Implementation: Learning Policies with External Memory

Robert Moss
Stanford University, Stanford, CA 94305
MOSSR@STANFORD.EDU
AA229/CS239

1 *Introduction*

This work introduces the idea of using *stigmergy*[1] when learning online policies in partially observable domains. Their idea was to incorporate setting and clearing external memory bits into the action-space to learn memoryless online policies. Thus, continuing to learn a memoryless mapping from observations or states to actions in otherwise highly non-Markovian domains. Other work can perform poorly in partially observable domains due to a strong Markov assumption and finding an optimal memoryless policy has been proven to be NP-Hard[2].

[1] Stigmergy: "The process of an insect's activity acting as a stimulus to further activity", i.e. indirect coordination through environmental changes.

[2] M. L. Littman, "Markov Games as a Framework for Multi-Agent Reinforcement Learning," in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, 1994.

2 *Approach*

The authors derive a simplification to the Value and Policy Search[3] algorithm (VAPS). The simplification focuses solely on policy search, thus bypassing the complications that arise in the online case when using the error gradient for value search. The main complication being the observation and action quantities needing to be sampled twice to avoid bias in gradient estimation. Yet the only way to get a new observation is to *perform* the action—which is unrealistic online. To avoid this, the authors purpose a simplification called VAPS(1).

[3] L. Baird and A. Moore, "Gradient Descent for General Reinforcement Learning," in *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, 1999.

The instantaneous error measurement associated with policy search is given by $e(z_t) = b - \gamma^t r_t$, where z_t is a state, action, and reward sequence at time t, the discount factor is γ, and r_t is the immediate reward at time t from the sequence. The authors set $b = 0$ for all of their experiments.

```
e(z::Sequence; b=0, γ=0.9) = b - γ^z.t * z.r
```

The policy search error function e is used to update the Q-values during trial runs. Given a learning rate α and an exploration trace $\Phi_{s,a,t}$ associated with the state-action pair (s, a) at time t, the simplified Q-value update is given by

$$Q(s,a) = Q(s,a) - \alpha e(z_t)\Phi_{s,a,t}. \tag{1}$$

Notice that since the authors set $b = 0$ in the error measurement function e, the Q-value update may be further simplified to $Q(s,a) + \alpha\gamma^t r_t \Phi_{s,a,t}$ (however, the Q-value updates were left as expressed in equation 1 for general error functions e). The learning rate α is set by a parameter α_0 and adds a decaying factor based on the trial number N (see figure 1).

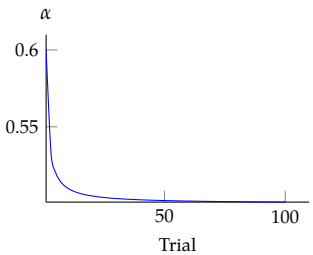

Figure 1. The learning rate decay is defined by $\alpha = \alpha_0 + 1/(10N)$, where N is the trial number and $\alpha_0 = 0.5$ for these experiments.

```
function update_q!(Q::Values, z::Sequence, N::Trial)
    (s::State, a::Action, t::Time) = (z.s, z.a, z.t)
    visit!(s, a, t)
    Q[s,a] = Q[s,a] - α(N)*e(z)*exploration_trace(Q, s, a, t, N)
end
```

Algorithm 2.1. Q-value updates based on the exploration traces $\Phi_{s,a,t}$ given a sequence z and a trial number N.

Similar to the SARSA(λ) algorithm, VAPS uses eligibility traces (or exploration traces as they're called in this paper). The exploration traces are used to keep track of the number of times that the agent explored a state-action pair, and then used to back-propagate the Q-values along the chain of state-action pairs that led to that reward. For some state s and action a, the exploration trace at time t is given by

$$\Phi_{s,a,t} = \frac{1}{c}\left[N^t_{s,a} - N^t_s P(a_t = a \mid s_t = s)\right] \tag{2}$$

$$= \frac{1}{c}\left[N^t_{s,a} - E\left[N^t_{s,a}\right]\right], \tag{3}$$

where c is some temperature parameter (see figure 2), the counter $N^t_{s,a}$ holds the number of times that action a has been executed in state s at time t, and the counter N^t_s holds the number of times that state s has been visited at time t.

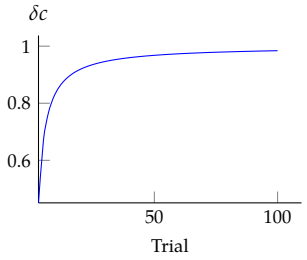

Figure 2. The temperature parameter c was decayed from c_{max} down to c_{min} by an incremental value of $\delta c = (c_{min}/c_{max})^{1/(N-1)}$, where N is the trial number.

```
function exploration_trace(Q::Values, s::State, a::Action, t::Time, N::Trial)
    c::Temp = max(cmax - δc(N), cmin)
    return 1/c * (Nsat[s,a,t] - Nst[s,t]*boltzmann_distribution(Q, s, a, c))
end
```

Algorithm 2.2. The exploration trace $\Phi_{s,a,t}$ assigns credit to state-action pairs proportional to the deviation from expected behavior.

For probabilistic action selection, the Boltzmann distribution is used:

$$P(a_t = a \mid s_t = s) = \frac{e^{Q(s,a)/c}}{\sum_{a'} e^{Q(s,a')/c}}. \tag{4}$$

```
function boltzmann_distribution(Q::Values, s::State, a::Action, c::Temp)
    return exp(Q[s,a]/c) / sum(a'->exp(Q[s,a']/c), actions)
end
```

Algorithm 2.3. The Boltzmann distribution will randomly select actions as a function of their Q-value.

The combination of the state counter N^t_s weighted by the action selection probability from the Boltzmann distribution can be expressed as the expected value of the state-action counter $N^t_{s,a}$ given by $E\left[N^t_{s,a}\right]$ (as seen in equation 3).

3 *Discussion*

The clarity in the mathematics and simplification of the original VAPS algorithm made the implementation relatively easy. Luckily, the authors were open about the hyperparameters they used. Namely, α_0, λ, $c_{\max}$, $c_{\min}$, and b. Although they omitted the value for the discount factor γ, in this test implementation we set it to $\gamma = 0.9$. The current limitations we see are in scaling VAPS(1) to large state-spaces. Due to the counters used in the exploration trace, dimensionality constraints may be reached based on the amount of state and action-space exploration by the agent.

References

1. L. Baird and A. Moore, "Gradient Descent for General Reinforcement Learning," in *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, 1999.
2. M. L. Littman, "Markov Games as a Framework for Multi-Agent Reinforcement Learning," in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, 1994.

A Appendix: Auxiliary Code

This section highlights Julia code otherwise unnecessary for the main body of the text. Namely, helper functions and definitions of custom data types and aliases.

```julia
using DataStructures # for DefaultDict

# Data type aliases
const State = Observation = Action = Trial = Int64
const Reward = Time = Temp = Float64
const Actions = Vector{Action}
const Values = DefaultDict{Tuple{State, Action}, Reward}
const Counter_st = DefaultDict{Tuple{State, Time}, Int64}
const Counter_sat = DefaultDict{Tuple{State, Action, Time}, Int64}

actions = Actions([1,2]) # List of actions
memory = BitVector([0]) # One-bit memory setting
push!(actions, 0) # Append bit-setting action

# Experience sequence
mutable struct Sequence
    s::State
    a::Action
    r::Reward
    t::Time
    Sequence(s,a,t) = new(s, a, R(s,a), t)
end

const cmax = 1.0 # Maximum temperature
const cmin = 0.2 # Minimum temperature

Q = Values(0) # Q-value look-up table
N_sat = Counter_sat(0) # Counter for (s,a,t)
N_st  = Counter_st(0) # Counter for (s,t)

α(N::Trial; α0=0.5) = α0 + 1/(10N) # Learning rate with decay
δc(N::Trial) = (cmin/cmax)^(1/(N-1)) # Temperature decay
R(s::State, a::Action) = s == State(9) ? 1 : 0 # Reward function

# Increment the visit counters
function visit!(s::State, a::Action, t::Time)
    N_sat[s,a,t] += 1
    N_st[s,t] += 1
end
```