better for realizing higher-level artificial intelligence.

5 Conclusions

This paper presented our research idea for an RL algorithm that models both the environment and the policy as a distribution over Haskell programs. The algorithm will also implement imitation, and hopefully, can deal with the exploration/exploitation trade-off in a better way than conventional approaches.

Haskell is an artificial, general-purpose programming language, and thus it is unlikely that human brains are actually planning their behavior in Haskell. However, λ calculus, which is the model of computation on which the Haskell language is built, can be a good option for modeling consciousness. We think that consciousness is the functionality to make the thinking process object of thinking and communication. If this is correct, λ calculus, which is designed to make functions object of computation, can be the best tool for explaining consciousness and discussing it.

Acknowledgements. The author thanks anonymous reviewers who helped improving the paper, especially who mentioned [8].

References

1. Hutter, M.: Self-optimizing and pareto-optimal policies in general environments based on bayes-mixtures. In: Kivinen, J., Sloan, R.H. (eds.) COLT 2002. LNCS (LNAI), vol. 2375, pp. 364–379. Springer, Heidelberg (2002). http://dx.doi.org/10.1007/3-540-45435-7_25
2. Hutter, M.: Universal algorithmic intelligence: a mathematical top $\rightarrow$ down approach. In: Goertzel, B., Pennachin, C. (eds.) Artificial General Intelligence. Cognitive Technologies, pp. 227–290. Springer, Heidelberg (2007). http://www.hutter1.net/ai/aixigentle.htm
3. Katayama, S.: Systematic search for lambda expressions. In: Sixth Symposium on Trends in Functional Programming, pp. 195–205 (2005)
4. Katayama, S.: Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In: Ho, T.B., Zhou, Z.H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 199–210. Springer, Heidelberg (2008)
5. Katayama, S.: Towards human-level inductive functional programming. In: Bieger, J., Goertzel, B., Potapov, A. (eds.) AGI 2015. LNCS, vol. 9205, pp. 111–120. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-21365-1_12
6. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
7. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
8. Orseau, L.: Optimality issues of universal greedy agents with static priors. In: Hutter, M., Stephan, F., Vovk, V., Zeugmann, T. (eds.) Algorithmic Learning Theory. LNCS, vol. 6331, pp. 345–359. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-16108-7_28
9. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning, 1st edn. MIT Press, Cambridge (1998)
10. Veness, J., Ng, K.S., Hutter, M., Uther, W., Silver, D.: A Monte-Carlo AIXI approximation. J. Artif. Intell. Res. **40**, 95–142 (2011)
11. Willems, F.M.J., Shtarkov, Y.M., Tjalkens, T.J.: The context tree weighting method: basic properties. IEEE Trans. Inf. Theor. **41**, 653–664 (1995)