Realizing Undelayed *n*-Step TD Prediction with Neural Networks

Janis Zuters

Faculty of Computing, University of Latvia
Raina bulvaris 19-434, LV-1586 Riga, Latvia
janis.zuters@lu.lv

***Abstract*—There exist various techniques to extend reinforcement learning algorithms, e.g., eligibility traces and planning. In this paper, an approach is proposed, which combines several extension techniques, such as using eligibility-like traces, using approximators as value functions and exploiting the model of the environment. The obtained method, undelayed *n*-step TD prediction (TD-P), has produced competitive results when put in conditions of not fully observable environment.**

I. Extension Techniques of Reinforcement Learning

A. Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning to handle delayed reward. The idea behind all eligibility traces is very simple. Each time a state is visited it initiates a short-term memory process, a trace, which then decays gradually over time. This trace marks the state as eligible for learning.[2]

Algorithm *sarsa_lambda* (n)
Q – state-action values
$\pi(Q)$ – policy derived from Q
γ – discount rate
λ – decay rate
α – learning rate
N – maximum eligibility trace length
Begin
Initialize Q
Do many times
$Trace \leftarrow$ Empty
$s \leftarrow$ random nonterminal state
$a \leftarrow$ choose action from state s according policy $\pi(Q)$
While s is not final state
Append $<s, a>$ to *Trace*
$r, s' \leftarrow$ Take action a, observe reward and next state
$a' \leftarrow$ choose action from state s' according policy $\pi(Q)$
$\delta \leftarrow r + \gamma Q(s',a') - \gamma Q(s,a)$
$\lambda^* \leftarrow 1$
For N steps $<s^*, a^*>$ in *Trace* from the end backwards
$Q(s^*,a^*) \leftarrow Q(s^*,a^*) + \alpha\delta\lambda^*$
$\lambda^* \leftarrow \gamma\lambda\lambda^*$
$s \leftarrow s'$
$a \leftarrow a'$

Fig. 1. Sarsa(λ) (with limited trace length)

There are two ways to view eligibility traces. The more theoretical view of eligibility traces is called the forward view, and the more mechanistic view is called the backward view. The forward view is most useful for understanding what is computed by methods using eligibility traces, whereas the backward view is more appropriate for developing intuition about the algorithms themselves.[3]

Fig. 1 depicts traditional Sarsa algorithm armed with eligibility traces. For simplicity and consistency, it is assumed that the environment is deterministic and the algorithms use state-action values (not just state values).

To view the same process according the forward view, we should look forward from the current state (to the subsequent states) to compute its value; and it can be carried out through making a delay in modification of state-action values.

B. Reinforcement Learning and Neural Networks

Standard reinforcement learning algorithms assume that state-action values are stored in a table – one value per state-action pair.

Algorithm *sarsa_neural*
Ψ – supervised neural network to store state-action values
$\pi(\Psi)$ – policy derived from Ψ
γ – discount rate
«·» – set of features present in (·)
Begin
Initialize Ψ
Do many times
«s» $\leftarrow$ random nonterminal state
$a \leftarrow$ choose action from «s» according policy $\pi(\Psi)$
While «s» does not represent terminal state
r, «s'» $\leftarrow$ Take action a, observe reward and next state
$a' \leftarrow$ choose action from «s'» according policy $\pi(\Psi)$
$v \leftarrow r + \gamma\Psi$(«$s'$,$a'$»)
Train Ψ with «s, a», v
«s» $\leftarrow$ «s'»
$a \leftarrow a'$

Fig. 2. Classic Sarsa algorithm complemented by a neural network to store state-action values

Unfortunately this is not always possible, and big total amount of states and actions is only one of possible reasons. Moreover, there can be too complicated to accurately fill such a table of values, or even to identify all the possible states and actions, because state-action space may include continuous variables or complex sensations. In this case the issue is that of generalization; and a supervised neural network (e.g., multi-layer perceptron) is a typical choice to realize that.

The simplest way of using a neural network Ψ is to apply it instead of value table Q (see Fig. 2 for the classic Sarsa algorithm with a neural network as a value function). To

better describe the algorithm in such conditions, it is worth to add a notion of set of features, e.g., «s» denotes a set of features of some state *s*. In this paper, it is assumed that possible actions are countable, and «*a*» means just the code of action *a* obtained in order to be passed to a neural network as an input.

Eligibility traces are also used combined with function approximators (see [1]). Accordingly also neural networks can be incorporated into a reinforcement learning algorithm with eligibility traces (Fig. 3).

Algorithm ***sarsa_neural_lambda*** (*N*)
Ψ – supervised neural network to store action values
π(Ψ) – policy derived from Ψ
γ – discount rate
λ – decay rate
η – learning rate for Ψ
N – maximum eligibility trace length
«·» – set of features present in (·)
Begin
Initialize Ψ
Do many times
Trace ← Empty
«s» ← random nonterminal state
a ← choose action from «s» according policy π(Ψ)
While «s» does not represent terminal state
Append <«s», *a*> to *Trace*
r, «s'» ← Take action *a* from «s», observe result
a' ← choose action from «s'» according policy π(Ψ)
$v \leftarrow r + \gamma\Psi(\text{«}s'\text{»},a')$
$\eta^* \leftarrow \eta$
For *N* steps $<s^*, a^*>$ in *Trace* from the end backwards
Train Ψ with <«*s*, *a*», *v*> using learning rate η^*
$\eta^* \leftarrow \eta^*\gamma\lambda$
«s» ← «s'»
a ← *a*'

Fig. 3. Sarsa(λ) (with limited trace length) adjusted for using neural network as a value function

C. Planning in Reinforcement Learning

By planning we mean looking ahead in a model of an environment in order to enhance the reinforcement learning process. The model is used to simulate the environment and get some simulated experience.

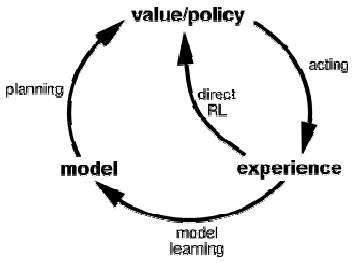

Fig. 4. Relationships among learning, planning, and acting[3][1]

Extension of reinforcement learning with planning is carried out by integrating it with learning and acting processes (See Fig. 4).

There exist various methods to integrate planning into reinforcement learning. A well known one is that of Dyna[4]. In Dyna, to extend learning from real experience, some learning from simulated experience is also added.

The idea of Dyna algorithm is the following: in each step of the episode, additional training is performed for randomly chosen state-action pairs, according the value obtained by the help of the environment model.

Fig. 5 shows the Dyna algorithm [4] adapted for Sarsa learning and neural network as a value function.

Algorithm ***dyna_sarsa_neural***
Ψ – supervised neural network to store state-action values
M – supervised neural network to model environment
π(Ψ) – policy derived from Ψ
γ – discount rate
«·» – set of features present in (·)
Begin
Initialize Ψ, M
Do many times
«s» ← current (nonterminal) state
a ← choose action from «s» according policy π(Ψ)
r, «s'» ← Take action *a*, observe reward and next state
a' ← choose action from «s'» according policy π(Ψ)
$v \leftarrow r + \gamma\Psi(\text{«}s'\text{»},a')$
Train Ψ with «*s*, *a*», *v*
Train M with <«*s*, *a*», <*r*, «*s*'»>>
Repeat *N* times
$\text{«}s^*\text{»} \leftarrow$ random previously observed state
$a^* \leftarrow$ random action previously taken in *s*
$r^*, \text{«}s^{**}\text{»} \leftarrow M(\text{«}s^*\text{»},a^*\text{»})$ *% Take action "in model"*
$a^{**} \leftarrow$ choose action from $\text{«}s^{**}\text{»}$ according policy π(Ψ)
$v \leftarrow r^* + \gamma\Psi(\text{«}s^{**}\text{»},a^{**}\text{»})$
Train Ψ with «s^*, a^*», *v*

Fig. 5. Dyna-Sarsa algorithm

II. UNDELAYED N-STEP TD PREDICTION

This chapter is to describe the "Undelayed *n*-step TD prediction" (TD-P), the author's proposed approach to organize forward-looking mechanism through exploring the environment model, realized by a neural network. In some technical details, this approach links both with eligibility trace mechanism (for using a trace to visit other states) and the Dyna algorithm (for using a model to predict the behaviour of the environment).

A. Inspiration

The main idea of the proposed approach is just 'looking forward' for additional information (instead of looking back, used in most algorithms because of being less complicated in terms of computation) to enhance the algorithm.

Theoretically, looking forward can be realized via several methods, e.g., *n*-step TD prediction, where state-action values are updated in the direction of *n*-step target *R* (See Eq. (1), adapted from [3]):

$$R_t^{(N)}(s,a) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \\ + ... + \gamma^{N-1} r_{t+N} + \gamma^n Q(s_{t+N}, a_{t+N}) \quad (1)$$

Here, looking forward is formal, actually realized through delay mechanism (i.e., it is just a 'forward view', not real looking forward). A "true" looking forward is possible through involving prediction of future states.

B. The Algorithm

N-step prediction can be realized directly through looking forward in the model of environment (instead of the environment itself, like in the Dyna algorithm (Fig. 5)) thus modelling future steps of the episode (So Eq. (1) turns into Eq. (2)):

$$R_t^{(N)}(s,a) = r_{t+1} + \gamma r_{t+2}^{\mathrm{M}} + \gamma^2 r_{t+3}^{\mathrm{M}} + \\ + ... + \gamma^{N-1} r_{t+N}^{\mathrm{M}} + \gamma^N Q(s_{t+N}^{\mathrm{M}}, a_{t+N}) \quad (2)$$

where r_i – actual reward; $r_i^{(M)}$ – modelled future reward; $s_i^{(M)}$ – modelled future state.

Procedure *sarsa_P* (*N*)
  Ψ – supervised neural network to store action values
  M – supervised neural network to model environment
  π(Ψ) – policy derived from Ψ
  γ – discount rate
  N – maximum forward trace length
  n(·,·) – forward trace length function
  «·» – set of features present in (·)
Begin
  Initialize Ψ, M
  $T \leftarrow 0$
  Do many times
    $T \leftarrow T + 1$
    «s» ← random nonterminal state
    a ← choose action from «s» according policy π(Ψ)
    While «s» does not represent terminal state
      r, «s'» ← Take action a from «s», observe result
      $v \leftarrow r$
      $\gamma^* \leftarrow \gamma$
      «s^*» ← «s'»
      $n^* \leftarrow \boldsymbol{n}(T)$
      While «s^*» does not represent terminal state and $n^* \geq 0$
        a^* ← choose action from «s^*» according policy π(Ψ)
        r^*, «s^{**}» ← M(«s^*,a^*») *% Take action "in model"*
        $n^* \leftarrow n^* - 1$
        If «s^{**}» represents terminal state or $\tau^* \leq 0$
          $v \leftarrow v + \gamma^* r^*$
        Else
          $v \leftarrow v + \Psi$(«s^*,a^*»)
        $\gamma^* \leftarrow \gamma\gamma^*$
        «s^*» ← «s^{**}»
      Train Ψ with <«s, a», v>
      Train M with <«s, a», <r, «s'»>>
      a' ← choose action from «s'» according policy π(Ψ)
      «s» ← «s'»
      $a \leftarrow a'$

Fig. 6. Sarsa-P: Sarsa with TD-P

Unlike the Dyna algorithm (Fig. 5) and also eligibility-trace-based algorithms (Fig. 3), in this approach, additional activities within a step of an episode concern predicted future states, instead of previously observed ones.

There's one more complication within the context of this approach: in the beginning, while the available information on the environment is insufficient, it isn't reasonable to look far forward. Therefore, to avoid "unacceptable looking forward" in the beginning of the trial, the maximum count of steps (to look forward) *N* should be replaced by some function $\mathbf{n}(\cdot,\cdot)$ (See Eq. (3)), designed in a way that for the first episodes no looking forward is performed.

$$R_{T,t}^{(\mathrm{n}(N,T))}(s,a) = r_{T,t+1} + \gamma r_{T,t+2}^{\mathrm{M}} + ... + \\ \gamma^{\mathrm{n}(N,T)-1} r_{T,t+\mathrm{n}(N,T)}^{\mathrm{M}} + \gamma^{\mathrm{n}(N,T)} Q(s_{T,t+\mathrm{n}(N,T)}^{\mathrm{M}}, a_{T,t+\mathrm{n}(N,T)}) \quad (3)$$

where T – number of the episode, N – maximum forward trace length.

The pseudocode of Fig. 6 depicts Sarsa-P algorithm: the approach of TD-P applied to the Sarsa algorithm.

C. Benefits of the Approach

Of course, there's no idea to apply neural-networks-based algorithms for "simple" problems, e.g., for those with fully observable environment and easy identifiable states. Any practical benefits for such algorithms can be expected only in "hard conditions", so here we assume that the problems are hard enough to be worth using neural networks to model the value function.

Advantages of approaches with eligibility traces over traditional TD algorithms are well known – such algorithms are faster.

In terms of ideology, the main advantage of the proposed approach of TD-P over eligibility traces is diminished amount of training operations needed for the neural network, realizing the state-action value function:

- In algorithms with eligibility traces, the value is computed for the current state, and then propagated back over the trace by performing training many times – for each state-action in the trace;
- In this TD-P approach, travelling over the trace is done just to compute the value, and then training is performed only once – for the current state.

Although this advantage tends to be a bit heuristic (because it can be sensed only in special conditions); it can be considerable, if the training process is heavy to perform.

III. Experimental Work

A. The Problems to Solve

The experimentation was done on relative simple problems just handled as if the environment was not fully observable. The goal of the experimentation was to show such an approach working, as well as be able to outperform other methods in certain "hard" conditions.

The idea of the experiments was very simple: the algorithms were tested on two deterministic grid-world problems: 5×6 and 7×8 (See Figs 7 and 8) and the outputs were compared against the "suboptimal policies" (The set of suboptimal policies was obtained as statistics through traditional TD algorithms as a set of possible actions for each state encountered at least in 5% of cases). Each trial was run until a suboptimal policy was obtained (good result) or 100,000 episodes were reached (bad result). The discount rate γ=0.9 was used. The environments of the problems were set to behave as if they were not fully observable.

White cells denote normal (nonterminal) states; black cells stand for the wall, while shaded cells represent terminal states. A number in a cell denotes the reward of a move to this cell.

There are 4 actions: *up*, *down*, *left*, *right*, which cause the agent to move one cell in the corresponding direction. The state remains unchanged if the agent tries to go off the grid or against the wall.

-1	-1	-1		10	-1
-1	-1	-1	-1		-1
-1	-10	-10		-10	-1
-1	-1	-1	-1		-1
-1	-1	-10	-1	-1	-1

↓	←↓	←↓			←
↓	←	←	←		↑
↓					↑
→	→	→	↓		↑
↑→	↑		→	→	↑

Fig. 7. Problem 5×6 and its suboptimal policies

-1	-1	-1	-1	-20	-1	-1	100
-1					-1		
-1	-1	-1	-1		-1	-1	-1
-1	-1		-1		-1	-1	-1
			-1		-5	-5	-1
-1	-1	-1	-1	-5	-8	-1	-1
-1	-1	-1	-1	-10	-1	-1	-1

↓→	→	→	→	→	→	→	
↕					↑		
↑	↓↔	↔	←↓		↑	←	←
↑→	←↑		↕		↑	←↑	←↑
			↕		↑	↑	↑
↓→	↓→	↓→	↕→	↓↔	↑→	↑→	↑
↑→	↑→	↑→	↑→	↑→	↑	↑→	↑

Fig. 8. Problem 7×8 and its suboptimal policies

B. Configuration of the Testing Environment

The problems were tried to solve via three following algorithms, all of which used a neural network as a value function:

- Sarsa (Fig. 2);
- Sarsa(λ) (Fig. 3) with two different maximum trace lengths – 3 and 5;
- Sarsa-P (Fig. 6) with two different maximum trace lengths – 3 and 5.

For neural networks, the following coding was used:

- State – two numbers from the interval [0, 1] (corresponding to the horizontal and vertical position of the state in the grid);
- Action – one number from the interval [0, 1].

Neural network (multi-layer perceptron) architectures used (one hidden layer):

- Value function: 3-40-1
- Transition rewards (Sarsa-P only): 3-40-1
- Transition states (Sarsa-P only): 3-40-2

Other parameters of the experiments:

- Maximum count of episodes in a trial – 100,000.
- Learning rates for neural networks – 0.1.
- Decay rate λ=0.9 (Sarsa(λ) only)
- Policies were computed from the corresponding value functions in ε-greedy manner with ε=30%.

The forward trace length function $\mathbf{n}(\cdot,\cdot)$ was realised by the following function (Eq. (4)):

$$\mathrm{n}(N,T) = \begin{cases} 0; & T = 0 \\ \mathrm{n}(N,T-1) + g(N - \mathrm{n}(N,T-1)); & T > 0 \end{cases} \quad (4)$$

where T – number of the episode, N – maximum forward trace length, g – change rate (for the current experimentation g was set to 0.0001).

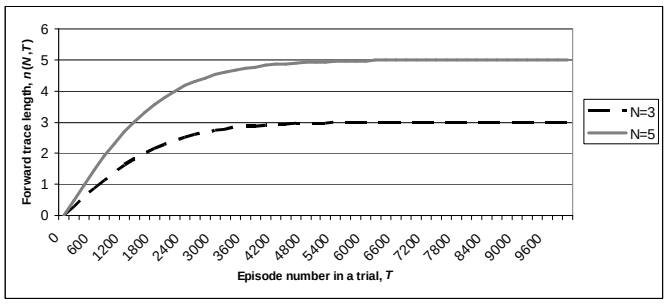

Fig. 7. Illustration of the forward trace length function $\mathbf{n}(\cdot,\cdot)$ (Eq. (4)) used in Sarsa-P algorithm (Fig. 6) to avoid looking forward too far in the first episodes

C. Experimental Results

The total of 164 experimental trials was performed to form the final results (Fig. 8). Two of there problems were solved via five different variants of the algorithms (2×5=10 cases). Amount of trials which converged to a suboptimal policy (i.e., a satisfactory result) was obtained.

Ca. 58% of trials of the "easiest" problem 5×6 converged successfully. Best results were obtained by Sarsa(λ).

Only about 18% of trials converged in the case of the "harder" problem 7×8, and here the best result was reached by the Sarsa-P algorithm.

Summary of the results for different kinds of the Sarsa algorithm featured by neural networks as value function:

- Best results for the problem 5×6 were reached by Sarsa(λ);
- Sarsa(λ) was generally better than traditional Sarsa. This would be just a little complement to credibility of experimentation with such configuration;
- Best results for the problem 7×8 were reached by Sarsa-P;
- Traditional Sarsa was absolute unable to solve the problem 7×8;
- Sarsa-P was surprisingly weak on solving the "easy" problem 5×6.

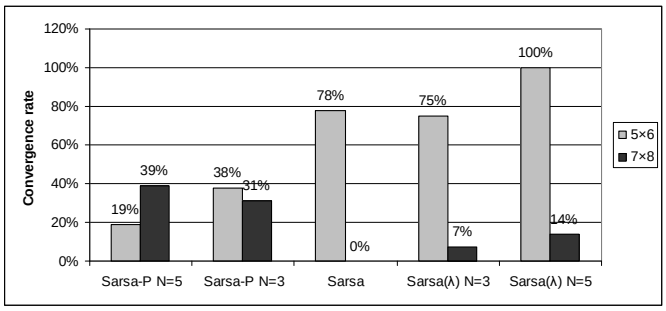

Fig. 8. Experimental results – percentage of trials converged, according the problems solved and algorithms applied

The most "disappointing" ones were the results of Sarsa-P on solving the "easiest" 5×6 problem (19% and 38% depending on the maximum length of the trace). It would probably signal for instability of the algorithm or its current realization.

IV. Conclusion

The proposed approach combines several methods or techniques to extend reinforcement learning:

- Using traces (like eligibility traces) to visit other states;
- Using approximators for value functions;
- Exploiting the model of the environment to obtain additional information.

The approach has been designed for use in specific "hard" conditions of problem solving – not fully observable environment with potentially unidentifiable states.

The main goal of the experimentation was to show such an approach working, and this has been reached – the corresponding algorithm has shown competitive results against other methods.

Although Sarsa-P was weak in solving one of the problems: 5×6; it clearly outperformed other algorithms on solving the "more complicated" problem 7×8, and this keeps the proposed approach of TD-P promising for further development.

References

[1] K. Framling. "Replacing eligibility trace for action-value learning with function approximation," *European Symposium on Artificial Neural Networks*, Bruges, Belgium: Apr. 2007

[2] S. P. Singh and R. S. Sutton, "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning*, 22:123-158, 1996.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning. An introduction.* Cambridge,MA: MIT Press/A Bradford Book, 1998.

[4] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *SIGART Bulletin*, 2:160–163, 1991.