A Scalable Recurrent Neural Network Framework for Model-free POMDPs

April 3, 2007

Zhenzhen Liu, Itamar Elhanany

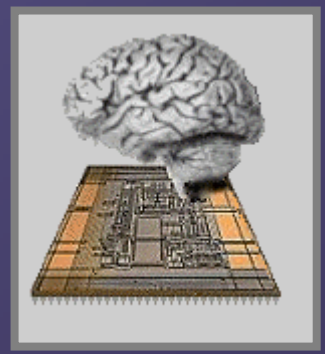
Machine Intelligence Lab
Department of Electrical and Computer Engineering
The University of Tennessee
http://mil.engr.utk.edu

Outline

- Introduction
- Background and motivation
- TRTRL/SMD
- Simulation results
- Summary and future work

Ingredients for building "Intelligent Machines"

- Implementation platform?
  - Must scale
  - Mammal brain as a reference model?
    - Massively parallel architecture
    - Operates at (relatively) low speeds
    - Fault-tolerant
  - Software vs. Hardware
    - If hardware, what technology? (FPGA, VC VLSI, Analog VLSI)
- (Nonlinear) Function approximation
  - Dealing with high-dimensional problems
    - Optimal policy is unattainable
  - Capturing spatiotemporal dependencies
  - RNNs, Bayesian Networks, Fuzzy?
  - Biologically-inspired schemes



Scaling ADP

- Goals …
  - To address high-dimensional state and/or action spaces
  - Support online learning
  - Deal with partially observable scenarios (e.g. POMDPs)
  - Hardware realizable
- Approach taken …
  - Employ recurrent neural networks (RNNs)
    - Improved learning algorithm that scales
    - Devised hardware-efficient architecture
  - Embed within approximate Q-Learning framework

The Real-Time Recurrent Learning (RTRL) Algorithm

- Originally proposed in 1989 for arbitrary RNN topology
- Stochastic gradient-based online algorithm
- Activation function of neuron k is defined by:

$$y_k(t+1) = f_k(s_k(t)),$$

$$z_k(t) = \begin{cases} x_k(t) & if\ k \in input \\ y_k(t) & if\ k \in output \end{cases}$$

where s_k is the weighted sum of all activations leading to neuron k.

- The network error at time t is defined by:

$$J(t) = \frac{1}{2} \sum_{m \in \text{outputs}} [d_m(t) - y_m(t)]^2 = \frac{1}{2} \sum_{m \in \text{outputs}} [e_m(t)]^2$$

where $d_m(t)$ denotes the desired target value for output neuron m

Updating the weights

- The error is minimized along a positive multiple of the performance measure gradient such that

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} = \alpha \sum_{k \in outputs} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}$$

- The partial derivatives of the activation function with respect to the weights are identified as sensitivity elements and denoted by

$$p_{ij}^{k}(t) = \frac{\partial y_k(t)}{\partial w_{ij}}$$

Updating the Sensitivities in RTRL

- The sensitivities of node k with respect to a change in weight w_{ij} are updated using the recursive expression

$$p_{ij}^{k}(t+1) = f_k'(s_k(t))\left[\sum_{l \in N} w_{kl}\, p_{ij}^{l}(t) + \delta_{ik} z_j(t)\right]$$

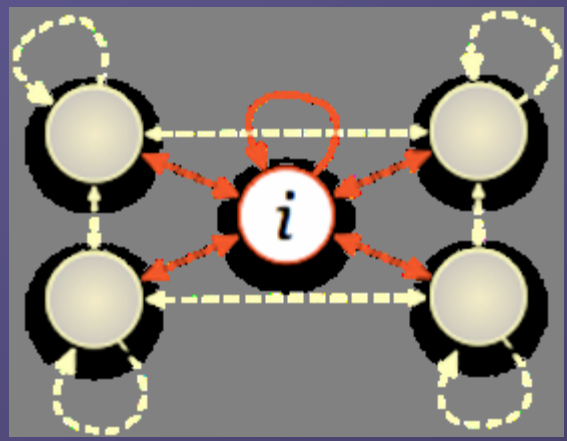
Each neuron performs $O(N^3)$ multiplications, yielding a total computational complexity of $O(N^4)$

The storage requirements are dominated by the weights and the sensitivities resulting in $O(N^3)$ storage requirements

Truncated RTRL (TRTRL)

Motivation:
To obtain a scalable version of the RTRL algorithm while minimizing performance degradation.

How?
Biologically-inspired approach: limit the sensitivities of each neuron to its ingress (incoming) and egress (outgoing) links.



Revising Sensitivity Updates for TRTRL

- For all nodes not in the output set, the ingress sensitivity function for node i is given by

$$p_{ij}^{i}(t+1) = f_i'(s_i(t))\left[w_{ij}\, p_{ij}^{j}(t) + z_j(t)\right]$$

- The egress sensitivities for node i are updated by

$$p_{ij}^{j}(t+1) = f_j'(s_j(t))\left[w_{ji}\, p_{ij}^{i}(t) + \delta_{ij}\, y_j(t)\right]$$

- For the output neurons, a nonzero sensitivity element must exist in order to update the weights, yielding

$$p_{ij}^{o}(t+1) = f_o'(s_o(t))\left[w_{oi}\, p_{ij}^{i}(t) + w_{oj}\, p_{ij}^{j}(t) + \delta_{io} z_j(t)\right]$$

Storage and Computational Complexity of TRTRL

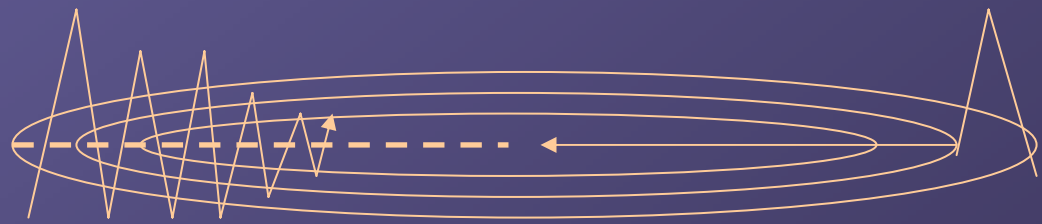
- The network architecture remains the same with TRTRL (there's a weight between each two neurons)
- Only the calculation of sensitivities is reduced
- The computational load for each neuron becomes $O(KN)$ where K denotes the number of output neurons

The computation complexity was reduced from $O(N^4)$ to $O(KN^2)$

The storage requirement was reduced from $O(N^3)$ to $O(N^2)$

Stochastic-Meta Descent (SMD – N. Schraudolph et al.)

- Gradient descent techniques often suffer from slow converges, particularly for ill-conditioned problems
- Mainstream approach: utilize second-order information, e.g. LM, Newton methods (all utilize Hessian matrix)
  - However, these are computationally heavy
- Stochastic meta-descent (SMD) has recently been proposed as a "cheap second-order gradient technique"
  - Employs an independent learning rate for each weight
  - Utilizes Hessian information in local step size



$$w_{ij}(t+1) = w_{ij}(t) + \lambda_{ij}(t)\delta_{ij}(t)$$

SMD adopted for TRTRL

- We adopted SMD for TRTRL (first work in applying SMD to RNNs)
- Approach - adapt learning rate along exponentiated gradient descent direction

$$\ln \lambda_{ij}(t) = \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial \ln \lambda_{ij}},$$

$$\ln \lambda_{ij}(t) = \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial w_{ij}(t)} \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}}$$

$$= \ln \lambda_{ij}(t-1) + \mu \delta_{ij}(t) v_{ij}(t)$$

safeguard factor against unreasonably small, or negative, values

$$\lambda_{ij}(t) = \lambda_{ij}(t-1) \max\left(\rho, 1 + \mu \delta_{ij}(t) v_{ij}(t)\right)$$

using relationship $e^x \approx 1 + x$

SMD adopted for TRTRL (cont.)

- Adapt gradient trace

$$v_{ij}(t+1) = \beta v_{ij}(t) + \lambda_{ij}(t)\left(\delta_{ij}(t) - \beta\left(H_t v(t)\right)_{ij}\right)$$

- H_t is the instantaneous Hessian (the matrix of second derivatives $\partial^2 J/\partial w_{ij} w_{kl}$ of the error J with respect to each pair of weights) at time t
- The product of the Hessian and an arbitrary vector

$$Hv = R_v\{\nabla_w\} = \frac{\partial}{\partial r}\nabla_{(w+rv)}\big|_{r=0}$$

which for TRTRL yields

$$-\left(H_t v(t)\right)_{ij} = R_v\left\{\sum_{o \in output} e_o(t)\, p_{ij}^o(t)\right\} = \sum_{o \in output}\left[e_o(t) R_v\left\{p_{ij}^o(t)\right\} - R_v\left\{y_o(t)\right\} p_{ij}^o(t)\right]$$

SMD in TRTRL

- To complete the analysis, the R-operator on S, Y, P is

$$R_v\{y_o(t)\} = f'(s_o(t))R_v\{s_o(t)\} \qquad R_v\{s_o(t)\} = \sum_{l \varepsilon U \cup I} v_{ol}(t) z_l(t),$$

$$R_v\{p_{ij}^o(t)\} = f''(s_o(t))R_v\{s_o(t)\} \cdot \left[w_{oi} p_{ij}^i(t) + w_{oj} p_{ij}^j(t) + \delta_{io} z_j(t)\right] + f'(s_o(t))\left[v_{oi} p_{ij}^i(t) + v_{oj} p_{ij}^j(t)\right]$$

- We also added adaptive global meta-learning rate by defining

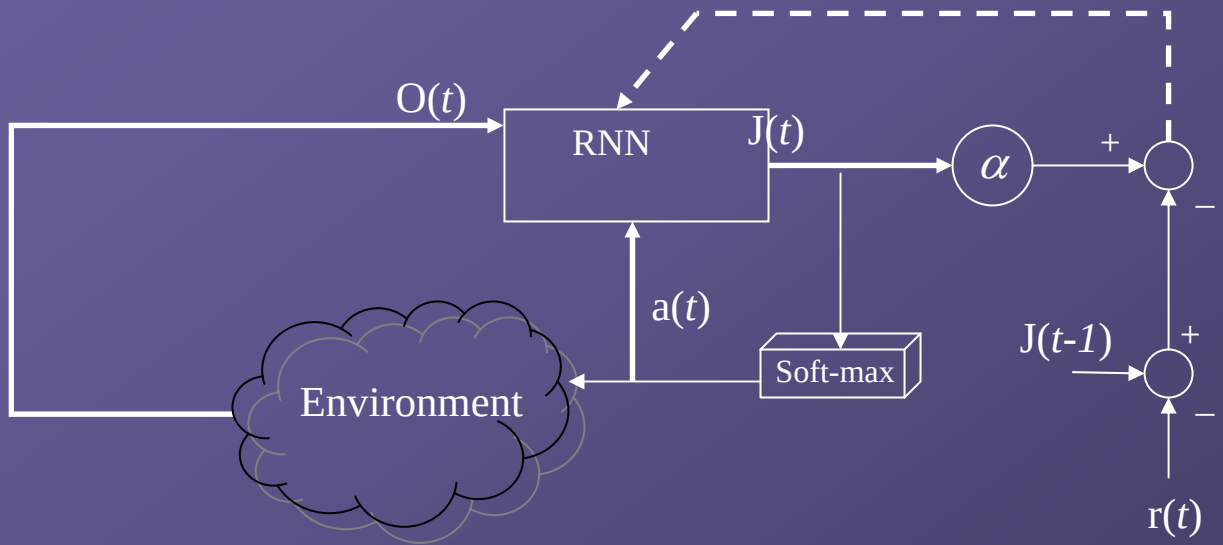
$$\varphi_{ij}(t) = -\frac{\partial J(t)}{\partial \ln \lambda_{ij}} = \delta_{ij}(t) v_{ij}(t)$$

to yield

$$\mu_{ij}(t) = \mu_{ij}(t-1)\left(1 + \eta \varphi_{ij}(t) \varphi_{ij}(t-1)\right)$$

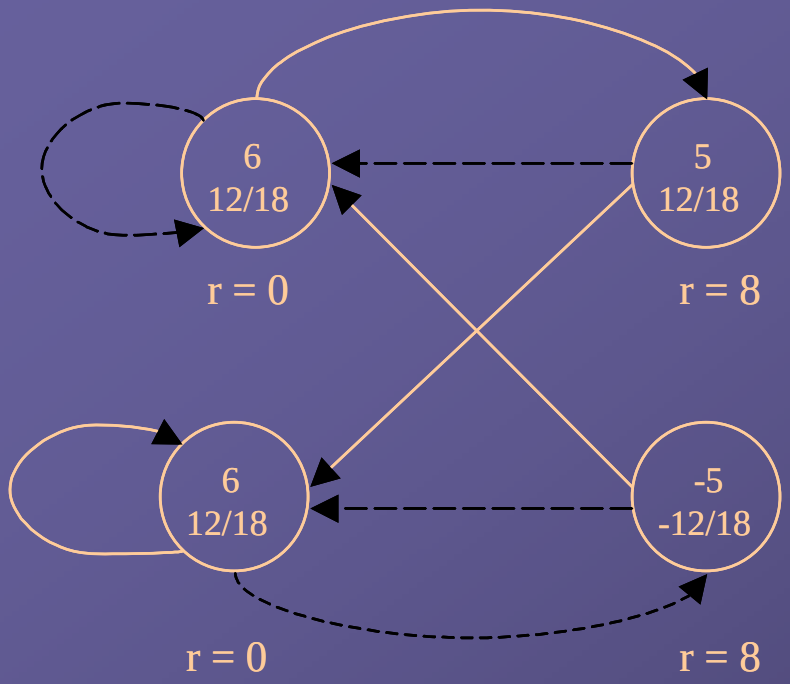
Using TRTRL/RNN for Solving POMDP

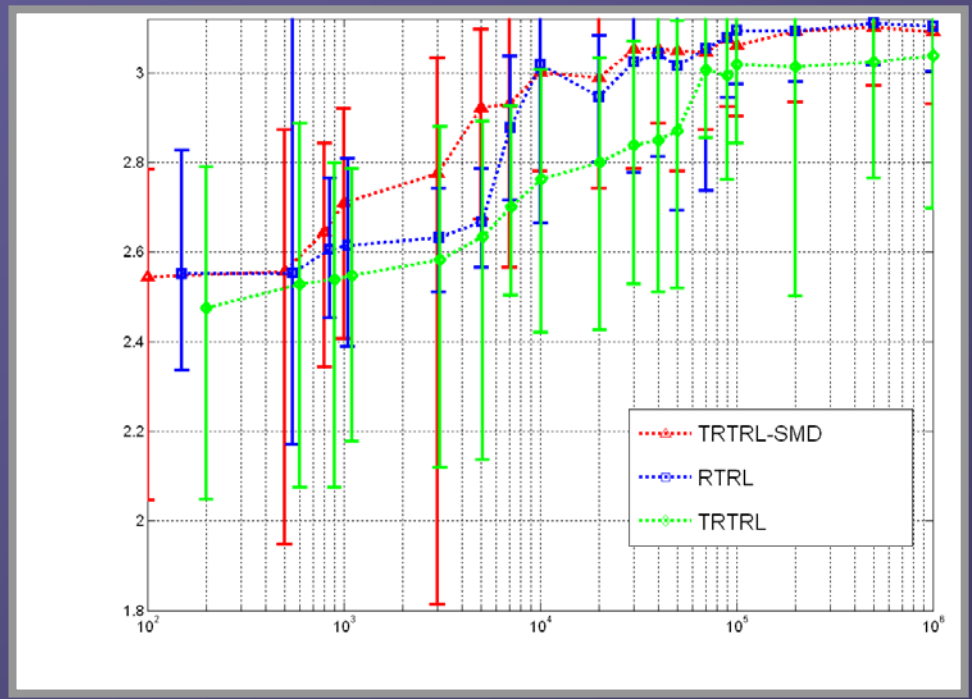
- Recall that the motivation for using RNNs was to solve POMDPs



- In each step: (1) feedforward all actions, (2) find the one with maximal (soft-max) value (J), (3) apply corresponding action to the environment, (4) get next reward and update weights

Example 2 - Four State POMDP





- 4-state POMDP with identical (confusing) observations
  - Agent needs to 'remember' prior observation to infer state
  - 15 internal neurons and 1 output neuron

Summary

- Scalable, efficient RNNs
  - Vital tool for addressing high-dimensional POMDPs
  - Introduced a fast, hardware-efficient learning algorithm and architecture
  - Slightly improved SMD technique (adaptive global learning rate)
- Successfully applied TRTRL-SMD in solving POMDP
  - Pathway for addressing practical problems
  - Scalable framework for ADP with RNNs