Sequential Constant Size Compressors for Reinforcement Learning

Linus Gisslén, Matt Luciw, Vincent Graziano, and
Jürgen Schmidhuber

IDSIA, University of Lugano
6928, Manno-Lugano, Switzerland
{linus,matthew,vincent,juergen}@idsia.ch

Abstract. Traditional Reinforcement Learning methods are insufficient for AGIs who must be able to learn to deal with Partially Observable Markov Decision Processes. We investigate a novel method for dealing with this problem: standard RL techniques using as input the hidden layer output of a Sequential Constant-Size Compressor (SCSC). The SCSC takes the form of a sequential Recurrent Auto-Associative Memory, trained through standard back-propagation. Results illustrate the feasibility of this approach — this system learns to deal with high-dimensional visual observations (up to 640 pixels) in partially observable environments where there are long time lags (up to 12 steps) between relevant sensory information and necessary action.

Keywords: recurrent auto-associative memory, reinforcement-learning

1 Introduction

The classical approach to RL [22] makes strong assumptions such as: the current input of the agent tells it all it needs to know about the environment. However, real-world problems typically do not fit this simple Markov Decision Process (MDP) model, as they are of the partially observable POMDP type, where the value function at a given time depends on the history of previous observations and actions. It remains an open problem as how some developmental and general agent may learn to handle Partially Observable Markov Decision Problems (POMDPs) in real-world environments. Recent extremely general RL machines for POMDPs [11] are *theoretically optimal*. However, these are not (yet) nearly as practical as simpler, yet general (though non-optimal and non-universal), solvers based on RL with Recurrent Neural Networks (RNNs).

In this paper we introduce a novel RNN approach for solving POMPDs with a RL machine, potentially useful for scaling up AGIs. Let us quickly review previous work in this vein. The neural bucket brigade algorithm [18] is a biologically plausible, local RNN RL method. Adaptive RNN critics [19, 3] extend the adaptive critic algorithm [4] to RNN with time-varying inputs. Gradient-based RL based on interacting RNNs [20] extend Werbos' work based on feed-forward nets [24]: One RNN (the model net) serves to model the environment,

the other one uses the model net to compute gradients maximizing reinforcement predicted by the model. Recurrent Policy Gradients and Policy Gradient Critics [25] can be used to train RNN such as LSTM [10] — these significantly outperformed other single-agent methods on several difficult deep memory RL benchmark tasks. Many approaches to *evolving* RNNs (Neuroevolution) have been proposed [27]. One particularly effective family of methods uses cooperative coevolution to search the space of network components (neurons) instead of complete networks [14, 6]. CoSyNE was shown [7] to be highly efficient, besting many other methods including both single-agent methods such as Adaptive Heuristic Critic [2], Policy Gradient RL [23], and evolutionary methods like SANE, ESP, NEAT [21], Evolutionary Programming [16], CMA-ES [9], and Cellular Encoding [8]. Finally, Natural Evolution Strategies [26] for RNNs use *natural* gradients [1] to update both objective parameters and strategy parameters of an Evolution Strategy with a Policy Gradient-inspired derivation from first principles; results are competitive with the best methods in the field.

Here, instead of using a RNN controller, we develop an unsupervised learning (UL) layer that presents a representation of the spatiotemporal *history* to a non-recurrent controller developed through standard RL. The UL takes the form of a Sequential Constant-Size Compressor (SCSC), which can be trained in an unsupervised fashion to sequentially compress the history into a constant size code. Providing that the essential aspects of the history are captured unambiguously by the SCSC, the code that emerges is suitable for classical RL. If successful, a SCSC obviates the need for an RNN controller on the RL layer and makes the partially-observable problem tractable for MDP methods.

Our choice of SCSC is the Recurrent Auto-Associative Memory (RAAM) which has been well-studied in the area of natural language processing by Pollack et al. [15, 12] for two decades. The RAAM can be used as a sequential compressor (sRAAM): given a current data point and a representation of the current history it produces a representation of the new history. Conversely, given a history the sRAAM can reconstruct the previous data point and a representation of the previous history, so, theoretically it may be able to reproduce the entire history. Practically, it can be realized as an autoencoder neural network, and it is amenable to unsupervised training by standard back-propogation.

Our choice of a RAAM-based UL layer to overcome non-Markovian environments is partially motivated by the recent success seen by using less general *feedforward* auto-encoders to pre-train in unsupervised fashion a deep feedforward (non-recurrent) neural net [5]. Such stacks of auto-encoders have already been used as preprocessing for RL [13]. Here, sRAAM can be viewed as a significant generalization thereof: not only can spatial patterns be compactly encoded, but so can spatial-temporal patterns. The spatial-temporal compression achieved by the sRAAM potentially yields a Markovian code that significantly simplifies the RL problem.

In what follows, we describe the first systems, *SERVOs*, which combine a sequential constant-size compressor with reinforcement-learning. We examine the interplay between SCSC and RL under resource constraints for both. Ex-

periments show the strength of the approach for high-dimensional observation sequences and long time lags between relevant events.

2 Sequential Recurrent Auto-Associative Memory

Assume we have a temporal sequence of data points $\mathcal{H}_n = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)$ where $\boldsymbol{p} \in \mathbb{R}^N$, which we shall refer to as the *history* at time $t = n$.

sRAAM is an example of a compressor that sequentially stores sequences into block of fixed size. An (N, M)–sRAAM is given by a pair of mappings (E, F),

$$E\colon \mathbb{R}^{N+M} \to \mathbb{R}^M$$
$$F\colon \mathbb{R}^M \to \mathbb{R}^{N+M},$$

where N is the dimension of the data points and M is the size of the memory block. The mappings E and F are often determined by parameters, and in such cases we make no distinction between the parameters and the mappings determined by them. Typically, sRAAM is implemented using a multi-layer perceptron with at least one hidden layer, which we shall refer to as the *code layer*. The code layer is the domain of F and the codomain of E. The input and output layers have size $N + M$, making it an autoencoder, and the code layer has size M.

Formally, the sRAAM compresses and decompresses a history as follows: Given data point $\boldsymbol{p}_i \in \mathbb{R}^N$ and a history, represented by $\boldsymbol{h}_{i-1} \in \mathbb{R}^M$ we can represent the new history, at time i with the map E,

$$E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}) = \boldsymbol{h}_i.$$

Likewise, given a representation of a history $\boldsymbol{h}_i$ the mapping F is used to recover the data point $\boldsymbol{p}_i$ and the previous representation of the history $\boldsymbol{h}_{i-1}$,

$$F(\boldsymbol{h}_i) = \boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}.$$

The representation $\boldsymbol{h}_0$ of empty history $\mathcal{H}_0 = \emptyset$ needs to be decided upon to encode any history. See Figure 1.

In practice $F(E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1})) \neq \boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}$, so we write the result of the compress—decompress step as

$$F(E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1} i)) = \widehat{\boldsymbol{p}}_i \oplus \widehat{\boldsymbol{h}}_{i-1}.$$

Given a representation $\boldsymbol{h}_n$ of the history $(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)$, found by iterating over E, we can decode the entire history $(\widehat{\boldsymbol{p}}_1, \ldots, \widehat{\boldsymbol{p}}_n)$ by iterating over F. We say that an sRAAM is *trained* when

$$\|\widehat{\boldsymbol{p}}_i - \boldsymbol{p}_i\| \leq \gamma^{k-i}\epsilon$$

for $1 \leq i \leq k$. The parameter $\gamma \geq 1$ is used to relax the importance of recovering the data points $\boldsymbol{p}$ as the sRAAM decodes further back in time. Since we intend

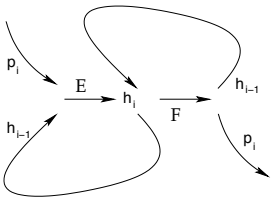


Fig. 1: sRAAM architecture. E, the *compressor*, takes data point $\boldsymbol{p}_i$ and a representation $\boldsymbol{h}_{i-1}$ of the history $\mathcal{H}_{i-1} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{i-1})$ and maps it to a representation $\boldsymbol{h}_i$ of the new history $\mathcal{H}_i$. F, the *decompressor*, takes a representaion $\boldsymbol{h}_i$ of the history and maps it to the previous data point $\boldsymbol{p}_i$ and representation $\boldsymbol{h}_{i-1}$.

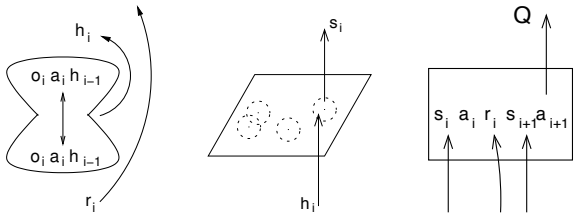
to use an SCSC to generate a representation $\boldsymbol{h}$ of a history $\mathcal{H}$ for the purpose of supplying a state to an RL module it is usually not necessary to put $\gamma = 1$.

Since we have a target $\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}$ to train $\widehat{\boldsymbol{p}}_i \oplus \widehat{\boldsymbol{h}}_{i-1}$ towards for each point of the history, gradient based methods for the RAAM are attractive. Classically, the sRAAM has always been realized as an autoencoder and trained using back-propogation. That is, the weights (E, F) of the network are updated so that the output of the autoencoder $F(E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1})) = \widehat{\boldsymbol{p}}_i \oplus \widehat{\boldsymbol{h}}_{i-1}$ is more like the input $\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}$. This is the only realization that we consider in this paper; we are using an out-of-the-box sRAAM in a new way.

One major concern with this training method is that the network is being trained on moving targets. After performing a step of back-propogation the mapping E changes, which in turn changes the input $\boldsymbol{p} \oplus \boldsymbol{h}$ at the next time step since $\boldsymbol{p}$ is changed. This is an issue that does not arise for an autoencoder which does not use a virtual recurrent connection. That said, the method of training is *often* successful, and it avoids the computational costs associated with methods such as back-propogation through time.

3 SERVO: SCSC assisted RL

Here we introduce a proto-type version of an architecture which combines SCSC and RL, and refer to such systems as *SERVOs*. The UL layer which uses an sRAAM, which assumes the form of an autoencoder neural network, and the RL layer uses SARSA(λ) [22]. The training of the two layers takes place independently in a back-forth manner. After a number of episodes the sequential compressor is trained. After training the compressor, the code is passed through an intermediary layer which is used to establish internal states and is determined by straight-forward clustering, or Vector Quantization (VQ). Using experience replay, the value-fuction is then learned using SARSA(λ) on the states provided by the intermediary layer. The use of an internal layer allows the system to do tabular RL. After the reinforcement-learning, the agent interacts with the environment using the updated policy to collect more samples, these samples are used to repeat the process: train the compressor (E, F), update the intermediary layer V, use experience replay to generate a new policy Q. See Figure 2.



(a) sRAAM (b) VQ (c) SARSA(λ)

Fig. 2: SERVO. The sRAAM autoencoder is trained using back-propogation on $\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}$. After training $\boldsymbol{h}_i$ is generated using E and is passed the internal state layer. VQ assigns an internal state $\boldsymbol{s}_i$ to the represntation $\boldsymbol{h}_i$ of the history. The RL layer receives Markovian data and learns an action-value function using SARSA(λ).

3.1 UL Layer: sRAAM

We first describe how to train on a single episode $\mathcal{H} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)$, where $\boldsymbol{p}_i$ is a vector representation of both the observation and action at step i. Start with an arbitrary choice for the representation $\boldsymbol{h}_0$ of the empty history $\mathcal{H}_0 = \emptyset$. The training process walks over the history: Given $\boldsymbol{p}_i$ and $\boldsymbol{h}_{i-1}$ use the autoencoder (E, F) to generate an error δ,

$$\delta = \boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1} - F(E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}))$$

then perform a step of back-propogation on the autoencoder. After updating the weights E and F, find a represention $\boldsymbol{h}_i$ of the history up to step i,

$$\boldsymbol{h}_i = E(\boldsymbol{p}_i \oplus \boldsymbol{h}_{i-1}),$$

then iterate, using $\boldsymbol{p}_{i+1}$ and $\boldsymbol{h}_i$.

Given the current policy the agent interacts with its environment for I episodes to generate a collection of episodes $\Xi = \{\mathcal{H}\}_I$. The sRAAM is trained by repeatedly sampling from Ξ and then performing an epoch of back-propogation by walking over the observation-action pairs in the episode. See Figure 2a.

3.2 Internal State Layer: VQ

The building of the internal state layer takes place simultaneously with the reinforcement learning. To maintain a clear exposition we present the two processes separately. After training the sRAAM we walk through each episode $(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)$ to generate representations of the history at each step: $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n)$. For an (N, M)-sRAAM the code lives in M-dimensional space. We generate a set of internal states by a (cheap) clustering all the representations $\{\boldsymbol{h}\}$ produced by each of the episodes. Let $\mathcal{S}$ be a collection of points $s \in \mathbb{R}^M$ representing the internal states of the SERVO. The internal state layer is initialized to the empty set after training the UL layer. Fix a value for the parameter κ. Given a point $\boldsymbol{h}$ the point

$\boldsymbol{s}^*$ in $\mathcal{S}$ closest to it is found. The point $\boldsymbol{h}$ is added to $\mathcal{S}$ if the squared Euclidean distance between $\boldsymbol{h}$ and $\boldsymbol{s}^*$ is greater than κ. Otherwise, for the purpose of the reinforcement-learning, $\boldsymbol{h}$ is identified with $\boldsymbol{s}^*$. This layer is built by randomly choosing an episode and then considering, in order, all the $\boldsymbol{h}$ associated with the episode. The layer is finished being built, as is the reinforcement-learning after each episode has been walked through exactly once. See Figure 2b.

3.3 RL Layer: SARSA(λ)

A review of SARSA(λ) can be found in [22]. We learn on each of the episodes as the internal state layer is built, as follows. For a given episode the data coming into the system can be parsed into 5-tuples, $(\boldsymbol{o}_i, \boldsymbol{a}_i, r_i, \boldsymbol{o}_{i+1}, \boldsymbol{a}_{i+1})$. The SCSC maps $\boldsymbol{o}_i \oplus \boldsymbol{a}_i$ to $\boldsymbol{h}_i$, as explained above. The internal layer maps $\boldsymbol{h}_i$ to an internal state $\boldsymbol{s}_i$. This mapping is determined while the learning proceeds; if $\boldsymbol{h}_i$ is not within $\sqrt{\kappa}$ of the point $\boldsymbol{s}^*$ to which it is closest, then $\boldsymbol{h}_i$ is added to the internal layer and it is mapped to itself. Otherwise $\boldsymbol{h}$ is mapped to $\boldsymbol{s}^*$. Finally, the 5-tuple that is passed to the RL layer has the form, $(s_i, a_i, r_i, s_{i+1}, a_{i+1})$. The function Q is trained on each of the episodes in an arbitrary order using *off-line* SARSA(λ). See Figure 2c.

4 Experiments and Results

4.1 Partially Observable Vision Maze

As a proof of concept to show that this system handles high-dimensional observations with some memory requirement, we performed a visual navigation experiment (see Figure 3). The agent's observations are given by its internal camera, always aimed forward, and its actions are to go forward, rotate left or right, or turn around. Each observation (16×10 pixels) also contains Gaussian noise to avoid possible trivial solutions where the agent memorizes all the views. Each episode begins by placing the agent at a random position and orientation in the maze.

We used a large three layer sRAAM with a code layer of size 100. Observations are 160-dimensional and there are 4 actions, therefore the shape of our network is $264 - 100 - 264$. Each episode lasts until either the goal is reached or the agent has taken 250 actions. **NB:** Using a random walk requires an average of 220 actions to reach the goal, and on average only one-quarter of the walks reach the goal within the 250 allotted actions. A training iteration consisted of data gathering: 2000 walks/rollouts generated using the current policy, followed by training the sRAAM (learning rate 0.01) for 300 epochs, followed by experience replay to develop the value function (discount factor 0.8, $\kappa = 0.9$, and SARSA learning rate 0.1). The actions shifted from 50% exploration to pure exploitation linearly through the training iterations.

We compared SERVO to the state-of-the-art SNES [17] algorithm, which directly searches the weight space of RNNs to find better controllers. SNES generates a population of controllers from a gaussian distribution, and based on

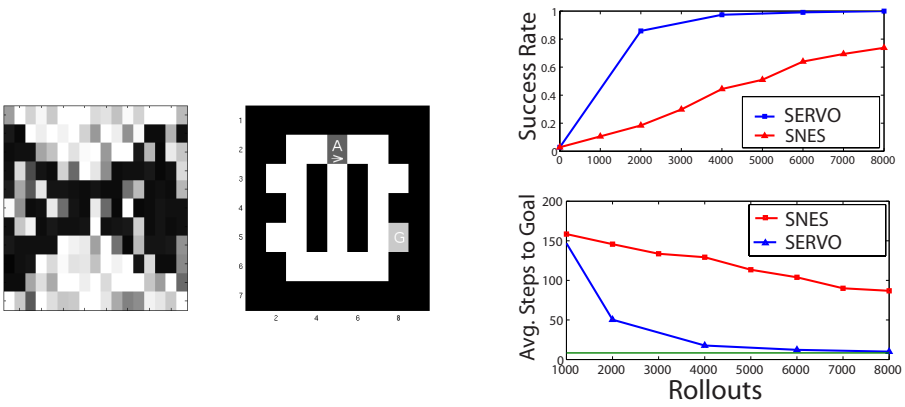


Fig. 3: Left: Example 16x10 dimensional observation. Middle: Top-down maze view (never observed by the agent). Right: Experimental comparison between SERVO and SNES. A "roll-out" is a single controllers interaction with the environment: from a start position, actions are taken until the goal is reached or time runs out. Compared are the average number of steps and success rate between the best SNES controller and SERVO, averaged over 10 experiments. The green line (constant value) represents optimal performance.

the fitness evaluation (each individual started in 50 random start positions), computes the natural gradient to move the distribution to a presumably better location. Figure 4 for a comparison of the two methods. Due to gradient information provided by the fitness function, SNES is also able to deal with this task.

4.2 Learning to Wait

We try a task with higher-dimensional inputs and explicitly require longer memory (up to 12 steps). The agent is placed at one end of a corridor, and can either move forward or wait. The goal is to move through the door at the far end of the corridor, where it is given a visual signal that vanishes in the next frame. One of the signals, *A, B, C, D*, is shown for a single frame when the agent reaches the door, corresponding to a waiting time of *6, 8, 10,* and *12* frames respectively. The agent receives a (positive) reward when it waits the exact number of frames indicated before exiting, otherwise the agent receives no reward and goes back to the start. The episode ends either when the agent walks through the door or 20 frames have passed (to avoid extremely long episodes). This is a difficult task: in the case of letter "D" a random policy will on average require 2^{12} trials to make one successful walk.

The agent receives noisy visual input in the form of 32×20 pixel image. We had trouble getting SERVO to work robustly (with noise) in this task, so we first had to train and use an autoencoder for de-noising the observations before they are passed to the sRAAM. The autoencoder was trained on-line over 5k random

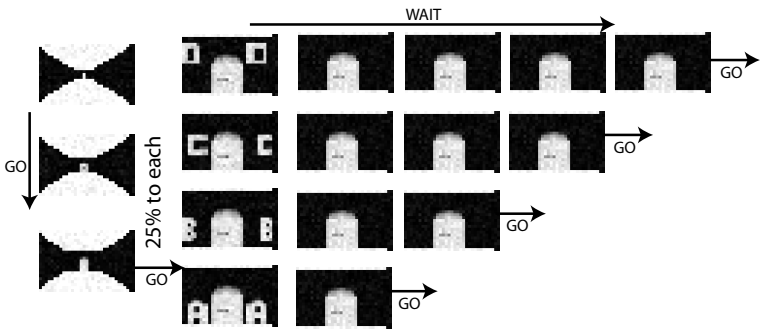


Fig. 4: Shown are the various views of the corridor from the agent's vantage point. The leftmost images are observed as the agent approaches the door. The second column shows the various wait signals, indicating the number of frames to wait before exiting.

walks. After training the autoencoder, the agent performs a series of random walks (approx. 100k) to collect encoded training samples for the SERVO. The SERVO is then trained batchwise: (1) 200 epochs of training to compress the *successful* episodes, first training the UL layer and then training the RL layer, as described in Section 3. (2) the agent again interacts with the environment for 100 episodes to evaluate its policy. Training continues until the agent has achieved better than 90% success rate[1].

In this task, there are only a few general sequences worth encoding. They are difficult to find and locating one does not help to find the others. A "fitness landscape" for RNN controllers in this task would be made up of sharp ridges and vast plateaus. It may seem that all that can be done here is to find and store the best sequences. Yet, the SERVO technique goes further and compresses these sequences. A representation of a current sequence can then be located in the space of compressed previously seen valuable seqeunces (the VQ layer). The current sequence can then be identified with the closest prototype.

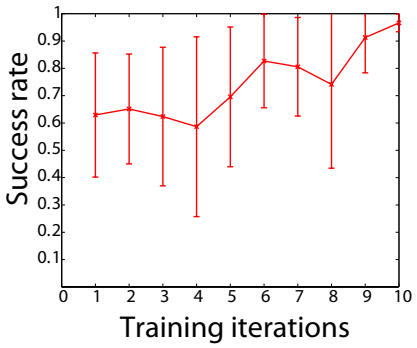


Fig. 5: The average success rate (n=10), with standard deviation for the corridor experiment. After each training episode (200 epochs) the SERVO is tested for 100 episodes.

[1] There is a video of SERVO operation at `www.idsia.ch/~gisslen/SERVOagent.html`

5 Discussion

In problems with high-dimensional observations and deep memory requirements, direct search of weight-space for an RNN controller is quite a difficult task (even with a state-of-the-art method such as SNES). It must find relevant regularities to build on in parameter space using only the fitness measures of the individuals. The high-dimensionality and generality of some problems may be too difficult for direct evolutionary search. In contrast, the SERVO architecture decouples the problem of encoding the relevant spatiotemporal regularities from learning how to act on them. The SERVO separates the learning problem into two components: (1) unsupervised learning of an autoencoder to provide a (quasi-)Markovian code, and (2) classical reinforcement-learning.

The compression capacity of the sRAAM is limited, and cannot be expected to recall all histories of a given length. However, since the sequences are generated by way of reinforcement learning the compressor can in principle learn to represent the important histories unambiguously. This biased training of the unsupervised layer allows the agent to improve its policy, steering it towards increasingly valuable sequences, thereby further refining the UL layer.

We have demonstrated that the use of an SCSC is a competitive method for solving high-dimensional POMDPs with long time lags. Yet, the current system is not sufficiently stable for real-world AGIs. Future work will refine this first-generation SERVO architecture.

Acknowledgments We would like to thank Faustino Gomez for his insights and thoughtful feedback. We would also like to thank Bas Steunebrink and Jan Koutník for helping with coding. This research was funded in part through the following grants: Sinergia (CRSIKO 122697), NeuralDynamics (grant 270247), NanoBioTouch (36RANANO), IM-Clever (231722), and SNF (200020-122124).

References

1. S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.
2. Charles W. Anderson. Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Labs, Waltham, MA, 1987.
3. B. Bakker. Reinforcement learning with Long Short-Term Memory. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*. MIT Press, Cambridge, MA, 2002.
4. A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.
5. Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Neural Information Processing Systems (NIPS)*, 2007.
6. F. J. Gomez and R. Miikkulainen. Solving non-Markovian control tasks with neuroevolution. In *Proc. IJCAI 99*, Denver, CO, 1999. Morgan Kaufman.
7. F. J. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. In *ECML 2006: Proceedings of the 17th European Conference on Machine Learning*. Springer, 2006.

8. Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. Technical Report NC-TR-96-048, NeuroCOLT, 1996.
9. N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.
10. S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
11. M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2004. (On J. Schmidhuber's SNF grant 20-61847).
12. J.F. Kolen and J.B. Pollack. Back propagation is sensitive to initial conditions. *Advances in neural information processing systems*, 3:860–867, 1991.
13. S. Lange and M. Riedmiller. Deep Auto-Encoder Neural Networks in Reinforcement Learning. *IJCNN*, 2010.
14. D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.
15. J.B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–105, 1990.
16. N. Saravanan and David B. Fogel. Evolving neural control systems. *IEEE Expert*, pages 23–27, June 1995.
17. Tom Schaul, Tobias Glasmachers, and Jürgen Schmidhuber. High dimensions and heavy tails for natural evolution strategies. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2011.
18. J. Schmidhuber. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412, 1989.
19. J. Schmidhuber. Recurrent networks adjusted by adaptive critics. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, Washington, D. C.*, volume 1, pages 719–722, 1990.
20. J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3 (NIPS 3)*, pages 500–506. Morgan Kaufmann, 1991.
21. Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.
22. R. Sutton and A. Barto. *Reinforcement learning: An introduction.* Cambridge, MA, MIT Press, 1998.
23. R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063. MIT Press, 2000.
24. P. J. Werbos. Neural networks for control and system identification. In *Proceedings of IEEE/CDC Tampa, Florida*, 1989.
25. D. Wierstra and J. Schmidhuber. Policy gradient critics. In *Proceedings of the 18th European Conference on Machine Learning (ECML 2007)*, 2007.
26. Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Congress on Evolutionary Computation (CEC)*, 2008.
27. Xin Yao. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 4:203–222, 1993.