The Coupled-Weight Neural Reinforcement Algorithm

JANIS ZUTERS
Artificial Intelligence Foundation Latvia;
Faculty of Computing, University of Latvia
Raina blvd. 19, Riga, LV-1586
LATVIA
janis.zuters@lu.lv

Abstract: - Reinforcement learning refers to a set of machine learning problems for optimal control, and there are a lot classical algorithms and methodologies to solve them. Typically such algorithms are based on the value-function approach, where values of the states (or actions) are iteratively computed, and this set of values infers the policy. Although neural networks are known to be used within such systems, if so, a neural module is typically delegated only with a subfunction, e.g., to model the value table. This work introduces a brand new neural-network-based reinforcement learning algorithm CNR (Coupled-Weight Neural Reinforcement Algorithm), designed using ideas of reinforcement comparison, temporal difference learning, and Hebbian learning.

Key-Words: - Neural networks, Neural reinforcement algorithm, Reinforcement learning

1 Introduction and Motivation

Reinforcement learning (RL) is one of main paradigms of machine learning (ML), the objective of which is to compute – how to map situations to actions. This is a natural way to solve control problems. As for terminology, in addition to RL as a ML paradigm, it also refers to the respective set of problems. It is believed by many authors that reinforcement is more general to other types of machine learning.

In case of RL, the learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the two most important distinguishing features of reinforcement learning. [1]

Classical RL algorithms are mainly based on the value-function approach, where values of the states (or actions) are iteratively computed, and the set of values infers the policy. While we have neural network (NN) alternatives of supervised learning algorithms (e.g. back-propagation, radial basis function networks [2]), it is hard to find NN alternatives for RL problems. In RL algorithms, neural networks are typically used as approximators of internal data structures like value functions [2],[3],[4],[5].

The aim of the research is to obtain pure NN approaches to solve RL problems. In solving RL problems, NNs are attractive to the author in several aspects:

- NNs are more fault-tolerant and biologically plausible to compare to traditional algorithms,
- NNs typically exploit more general algorithmic approaches and data structures,
- Building NNs to solve RL problems is still a challenging issue.

This has lead to development of the new **Coupled-weight Neural Reinforcement algorithm** (**CNR**), which emerged from such known ideas as reinforcement comparison, temporal difference (TD) learning, Hebbian learning, as well as the actor-critic approach.

2 Related Work

3.1 Actor-Critic Methods

Classical RL algorithms (like dynamic programming, SARSA (*State-Action-Reward-State-Action*)) are explicitly based on the **value-function** approach, where values of the states (or actions) are iteratively computed, and this set of values infers the policy.

As an important step forward from these algorithms one should name **actor-critic methods** (Fig. 1), which are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. Here the value function is still present, but its impact to the policy is not so direct. Actor-critic methods are the natural extension of the idea of so called **reinforcement comparison** methods to TD learning and to the full RL problem. [1]

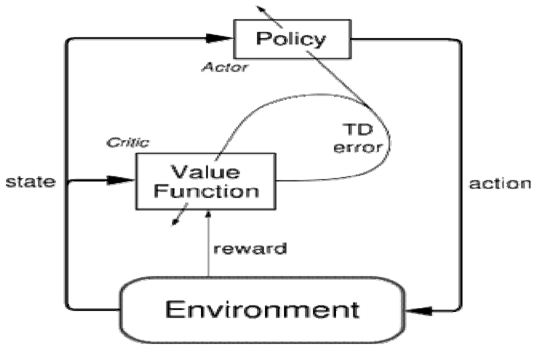

Fig. 1. The actor-critic architecture ([1])

Fig. 2 shows an actor-critic algorithm which lights up the two main principles of the approach (lines 007-008): (a) the value function *V* is present, (b) different data structures for the policy (*P*) and the value function.

Technically speaking, both structures, the policy and the value table, are improved in parallel, and they cooperate in a way.

```
Module actor_critic()
    m – amount of states in the environment
    n – amount of available actions
    P(1..m,1..n) – policy description (actor)
    V(1..m) – state value function (critic)
    η – learning rate
    γ – discount rate
Begin
001     P := initialize for all states and actions
002     V := initialize for all states
003     s := select an initial state of the environment
004     Do many times
005         a := select action according to policy description P
                (eventually, with exploration)
006         Apply action a to the environment, obtain reward r
                and next state s2
007         $P(s,a) := P(s,a) + \eta \cdot [r + \gamma \cdot V(s2) - V(s)]$
008         $V(s) := r + \gamma \cdot V(s2)$
009         s := s2
```

Fig. 2. Simplified actor-critic algorithm for continuing tasks (with no terminal states) (adapted from [1])

3.2 Neural Networks and Reinforcement Learning

If a neural network or a similar mechanism is used within a RL system, it typically is delegated only with a subfunction, e.g., to model the value function. Nonetheless there have been successful attempts to build a self-contained neural network to be able to solve RL problems. [6],[7]

Schafer [7] proposes recurrent neural networks (RNNs) to solve RL problems. In this approach, the recurrent control neural network combines system identification and determination of an optimal policy in one network, and in contrast to most RL methods, the optimal policy is computed directly without making use of a value function. To train the network, the shared weight extended back-propagation algorithm is used.

Although back-propagation based learning techniques are very popular also with neural reinforcement algorithms, they don't exactly by nature fit RL. Coulom [8] reviews the main difficulties in this context, and one is the risk of being struck by ill-conditioning. So, using back-propagation could potentially require additional technical efforts to overcome the difficulties.

Back-propagation is not the only option to train neural reinforcement systems. In [9], Hebbian learning is used to enhance the classical RL mechanism.

3 The CNR Algorithm

In this Section, the new Coupled-weight Neural Reinforcement algorithm is proposed. On one hand, the CNR algorithm can be viewed as putting together such techniques as reinforcement comparison, temporal difference (TD) learning, Hebbian learning, and placing them into a single-layer neural network frame. On the other hand, the new algorithm represents a self-contained neural network able to solve RL problems and is built without use of supervised learning methods. By far, the CNR algorithm still requires some justification and additional testing to make it complete.

3.1 Main Principles and Architecture of CNR

CNR is implemented through a one-layered feed-forward neural network with the following characteristics (Figs. 3, 4):

- State (*s*), presented through the input is a sequence of 0 and 1. In its simplest form, the size of the input pattern corresponds to the amount of states, and a single 1 represents the current state while the rest of the input pattern is filled with zeros.
- Reward (*r*) is a real number normalized into interval [0, 1].
- The network has one layer of neurons, and each neuron is preset to correspond to one RL action,

so the amount of neurons is the same as the amount of actions.

- After any iteration according to the activity earned, exactly one of the neurons becomes the winner, and so the action (*a*) is determined. Information about the winner neuron is stored as feedback *F*; learning in the winner neuron is performed differently.
- There are several structures for cumulating input and reward information (*S*, *S*\`, *R*).
- Neurons have **two types of weights**: *W*, *E*. In its simplest form, both sets resemble action-value-tables of traditional RL algorithms in a way. Both sets of weights are trained differently to be then coupled together in determining the action.
- Unlike action-values which are cumulated from rewards and can reach values that significantly exceed typical absolute values of rewards, weights *W* and *E* are computed through some kind of adaptation, so they are restricted in intervals [0, 1].

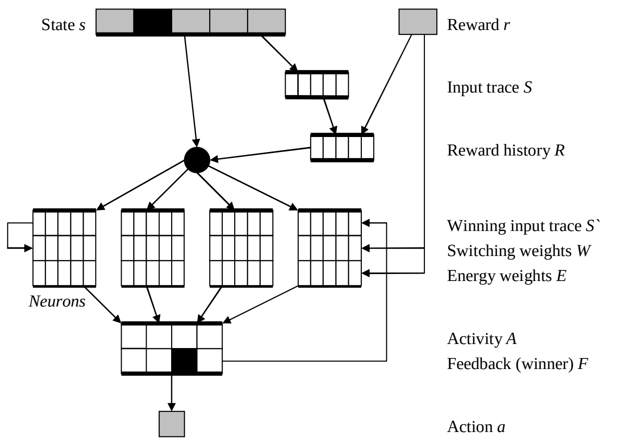

Fig. 3. Overview of the CNR architecture.

The effect of using both weight sets *W* and *E* was discovered accidentally. Initially two different algorithms were being developed trying to mimic various aspects of known RL techniques by a NN.

3.1 Detailed Description of CNR

The complete description of the algorithm is given in pseudocode (Figs. 4-6), but the most important constructs are reduplicated in form of equations.

Main loop of the CNR algorithm is the same as for traditional RL algorithms (Fig. 4, lines 105-110).

Input pattern *s* at time step *t* is represented by a sequence of 0s and 1s (in the simplest case, one ‘1’ and the rest ‘0’):

$$s(t) = \langle x \mid x \in \{0,1\} \rangle . \tag{1}$$

During each run (module ***run_network***, Fig. 5), one winner neuron is determined (lines 208-211) according to obtained activities but with randomness added.

Generally speaking, neuron activities are computed in a traditional way – as sums of products (2):

$$A_j(t) = \sum_i \begin{cases} s_i(t)E_{ji}(t); & \text{if } W_{ji}(t) > \omega \\ 0; & \text{otherwise} \end{cases}, \tag{2}$$

where *t* – time step, implemented in pseudocode through *T* and *T*\`, while the rest of notions here and on match exactly and are completely described there.

However the peculiarity with activity is that:

- The switching weights W are used to determine, whether not to ignore this synapse (line 206), they are typically close to marginal values of 0 and 1 by nature;
- The energy weights E are used to constitute the product (line 207), they are ideologically close to the notion of action-values.

```
Module CNR()
    n – neuron count in the network, as well as count of
            available actions in the environment
    m – input size, as well as weight count of a neuron
    s(1..m) – input pattern, representing current state, a vector
            of binary <0,1> values
    W(1..n,1..m) – switching weights
    E(1..n,1..m) – energy weights
    e – actual energy
    r – actual reward
    T(1..n,1..m) – time since last spike
    T`(1..n,1..m) – time since last spike delayed for one step
    R(1..m) – reward history for each input, used to compute W
    S(1..m) – input trace (for each input), used to compute R
    S`(1..n,1..m) – winning input trace (for each synapse), used
            to compute W
    F(1..n) – feedback; the only winner neuron is represented
            by true
Begin
101     W := initialize all with small values, bigger than ω
            (e.g., 0.1)
102     E := initialize all with 0.5
103     T := initialize all with 100 (any big positive integer)
104     Initialize R, S, S` (all values with 0)
105     s := select an initial state of the environment
106     Do many times
107         a := run_network_CNR(s)  # Compute action to be
            performed
108         Apply action a to the environment, obtain reward r
            and next state s2
109         train_network_CNR(s)
110         s := s2
```

Fig. 4. CNR algorithm.

The two types of weights are coupled together to achieve the goal.

During the training phase (module ***train_network***, Fig. 6), weights *W* and *E* are learned.

For the switching weights *W*, two different input traces (similar to eligibility traces) are computed:

- Traces *S* (lines 302-303) are maintained in order to compute reward history *R* (line 304). *R* are then used as reference rewards in the Hebbian-like update mechanism (line 314)
- Traces *S*` (lines 309, 312) are directly used in the update formula (line 314)

```
Module run_network_CNR(s) Returns a
    A(1..n) – activity, the resulting values of running the
              neurons
    a – (number of the) action computed by the system <1..n>
    ρ – random activity rate [0..1] (e.g., 0.1)
    ω – minimum productive weight [0..1] (e.g., 0.05)
Begin
201   F := initialize all with false
202   A := initialize all with 0
203   Forall neurons j Do
204       A(j) := 0
205       Forall synapses i in j Do
206           If W(j,i) > ω Then
207               A(j) := A(j) + s(i) · E(j,i)
208   If (get random value from interval [0..ρ]) < ρ Then
209       a := choose action <1..n> in random
210   Else
211       a := index_max(A)
212   F(a) := true
```

Fig. 5. Running the network.

Weights *W* are adjusted with respect to input traces *S*` and the difference between actual reward r and the appropriate reference reward R_i, and restricted in interval [0, 1] (lines 314-316, (3) and (4)):

$$\Delta W_{ji}(t) = \eta(r(t) - R_i(t))S`_{ji}(t), \tag{3}$$

$$W_{ji}(t) = \begin{cases} 1; & \text{if } W_{ji}(t-1) + \Delta W_{ji}(t) > 1 \\ 0; & \text{if } W_{ji}(t-1) + \Delta W_{ji}(t) < 0 \\ W_{ji}(t-1) + \Delta W_{ji}(t); & \text{otherwise} \end{cases}. \tag{4}$$

Energy weights *E* are computed in a two-piece way, conceptually similar to that of TD learning ((5) and (6)):

- Impact of actual reward to *E* (line 321),
- Obtaining information from other energy weights with one-step delay (lines 326-327).

$$\Delta E^{(1)}_{ji}(t) = \sum_{j\in\{k|F_k(t)=true\}} \eta`\mu[r(t) - E_{ji}(t-1)]s_i(t)\cdot \tag{5}$$

$$\Delta E^{(2)}_{ji}(t) = \sum_{j\in\{k|F_k(t-1)=true\}} \eta`[e(t) - E_{ji}(t-1)], \tag{6}$$

where *e(t)* is actual energy at time *t* (lines 318, 320, 322):

$$e(t) = \sum_{j\in\{k|F_k(t)=true\}} \frac{\sum_i s_i(t)E_{ji}(t)}{\sum_i s_i(t)}. \tag{7}$$

In pseudocode, the mechanism *T*` is used to implement temporal delay (lines 307, 310, 313).

```
Module train_network_CNR(s)
    λ – decay rate for input traces [0..1] (e.g., 0.9)
    β – reward cumulation rate [0..1] (e.g., 0.05)
    η – learning rate for switching weights [0..1] (e.g., 0.1)
    η` – learning rate for energy weights [0..1] (e.g., 0.05)
    μ – instant energy adaptation rate [0..1] (e.g., 0.1)
Begin
        # Learning phase 1/3. Update input traces and
              reward history
301   Forall input values s(i) in s Do
302       If s(i) = 1 Then S(i) := s(i)
303       Else S(i) := S(i) · λ
304       R(i) := R(i) + β · S(i) · [r – R(i)]
        # Learning phase 2/3. Main update
305   Forall neurons j Do
306       Forall synapses i in j Do
              # update winning input traces and switching
              weights
307           T`(j,i) := T(j,i)
308           If F(j) And s(i) = 1 Then
309               S`(j,i) := s(i)
310               T(j,i) := 0
311           Else
312               S`(j,i) := S`(j,i) · λ
313               T(j,i) := T(j,i) + 1
314           W(j,i) := W(j,i) + η · [r – R(i)] · S`(j,i)
315           If W(j,i) > 1 Then W(j,i) := 1
316           Elseif W(j,i) < 0 Then W(j,i) := 0
          # update energy weights (1/2) and collect actual
              energy
317       If F(j) Then
318           e := 0
319           Forall synapses i in j Do
320               e := e + s(i) · E(j,i)
321               E(j,i) := E(j,i) + μ · η` · s(i) · [r – E(j,i)]
322           e := e / sum(s)
        # Learning phase 3/3. update of energy weights with
              "future" actual energy
323   Forall neurons j Do
          # update energy weights (2/2)
325       Forall synapses i in j Do
326           If T`(j,i) = 0 Then  # if fired in the previous
              step
327               E(j,i) := E(j,i) + η` · [e – E(j,i)]
```

Fig. 6. Training the network.

3 Experimental Results

The main goal of the experimentation was to validate the proposed concept of CNR, i.e, to show that a neural reinforcement algorithm is able work comparably to classical algorithms. For this purpose, an original experimentation framework in C++ was developed.

In the beginning, a set of benchmarks was prepared, based on four different deterministic grid-world problems with up to several tens of states and four possible actions (one of the problems is given in Fig. 7a):

- SARSA [1] was used as the benchmark algorithm to obtain a set of policies (Fig. 7b) by running it 100 times on each problem with discount rate γ=0.9.
- For a fixed state, a certain action was chosen as an alternative policy, if it was computed as the policy at least once from those 100 running times. The complete set of these alternative policies is then referred to as the suboptimal policies.

The new algorithm CNR was tested on the prepared benchmarks the following way:

- CNR was configured according to Table 1 and run 100 times for each out of four problems in the training mode – 100,000 moves (or steps) in each trial.
- One state of a problem was encoded as a sequence of 0s with exactly one 1 to identify the state.
- The trained system then was run to the problem 1,000 steps in the validation mode (comparing the decision of the system to the benchmark).

To validate the algorithm, a special ‘validity rate’ was computed as a percentage of steps in which the decision of the CNR system matches any of the alternatives from the benchmark.

-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

(a)

↓→	→	→	→	→	→	→	▨
↕	■	■	■	■	↑	■	■
↑	↓↔	↔	←↓	■	↑	←	←
↑→	←↑	■	↕	■	↑	←↑	←↑
■	■	■	↕	■	↑	↑	↑
↓→	↓→	↓→	↕→	↓↔	↑→	↑→	↑
↑→	↑→	↑→	↑→	↑→	↑	↑→	↑

(b)

Fig. 7. Pproblem example, a 7×8 gridworld. Terminal state is depicted shaded: (a) problem definition – reward obtained, when moving to the appropriate state; (b) set of suboptimal policies, obtained by SARSA algorithm.

As CNR by construction is not directly suited for episodic problems (ones with terminal states), the problems were adjusted that when reaching a terminal state, the system automatically jumps to some non-terminal state in random (instead of stopping the episode).

We got ~95% validity rate for CNR algorithm, and this result conceptually shows the algorithm to solve RL problems.

CNR exploits the principles of several known RL mechanisms:

- Decay rate λ (Fig. 6), used in training switching weights, conceptually corresponds to trace-decay parameter (of the same name) λ of the ‘eligibility-traces’ mechanism.
- To mimic the effect of γ in training energy weights, a special adaptation rate μ is used, as well as different learning rates η and η\`.

Table 1. CNR-specific parameters and their values used in the experiments.

Param-eter	Description	Used value
λ	decay rate for input traces	0.8..0.95
β	reward cumulation rate	0.05
η	learning rate for switching weights	0.1
η\`	learning rate for energy weight	0.02
μ	instant energy adaptation rate	0.1
ρ	random activity rate	0.1
ω	minimum productive weight	0.03

4 Conclusion and Future Work

A new neural reinforcement algorithm CNR is built and shown to conceptually work. The computed validity rate is 95% shows that it doesn’t still work exactly as classical ones, so the model of CNR still requires further investigations in order to better understand its connection with classical RL notions, as well as ability to solve real-world control problems. The main goal of the research was to put a reinforcement learning mechanism into a neural architecture, and this is achieved.

The model should be extended to more complicated inputs (now a state is encoded as a sequence of 0s with exactly one 1, so the weights and reference rewards strongly resemble value tables), and this would probably require a multi-layered neural architecture. The second direction of potential research to consider is that of continuous action spaces.

As the joint effect of using both types of weights was recognized by part accidentally, we acknowledge the model underlying CNR to be further explored.

References:
[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning. An introduction*, Cambridge, MA: MIT Press/A Bradford Book, 1998.
[2] S. Haykin, *Neural networks: a comprehensive foundation*, 2nd ed. Prentice-Hall, Inc, 1999.
[3] J. Zuters, Realizing Undelayed N-Step TD Prediction with Neural Networks, *Proceedings of the 15th IEEE Mediterranean Electrotechnical Conference* (MELECON 2010), Valletta, Malta, April 26-28, 2010, pp. 102-106.
[4] B.-Q. Huang, G.-I. Cao, M. Guo, Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance, *Proceedings of the Fourth International Conference on Machine Learning and Cybernetics*, Guangzhou, China, August 18-21, 2005.
[5] C.J. Gatti, M.J. Embrechts, Reinforcement Learning with Neural Networks: Tricks of the Trade, *Advances in Intelligent Signal Processing and Data Mining*, Vol.410, 2013, pp. 275-310.
[6] A.M. Schaefer, D. Schneegass, V. Sterzing, and S. Udluft, A neural reinforcement learning approach to gas turbine control, *Proc. of the 20th Int. Joint Conf. on Neural Networks*, Orlando, 2007. MIT Press.
[7] A.M. Schafer. *Reinforcement Learning with Recurrent Neural Networks,* PhD thesis, University of Osnabruck, 2008
[8] M. R. Coulom, Feedforward Neural Networks in Reinforcement Learning Applied to High-dimensional Motor Control, *13th International Conference on Algorithmic Learning Theory*, 2002.
[9] R.J. Bosman, W.A. van Leeuwen, B. Wemmenhove, Combining Hebbian and reinforcement learning in a minibrain model. *Neural Netw.* 2004 Jan;17(1): 29-36.