Learning to Focus Reasoning in a Reactive Agent

Nima Asgharbeigi and Negin Nejati
Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA
{nimaa, negin}@stanford.edu

Abstract

In this paper we present an online learning algorithm that lets an agent architecture acquire an attentional strategy that is adapted to the environment. This allows the agent to focus its reasoning on the most rewarding parts of its knowledge base and hence achieve a higher performance under time and computational resource constraints. We use the Icarus agent architecture as an underlying framework and we present experimental results.

Introduction

In order to decide on its next action in an environment, an intelligent agent must reason about that environment. In complex domains, the reasoning phase can be computationally very expensive if the entire knowledge base is processed on every step. This leads to a major bottleneck on the performance of reactive agents, which are naturally time-constrained in real-world applications. Therefore, one needs mechanisms to direct the agent’s attention to the most rewarding subset of the knowledge base.

To predict this subset of the knowledge base, most previous work has relied on manual specification of conditions and priorities into the agent’s knowledge base. See, for example, Carbonell et al. (1991), Rosenbloom et al. (1993), Choi et al. (2004), and Wray et al. (1994). These methods are not only time consuming to implement but also subjective and prone to error. In contrast, a learning methodology gives the agent the opportunity to autonomously acquire an attention scheme adapted to the environment. In this approach, the knowledge base is augmented with a predictive model that is learned from the rewards provided by the agent’s belief state.

Architectural Framework

Our approach builds on Icarus (Choi et al., 2004), a reactive agent architecture that supports reasoning and decision making. Its infrastructure represents knowledge in terms of concepts and skills. Concepts are Boolean and describe situations in the environment and skills describe how to respond to these situations. Icarus includes separate memories and processes for concepts and skills and for each of these has a long-term and a short-term memory. The long-term memories store the agent’s knowledge about a domain content in terms of concept/skill definitions, whereas short-term memories store the dynamic beliefs and intentions. Icarus also supports a third short-term memory called the perceptual buffer, which contains the outputs of the sensors that capture the characteristics of physical entities.

The long-term conceptual memory encodes familiar objects, relations, and situations in the form of concept definitions. Each concept has a name and arguments and is defined hierarchically in terms of lower-level concepts, sensory percepts, and arithmetic tests on them. More formally, it can have five optional fields: :percepts (the perceived entities), :positives (the lower-level concepts that must match), :negatives (the lower-level concepts that must not match), :tests (the numeric relations that must hold), and :reward (the internal reward function for the matched concept).

As an example, Table 1 exhibits two concept definitions from a driving domain.

stances of the concept definitions in long-term conceptual memory that can be inferred from the perceptual buffer. Closely related to concepts, skills constitute the second part of the knowledge base of an Icarus agent. Each skill specifies a set of sub-skills or actions to be executed in order to achieve a set of objective concepts, from a set of start conditions represented in terms of percepts and concepts. Similar to the concepts, there is a reward function associated with each skill which provides an internal source of reward that comes from the agent’s decisions.

Icarus operates in cycles. In its previous version, the architecture refreshes the contents of its perceptual buffer at the beginning of each cycle. Icarus continues by inferring all the matched instances of concepts in the hierarchy in a breadth-first, bottom-up manner. Finally, based on the belief state composed of the concepts in short-term memory, the agent finds all the applicable skills and selects the skill with highest utility to execute.

In the modified architecture that we describe here, the matching procedure (binding the concept variables to the existing objects) has been separated from the inference procedure (verifying if the concept instance is true or not). In the new method, all the concepts are instantiated but only the most important ones are inferred. The learning algorithm’s task is to capture the importance of these instances by estimating their values.

**Details of the Learning Algorithm**

Our approach involves an online learning algorithm that consists of two mechanisms, a reinforcement learning method and a generalization technique. The first mechanism uses attention-relevant values assigned to instances to determine the most rewarding subset. The second algorithm generalizes the instance-specific values that result from the reinforcement learning algorithm to value functions for their corresponding concept definitions.

**Reinforcement Learning Mechanism**

More formally, let \( \mathcal{U} \) be the set of all concept instances. We want to find the subset \( s \subseteq \mathcal{U} \), under time constraint, so that the accumulative reward, \( \sum_{u \in s} R_u \), is maximized. \( r_u \) is the reward given by the concept’s reward function if \( u \) is true and 0 otherwise.

In order to achieve this goal, the reinforcement learning algorithm tries to learn values \( V : \mathcal{U} \mapsto \mathbb{R} \) over the current set of concept instances \( \mathcal{U} \). We define the state \( s \) as the set of true concept instances inferred so far within the current cycle. We also define the fringe, denoted by \( F_s \), at any given state \( s \), to be the set of all concept instances that have not been inferred in the current cycle yet but whose children have all been inferred. A valid action \( a \) is the action to infer a single instance \( u_a \) from the fringe \( F_s \).

Notice that the state space \( S \), containing all possible states \( s \) can be extremely large. Therefore, using classical representation of \( Q(s,a) \) for Q-learning or \( V(s) \) for value learning is impractical. This necessitates a more compact way of representing our learned knowledge to make the problem tractable. Along similar (but not exactly the same) lines as in (Dietterich, 2000), we use the idea of value decomposition to represent the \( Q \)-function in terms of \( V \)-values:

\[
Q(s,a) = \sum_{u \in s} V(u) + V(u_a). \tag{1}
\]

Using equation (1) in the standard \( Q \)-function stochastic update rule, \( Q(s,a) := (1-\alpha)Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a')] \), we obtain the following \( V \)-value update rule:

\[
V(u_a) := (1-\alpha)V(u_a) + \alpha[R(s,a) + \gamma \max_{u \in F_s} V(u')], \tag{2}
\]

where \( \alpha \) is defined by:

\[
\alpha = \frac{1}{1 + \text{visits}(u_a)}, \tag{3}
\]

and \( \text{visits}(u_a) \) is the number of times \( V(u_a) \) has been updated.

The algorithm starts at the beginning of each cycle with a null state \( s = \varnothing \) and performs action selection and value iteration update until it runs out of time. At each iteration it selects the instance \( u \) with highest value in the fringe to infer. Then the instance is inferred and the state and fringe are updated accordingly. The algorithm also computes the reward \( r_{u} \) of the instance and uses it as the reward in the \( V \)-value update rule:

\[
V(u) := (1-\alpha)V(u) + \alpha[r_u + \gamma \max_{u \in F_s} V(u')], \tag{4}
\]

where \( F^u_s \subseteq F_s \) is the set of instances in the updated fringe for which \( u \) is a child, and \( \alpha \) is defined by (3).

Notice that the only difference between the update rules in (2) and (4) is that in (4) we have restricted the argument of \( \max \) to the set of instances that build directly on top of \( u \). This is because we think that the values of instances that can be built on top of \( u \) are more indicative of the quality of the action to infer \( u \), compared to the rest of the instance space.

Therefore we expect that the update rule (4) will perform better.

**Generalization Mechanism**

The generalization algorithm uses linear regression methods to incrementally update a linear fit \( h_c(x) = \theta^T x \) to the training examples \( S_c = (\{V(u),x(u)\} \mid u \in \mathcal{U}_c \cap s) \), for each concept definition \( c \). Here \( x(u) \) denotes the vector of attributes of the perceptions that appear in concept instance \( u \), and \( \mathcal{U}_c \subseteq \mathcal{U} \) is the set of all instances derived from concept definition \( c \).

From the implementation point of view, we are keeping the set of all the concept instantiations \( \mathcal{U} \) in memory and we update it whenever a new perception is added or an old one is deleted. We initialize the \( V \)-values for new concept instances by evaluating their corresponding \( h_c \) functions.
environment such as distribution, complexity, and dynamics. This allows us to study the behavior of the algorithm under
dermanent or a Gauss-Markov process with \( \sigma \) and a random initial point. For a fully dynamic environ-
ment of the overall algorithm is presented in Table 2.

Intuitively, our reinforcement learning mechanism captures
ter assistant so that the accumulative reward is maxi-
mized. In contrast, our generalization algorithm aims to find
how much each concept instance is worth, not only based on

In order to evaluate the behavior of our algorithm, we per-
formed experiments with a simplified abstract environment.
This allows us to study the behavior of the algorithm under
different circumstances by controlling the properties of the
environment such as distribution, complexity, and dynamics.

Our abstract environment consists of \( k \) percepts at any
cycle, each with five attributes and each attribute being ei-
ther constant or a Gauss-Markov process with \( \sigma^2 = 0.01 \)
and a random initial point. For a fully dynamic environ-
ment we chose \( k \) to evolve as a symmetric random walk with probabilities of going up and down each being 0.1. We also specify two concept definitions that generate \( k^2 \) and \( k^3 \) instantiations respectively.

### Experimental Evaluation

In order to evaluate the behavior of our algorithm, we per-
formed experiments with a simplified abstract environment.
This allows us to study the behavior of the algorithm under
different circumstances by controlling the properties of the
environment such as distribution, complexity, and dynamics.

### Sparse Instance Space

For the first set of experiments, we defined the concepts so
that only a small fraction of instantiations infer to true at any
cycle. Figure 1(top) shows the behavior of the algo-
rithm in a static environment in which there are \( k = 10 \)
percepts and only 18 instances are true out of 1100 instan-
tiations. The dotted line represents the total reward that the
environment offers, and the solid curve indicates the amount
of reward that the system obtains at each cycle from inferred
instances. This plot can be viewed as a learning curve for
our algorithm.

Figure 1(bottom) exhibits the results of a similar exper-
iment in a fully dynamic environment in which the num-
ber of true instances varies between 4 and 31. We observe
that the inferred reward tracks the total reward fairly well.
In a similar environment, we performed a comparison be-
tween behaviors of learning and non-learning systems. Fig-

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Reward</th>
<th>Inferred</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Performance of the algorithm in static (top) and
dynamic (bottom) environments with sparse instance space.
Figure 2: Comparison between learning and non-learning systems over cycles. As expected, the performance of our algorithm dominates that of the non-learning system.

In order to make a more reliable comparison between the learning and the non-learning systems, we measured the performance of both systems in terms of their average percentage of error over 100 cycles for different values of time limit decreasing from 0.08 to 0.005 second. Also, for each time limit, we averaged the results over 100 independent runs. Figure 3 presents the average error percentages together with 95% confidence intervals. Evidently, our learning system is less affected by time constraints than the non-learning system.

Dense Instance Space

Now we consider a more challenging environment in which a large number of instances are true at every cycle, but only a few instances offer significant reward. Figure 4 shows a sample performance of our algorithm for an environment in which the number of percepts was varying between 8 and 15, and an average 48% of all instances were true at any cycle. Most of these instance were low-reward and only 2 to 6 instances were offering the major contribution to the total reward.

Apparently our algorithm has difficulty in finding the most rewarding instances. This is because, as an online learning algorithm, it chooses its actions in a greedy manner. Therefore, instead of exploration, the system spends its time at every cycle inferring the true but low-reward instances it has found so far.

The easiest way to mitigate this problem is to use an \( \alpha \)-greedy action selection mechanism. We implemented a variant of \( \alpha \)-greedy algorithm in which the system selects a random instance from the set of instances that have never been inferred before (if any) with probability \( \alpha \) and infers it. Figure 5 demonstrates the average performance of this algorithm averaged over 40 runs for each value of \( \alpha \) with each run being 100 cycles long. In this example, the best value for \( \alpha \) seems to 0.4, which offers about 10% improvement compared to the purely greedy approach (\( \alpha = 0 \)). A more promising approach would be the idea of probabilistically persistent belief that we will introduce in the next section.

The experiments presented so far were not incorporating the generalization mechanism, that is, the initial \( V \)-value for new instances were being set to zero. The dashed curve in Figure 4 exhibits the performance of a system that utilizes the generalization mechanism to learn linear models and then uses those models to initialize the \( V \)-values for new instances that are born after cycle 50. This evidently shows a slight improvement in performance for cycles after the 50th cycle. The degradation of performance for cycles before this point is caused by the cost of generalization mechanism.

Our experiments reveal that the generalization mechanism is not as effective as expected. We realized that this is because the generalization mechanism is supposed to capture not only the values but also the likelihood of instances to hold true. Such modelling can be achieved by a combination of logistic and linear regressions, but not by linear regression alone. The logistic regression would try to capture the likelihood of an instance being true (given that all its children hold true) while the linear regression would capture the value of the instance. Therefore, the multiplication of the two would be a better initial approximation for the value of a new instance.

Related Work

The challenge of reasoning under resource constraints is almost as old as the interest in developing intelligent systems that behave reasonably in complex domains. This problem has been neglected in classical theories of normative be-
behavior. As a result, the common approach to mitigate the problem in practical intelligent systems has been to employ heuristic and normally domain-dependent control strategies to guide the reasoning. For example, the MRS (Meta-level Representation System) developed by Genesereth and Ginsberg (1985) enables the designer to write PROLOG-like control clauses that define how the domain-content clauses should be used by the agent.

Early notions of bounded rationality had focused on the discovery of satisficing strategies for problem solving. Horvitz (1989) discussed limitations of the normative approach in dealing with problems of real-world complexity and instead proposed to utilize the basis of normative rationality to reason about the reasoning process in problem solving. Along similar directions, Russell and Wefald (1989) have sought to develop a theoretical framework of metareasoning based on probability and decision theory.

In their analysis, Russell and Wefald considered computations as actions, which is the same basic insight employed in this work. However, we combined that idea with a learning approach in pursuit for a framework that promises the benefits of a developmental approach. In doing so, we borrowed ideas from relational reinforcement learning (Dzeroski et al., 1998) and hierarchical reinforcement learning (Dietterich, 2000). In contrast to most previous work on reinforcement learning, our notion of states and actions are completely internal, concerning the reasoning process of the system.

Concluding Remarks

Although we obtained encouraging results in one set of experiments, we observed a some weaknesses in our algorithm that should be addressed before exposing the system to real-world environments.

References


