Evolutionary Function Approximation for Reinforcement Learning

by Shimon Whiteson and Peter Stone, JMLR 7 (2006) 877-917

Discussion led by Xuejun Liao

18 July 2007
Overview

- Reinforcement learning – what’s the difference from supervised learning
- Temporal difference (TD) learning – **online** learning of optimal control
- Evolutionary approach to optimization
- Learning and evolving
- Algorithmic details
- Experiments
- Summary
Reinforcement learning in a nutshell

- A agent who learns to maximize **long-term** reward
- The agent lives in a space of states
- Taking different actions in a given state leads to different **immediate** rewards
- How can the agent predict **delayed** rewards?
- Markovian state dynamics allow the agent to look into the future
Temporal difference (TD) learning

- Bellman equation

\[ Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|s, a) Q^\pi(s', a'), \quad a' = \pi(s') \]

- Policy improvement

\[ \pi'(s) = \arg \max_a Q^\pi(s, a) \text{ is a better policy than } \pi \]

- Sarsa

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma Q(s', a') - Q(s, a) \right] \]

- Q-learning

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \]
Network representation of $Q(s, a)$

- Each input node corresponds to a state
- Each output node corresponds to an action
- Question: how to determine the network topology?
Evolutionary optimization of network topology

1. Specify a scheme to encode candidate solutions (e.g. binary streams)

2. Generate an initial population, each individual representing a candidate solution

3. Evaluate the fitness of each individual

4. A subset of fit individuals are selected – survival of the fittest

5. A series of operators (crossover, mutation) are applied to the selected individual to produce the next generation

6. Go back to step 3

The offsprings produced by crossover and mutation lie in the neighborhood of the parents
Mutation operators for network topology

Figure 1: Examples of NEAT’s mutation operators for adding structure to networks. In (a), a hidden node is added by splitting a link in two. In (b), a link, shown with a thicker black line, is added to connect two nodes.
Learning and evolving

- An individual is allowed to learn before being evaluated
- Encourage near optimal individuals to survive
- Larmarckian evolution: changes made during learning influence the offsprings
- Darwinian evolution: changes made during learning do not influence the offsprings
Algorithm 1 Q-LEARN($S, A, σ, c, α, γ, λ, ε_{td}, e$)

1: // $S$: set of all states, $A$: set of all actions, $σ$: standard deviation of initial weights
2: // $c$: output scale, $α$: learning rate, $γ$: discount factor, $λ$: eligibility decay rate
3: // $ε_{td}$: exploration rate, $e$: total number of episodes
4: 
5: $N ←$ INIT-NET($S, A, σ$) // make a new network $N$ with random weights
6: for $i ← 1$ to $e$ do
7:   $s, s' ←$ null, INIT-STATE($S$) // environment picks episode's initial state
8:   repeat
9:     $Q[] ← c \times$ EVAL-NET($N, s'$) // compute value estimates for current state
10:    with-prob($ε_{td}$) $a' ←$ RANDOM($A$) // select random exploratory action
11:    else $a' ← \text{argmax}_j Q[j]$ // or select greedy action
12:    if $s \neq$ null then
13:        BACKPROP($N, s, a, (r + γ \max_j Q[j]) / c, α, γ, λ$) // adjust weights toward target
14:    $s, a ← s', a'$
15:    $r, s' ←$ TAKE-ACTION($a'$) // take action and transition to new state
16:   until TERMINAL-STATE?($s$)
Algorithm 2: NeuroEvolution of Augmenting Topologies (NEAT)

Algorithm 2 NEAT(S, A, p, mₙ, mₗ, g, e)

1: // S: set of all states, A: set of all actions, p: population size, mₙ: node mutation rate
2: // mₗ: link mutation rate, g: number of generations, e: episodes per generation

3:  
4: P[] ← INIT-POPULATION(S, A, p)  // create new population P with random networks
5:  
6: for i ← 1 to g do
7:   for j ← 1 to e do
8:     N, s, s′ ← RANDOM(P[]), null, INIT-STATE(S)  // select a network randomly
9:     repeat
10:        Q[] ← EVAL-NET(N, s′)  // evaluate selected network on current state
11:        a′ ← argmaxᵢᵢ Q[i]  // select action with highest activation
12:        s, a ← s′, a′
13:        r, s′ ← TAKE-ACTION(a′)  // take action and transition to new state
14:     until TERMINAL-STATE?(s)
15:     N.episodes ← N.episodes + 1  // update total number of episodes for N
16:     P′[] ← new array of size p  // new array will store next generation
17:  for j ← 1 to p do
18:     P′[j] ← BREED-NET(P[])  // make a new network based on fit parents in P
19:     with-probability mₙ: ADD-NODE-MUTATION(P′[j]) // add a node to new network
20:     with-probability mₗ: ADD-LINK-MUTATION(P′[j]) // add a link to new network
21:     P[] ← P′[]
Algorithm 3 NEAT+Q(S, A, c, p, mn, ml, g, e, α, γ, λ, εtd)

1: // S: set of all states, A: set of all actions, c: output scale, p: population size
2: // mn: node mutation rate, ml: link mutation rate, g: number of generations
3: // e: number of episodes per generation, α: learning rate, γ: discount factor
4: // λ: eligibility decay rate, εtd: exploration rate
5: 
6: P[] ← INIT-POPULATION(S, A, p) // create new population P with random networks
7: for i ← 1 to g do
8:   for j ← 1 to e do
9:     N, s, s' ← RANDOM(P[]), null, INIT-STATE(S) // select a network randomly
10:    repeat
11:      Q[] ← c × EVAL-NET(N, s') // compute value estimates for current state
12:    
13:    with-prob(εtd) a' ← RANDOM(A) // select random exploratory action
14:    else a' ← argmax_k Q[k] // or select greedy action
15:    if s ≠ null then
16:      BACKPROP(N, s, a, (r + γmax_k Q[k])/c, α, γ, λ) // adjust weights toward target
17:  
18:  s, a ← s', a'
19:  r, s' ← TAKE-ACTION(a') // take action and transition to new state
20:  N. fitness ← N. fitness + r // update total reward accrued by N
21: until TERMINAL-STATE?(s)
22: N.episodes ← N.episodes + 1 // update total number of episodes for N
23: P'[] ← new array of size p // new array will store next generation
24: for j ← 1 to p do
25:   P'[j] ← BREED-NET(P[]) // make a new network based on fit parents in P
26:   with-probability mn: ADD-NODE-MUTATION(P'[j]) // add a node to new network
27:   with-probability ml: ADD-LINK-MUTATION(P'[j]) // add a link to new network
28: P[] ← P'[]
Algorithm 4 ε-GREEDY SELECTION($P, \varepsilon_{ec}$)

1: // $P$: population, $\varepsilon_{ec}$: NEAT’s exploration rate
2: 
3: \textbf{with-prob}($\varepsilon_{ec}$) return \textsc{Random}($P$) \hfill // select random network
4: \textbf{else} return $N \in P \mid \forall (N' \in P) N.\text{average} \geq N'.\text{average}$ \hfill // or select champion

Softmax selection: given $s$, select $a$ with probability \[
\frac{e^{Q(s,a)/\tau}}{\sum_{b \in A} e^{Q(s,b)/\tau}}
\]

Algorithm 5 SOFTMAX SELECTION($P, \tau$)

1: // $P$: population, $\tau$: softmax temperature
2: 
3: \textbf{if} $\exists N \in P \mid N.\text{episodes} = 0$ \textbf{then}
4: \quad return $N$ \hfill // give each network one episode before using softmax
5: \textbf{else}
6: \quad \textit{total} $\leftarrow \sum_{N \in P} e^{N.\text{average}/\tau}$ \hfill // compute denominator of Boltzmann function
7: \quad \textbf{for all} $N \in P$ \textbf{do}
8: \quad \quad \textbf{with-prob}($\frac{e^{N.\text{average}/\tau}}{\textit{total}}$) return $N$ \hfill // select $N$ for evaluation
9: \quad \textbf{else} $\textit{total} \leftarrow \textit{total} - e^{N.\text{average}/\tau}$ \hfill // or skip $N$ and reweight probabilities
Problem domain: mountain car
Q vs evolutionary learning

Uniform Moving Average Score Per Episode

NEAT+Q

NEAT

Q-Learning
Online vs offline learning

Uniform Moving Average Score Per Episode

- Epsilon-Greedy NEAT
- Softmax NEAT
- Off-Line NEAT

Score vs Episode (x1000)
Online evolutionary learning

Uniform Moving Average Score Per Episode

- Softmax NEAT
- Softmax NEAT+Q
- Off-Line NEAT
- Off-Line NEAT+Q

Score

Episode (x1000)
Online evolutionary learning on server job scheduling

(b) Server Job Scheduling
Summary

- Introduced an evolutionary approach to automatically discover effective representations for TD function approximators.

- Introduced on-line evolutionary computation, which employs selection mechanisms borrowed from TD methods to improve the on-line performance of evolutionary computation.

- Empirical studies demonstrate that evolutionary optimization of TD function approximators can significantly improve the performance of TD methods and on-line evolutionary computation can significantly improve evolutionary methods.

- A combination of the two offers a promising and general approach to reinforcement learning in large probabilistic domains.