Applied Artificial Intelligence, 20:35–55
Copyright © 2006 Taylor & Francis Inc.
ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839510500191299

Taylor & Francis
Taylor & Francis Group

APPLICATION OF EVOLUTIONARY COMPUTATION FOR EFFICIENT REINFORCEMENT LEARNING

Genci Capi □ *Faculty of Information Engineering, Fukuoka Institute of Technology, Higashi-ku, Fukuoka, Japan*

Kenji Doya □ *CREST, Japan Science and Technology Agency (JST), Computational Neuroscience Laboratories, Kyoto, Japan*

□ *In this paper, we propose a new method based on evolutionary computation for setting the metaparameters of reinforcement learning in order to match the demands of the task and reduce the learning time. In our method, we encode the metaparameters as the agent's genes and take the metaparameters of best-performing agents in the next generation. We investigate the influence of metaparameters on the agent learned policy and learning time. The results show that appropriate settings of metaparameters found by evolution have a great effect on the learning time and are strongly dependent on each other. In addition, by using the Cyber Rodent robot, we verified that metaparameters evolved in simulation are helpful for learning in real hardware.*

Reinforcement learning (RL) (Barto 1995; Doya 2000; Doya et al. 2001; Sutton and Barto 1998) is the process of learning the coordination of concurrent behaviors and their timing so as to optimize some performance cost. In RL, learning is contingent upon a scalar reinforcement signal, which provides evaluative information about how good an action is in a certain situation, without providing an instructive supervising cue as to which would be the preferred behavior in the situation. Behavioral research indicates that RL is a fundamental means by which experience changes behavior in both vertebrates and invertebrates (Donahoe 1997). In addition, RL has been successfully applied to a variety of dynamic optimization problems, such as game programs (Tesauro 1994), and resource allocation (Singh and Bertsekas 1997).

In RL, the learning capabilities are strongly dependent on a number of parameters, such as learning rate, the "temperature" of exploration, and the time scale of evaluation. These parameters are often called

Address correspondence to Genci Capi, Faculty of Information Engineering, Fukuoka Institute of Technology, 3-30-1 Wajiro Higashi, Higashi-ku, Fukuoka 811-0195, Japan. E-mail: capi@fit.ac.jp

metaparameters because they regulate the way detailed parameters of an adaptive system change with learning. The permissible ranges of such metaparameters depend on particular task and environment, making it necessary for a human expert to tune them usually by trial and error. But tuning multiple metaparameters is quite difficult due to their mutual dependency, e.g., if one changes the noise size, one should also change the learning rate. In addition, hand tuning of metaparameters is in a marked contrast with learning in animals, which can adjust themselves to unpredicted environments without any help from a supervisor.

In statistical learning theory, the need for setting the right metaparameters, such as the degree of freedom of statistical models and the prior distribution of parameters, is widely recognized. Theories of metaparameter setting have been developed from the viewpoints of risk-minimization (Vapnik 2000) and Bayesian estimation (Neal 1996). However, many applications of reinforcement learning have depended on heuristic search for setting the right metaparameters by human experts. In this paper, we apply evolutionary computation to set metaparameters based on the agent performance and learning time. In our method, the metaparameters of the RL algorithm are encoded as the agent's genes, and the metaparameters of best-performing agents are taken in the next generation.

Learning and evolution are complementary mechanisms for acquiring adaptive behaviors, within the lifetime of an agent and over generations of agents. There is much research on the relation between evolution and learning, such as the Baldwin effect (Baldwin 1896; French and Messinger 1994) and the combining of artificial neural network and genetic algorithm (Belew et al. 1992; Unemi 1994).

The specific questions we ask in this study are: 1) whether GA can successfully find appropriate metaparameters subject to mutual dependency, 2) how the metaparameters and initial weight connections affect the learning time, and 3) whether the metaparameters optimized by GA in simulation can indeed be helpful in hardware implementation of RL. In order to answer these questions, first we considered a battery capturing task for the Cyber Rodent (CR) robot (Capi et al. 2002). The mutual dependency among metaparameters is difficult to investigate when more than two metaparameters are optimized. For this reason, evolution considered two metaparameters: the learning rate and cooling factor of a Sarsa(λ) RL algorithm. Then, the learned capturing behavior is utilized in a survival behavior, where the robot must recharge itself by capturing active battery packs distributed on the environment. In this environment setup, our main focus is on how metaparameters influence the learning time. In addition of learning rate and cooling factor, the discount factor and initial weight connections are encoded in the genome.

Figure 2. The positions of battery packs are considered fixed in the environment and the CR robot is initially placed in a random position and orientation.

The agent can recharge its own battery by capturing active battery packs, which are indicated by a red LED color. After the robot captures the battery pack, it can recharge its own battery for a determined period of time (charging time), then the battery pack becomes inactive and its LED color changes to green. The battery becomes active again after the reactivation time. Therefore, in this environment the following parameters can vary:

- The number of battery packs.
- The reactivation time.
- The energy received by capturing the battery pack (by changing the charging time).
- The energy consumed by the agent for 1 m motion.

Based on the energy level and the distance to the nearest active battery pack, the agent can select among three actions: 1) capture the active battery pack, 2) search for a battery pack, or 3) wait until a battery pack becomes active. In the simulated environment, the batteries have a long reactivation time. In addition, the energy consumed for 1 m motion is low. Therefore, the best policy is to capture any visible battery pack (the nearest when there are more than one). When there is no visible active battery pack, the agent must search in the environment.

REINFORCEMENT LEARNING

Sarsa (λ)

The agent learns the battery capturing behavior based on *gradient decent Sarsa (λ)* RL algorithm (Sutton and Barto 1998). Based on the visual information, the agent determines the angle to the closest battery pack. Thus, the state s of the agent is the angle of the nearest visible battery pack, which varies from $-\pi/2$ to $\pi/2$. The state space is explored using a softmax method, employing the Boltzmann distribution.

To represent the action value function, we use a radial basis function network. The Gaussian functions are used as basis functions given as:

$$\phi(i, s) = e^{-\frac{\|s-c_i\|^2}{2\sigma_i^2}}, \tag{1}$$

TASKS AND ENVIRONMENTS

Battery Capturing Behavior

In order to investigate the effect of metaparameters on the agent-learned behavior and to know the interaction between them, we considered a capturing task where the CR robot has to capture the battery packs distributed in the environment. This task requires exploration in the environment and learning from delayed reward during limited lifetime. Thus, the settings of metaparameters, such as the learning rate and the randomness in action selection, critically affect the agent's performance.

In the first experiment, the agent is placed in a square environment of size 3.0 m × 3.0 m, containing a single battery pack. At each time step the agent receives the angle to the nearest battery pack as state input. The vision range is assumed to be between $-\pi/2$ and $\pi/2$. The agent starts learning in short distance relative to the battery pack, oriented so that the battery pack is always visible. In the late stage of learning, the position and orientation of the agent relative to the battery pack are randomly selected.

In addition, we considered the case when there are 15 battery packs, which decrease linearly from 15 to 1 during the agent's lifetime. The agent searches without getting repositioned. An obstacle avoidance behavior is generated when the agent gets a distance of 250 mm from walls. This allows us to study learning and metaparameter settings under more complex environmental conditions.

Surviving Behavior

In the second environment, the CR robot has to survive and increase its energy level. The environment has 8 battery packs, as shown in

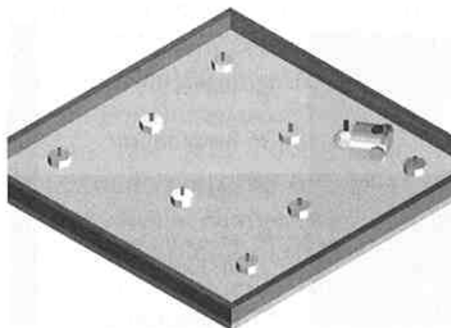

FIGURE 2 Environment of surviving behavior.

Figure 2. The positions of battery packs are considered fixed in the environment and the CR robot is initially placed in a random position and orientation.

The agent can recharge its own battery by capturing active battery packs, which are indicated by a red LED color. After the robot captures the battery pack, it can recharge its own battery for a determined period of time (charging time), then the battery pack becomes inactive and its LED color changes to green. The battery becomes active again after the reactivation time. Therefore, in this environment the following parameters can vary:

- The number of battery packs.
- The reactivation time.
- The energy received by capturing the battery pack (by changing the charging time).
- The energy consumed by the agent for 1 m motion.

Based on the energy level and the distance to the nearest active battery pack, the agent can select among three actions: 1) capture the active battery pack, 2) search for a battery pack, or 3) wait until a battery pack becomes active. In the simulated environment, the batteries have a long reactivation time. In addition, the energy consumed for 1 m motion is low. Therefore, the best policy is to capture any visible battery pack (the nearest when there are more than one). When there is no visible active battery pack, the agent must search in the environment.

REINFORCEMENT LEARNING

Sarsa (λ)

The agent learns the battery capturing behavior based on *gradient decent Sarsa (λ)* RL algorithm (Sutton and Barto 1998). Based on the visual information, the agent determines the angle to the closest battery pack. Thus, the state s of the agent is the angle of the nearest visible battery pack, which varies from $-\pi/2$ to $\pi/2$. The state space is explored using a softmax method, employing the Boltzmann distribution.

To represent the action value function, we use a radial basis function network. The Gaussian functions are used as basis functions given as:

$$\phi(i, s) = e^{-\frac{\|s-c_i\|^2}{2\sigma_i^2}}, \qquad (1)$$

where c_i is the center and σ_i is the width of the ith basis function. It follows that action value function is given by:

$$Q(s, a) = \sum_{i=1}^{n} \phi(i, s)\theta(i, a), \tag{2}$$

where $\theta(i, a)$ is the weight value for basis function i and action a. The Gaussian basis functions are uniformly distributed over one-dimensional input space. Furthermore, we used two binary basis functions to represent the situation when the battery packs get out of the agent's sight field: one for losing the battery pack to the left and one for losing to the right. The gradient descent updates the weight matrix for action value prediction as follows:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t, \tag{3}$$

where $\vec{\theta}_t$ are the network connection weights, α the learning rate, δ_t is the temporal difference error, and $\vec{e}_t$ is the trace matrix distributing δ_t over the states recently visited. δ_t and $\vec{e}_t$ are calculated as follows:

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t), \\ \vec{e}_t &= \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t), \end{aligned} \tag{4}$$

where γ and λ are, respectively, the discount factor and trace decreasing factor. α is the step size by which the incremental update rule approaches the estimation of the state action value function. The reward is given by:

$$r = \begin{cases} 0 & |s| > 0.2\pi \\ \frac{0.1}{\pi}(0.2\pi - |s|) & |s| \leq 0.2\pi \\ 1 & \textit{reaching battery} \end{cases}. \tag{5}$$

The main reward is given when the agent captures the battery pack. Furthermore, a small supplementary reward is given when a battery is visible in front of the agent.

The agent chooses an action $a \in 1, 2, \ldots, A$, where $A = 7$ is the number of possible actions. The wheel velocities are given by:

$$\begin{aligned} \omega_{lefWheel}(a) &= \omega_{Const} * \frac{a-1}{A-1}, \\ \omega_{RightWheel}(a) &= \omega_{Const} * \frac{A-a}{A-1}, \end{aligned} \tag{6}$$

where ω_{Const} is the constant angular velocity. In this way, the action spans from turning left to turning right. The probability of choosing an action

a at time t is given as:

$$P(a|s_t) = \frac{e^{Q_t(a,S_t)/\tau}}{\sum_{i=1}^{A} \left(e^{Q_t(a_i,S_t)/\tau}\right)}, \tag{7}$$

where τ is the *temperature* of the algorithm. When the temperature is high ($\lim_{\tau \longrightarrow \infty}$), the probability of choosing any action is equal. As the temperature decreases, the probability of choosing an action at time t is proportional to the action's Q-value. When the temperature approaches zero, the action with the highest Q-value is selected. After each action taken by the agent, the temperature is decreased exponentially:

$$\tau_{t+1} = \tau_{df}\tau_t, \tag{8}$$

where τ_{df} is the temperature decreasing factor. The RL performance is strongly related to the values of $\vec{\theta}_{initial}, \tau_{initial}, \tau_{df}, \alpha, \gamma, \lambda$, and the total number of steps. In our work, we considered the influence of the metaparameters α and τ_{df}. The values of other parameters are considered as follows:

$$\begin{aligned} \vec{\theta}_{initial} &\in [0, 0.05], \\ \tau_{initial} &= 10, \\ \gamma &= 1, \\ \lambda &= 0.1. \end{aligned} \tag{9}$$

Actor-Critic RL

In the actor-critic used in the surviving behavior, a computational unit, called the actor, continually produces actions. While it does so, a second computational unit, called the critic, continually criticizes the action taken. The actor adapts its action choices using the critic's information. The critic also adapts in the light of the changing actor. The critic's role is as a go-between, between the actions on the one hand, and the reward information on the other.

Our implementation of the actor-critic has three parts: 1) an input layer of agent state, 2) a critic network that learns appropriate weights from the state to enable it to output information about the value of particular state, and 3) an actor network that learns the appropriate weights from the state, which enables it to represent the action the agent should make in a particular state. The agent can selects among three actions: 1) capture the active battery pack, 2) search for an active battery pack, and 3) wait for a determined period of time. The wait behavior is interrupted if a battery becomes active or after a pre-determined period of time. Both networks (Figure 3 and Figure 4) receive as input a constant bias input, the CR battery level