Learning Context-free Grammars: Capabilities and Limitations of a Recurrent Neural Network with an External Stack Memory*

Sreerupa Das
Department of Computer Science
University of Colorado
Boulder, CO 80309
rupa@cs.colorado.edu

C. Lee Giles
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
giles@research.nec.nj.com

Guo-Zheng Sun
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
sun@sunext.umiacs.umd.edu

Abstract

This work describes an approach for inferring Deterministic Context-free (DCF) Grammars in a Connectionist paradigm using a *Recurrent Neural Network Pushdown Automaton (NNPDA)*. The NNPDA consists of a recurrent neural network connected to an external stack memory through a common error function. We show that the NNPDA is able to learn the dynamics of an underlying pushdown automaton from examples of grammatical and non-grammatical strings. Not only does the network learn the state transitions in the automaton, it also learns the actions required to control the stack. In order to use continuous optimization methods, we develop an analog stack which reverts to a discrete stack by quantization of all activations, after the network has learned the transition rules and stack actions. We further show an enhancement of the network's learning capabilities by providing hints. In addition, an initial comparative study of simulations with *first*, *second* and *third* order recurrent networks has shown that the increased degree of freedom in a higher order networks improve generalization but not necessarily learning speed.

Introduction

Considerable interest has been shown in language inference using neural networks. (For more traditional approaches to inference of grammars see [Miclet 90].) Recurrent networks in particular, with various training algorithms, have proved successful in learning regular languages, the simplest in the Chomsky hierarchy. Work by [Elman 90], [Giles 90], [Mozer 90], [Pollack 91], [Servan-Schreiber 91], [Watrous 92], and [Williams 89] have demonstrated that the recurrent nature of these networks is able to capture the dynamics of the underlying computation automaton. [Giles 92a] and [Watrous 92] have used higher order (higher dimensional weights) recurrent neural networks with no hidden layer and showed that such models are capable of learning state machines and appear to be at least as powerful as any multilayer network. Using a heuristic clustering method, [Giles 92a] showed that finite state automata could be extracted from the neural networks both during and after training. [Giles 92b] successfully demonstrated a method for learning an *unknown* grammar.

This work is concerned with inference of DCF grammars - moving up the Chomsky hierarchy. This recurrent neural network model, previously described by [Sun 90] and [Giles 90], has an external stack memory integrated through a hybrid error function, hence making it powerful enough to learn DCF grammars. Previous work by [Williams 89] showed that, given both the training set and action information of the read/write head of a Turing Machine, a recurrent network is capable of learning the finite state machine part of the Turing Machine that recognizes the training set. The model described here learns both the stack control (*push*ing and *pop*ing of the stack) and the state transitions of the underlying finite state automaton of the pushdown automaton. This is performed by extracting information only from the training data. The learning capabilities of the inferred Pushdown Automaton is enhanced by providing more information, *hints*, about the training strings. For other work on the use of recurrent neural networks for DCF inference, see [Allen 90] and [Pollack 90].

The stack is *external* and *continuous*. The reason for using an external stack, as opposed to an internal one, [Pollack 90], is that the external stack requires lesser resources for training. The continuous part permits the use of a continuous optimization method, in our case gradient-descent. We present a brief description of the model, discuss the dynamics of the stack action and give simulation results of learning performance.

Neural Network Pushdown Automaton (NNPDA)

The network consists of a set of fully recurrent neurons, called *State Neurons* which represent the states and permit classification and training of the NNPDA. One of the state neurons is designated as the *Output*

*To be published in the Proceedings of The Fourteenth Annual Conference of The Cognitive Science Society, July 29 – August 1, 1992, Indiana University.

a	.4
b	.5
c	.8
..	..

Table 1: *Left column indicates the content of the stack; Right column indicates the quantity of each alphabet on stack. Top of the stack is a.*

Neuron. The *State Neurons* get input (at every time step) from three sources, namely, from their own recurrent connections, from the *Input Neurons* and from the *Read Neurons.* The *Input Neurons* register external inputs to the system. These external inputs consist of sequences of characters of strings fed in one character at a time. The *Read Neurons* keep track of the symbol(s) on top of the stack. One non-recurrent neuron, called the *Action Neuron* indicates the stack action (push, pop or no-op) at any instance. The continuous valued activation of this neuron is used to perform analog actions (namely push and pop) on the stack. The architecture of the Neural Network is shown in *Figure 1.*

Many appropriate error functions could be devised. The one we chose to train the network consists of two error functions: one for legal strings and the other for illegal strings. For legal strings we require 1). the N-NPDA must reach a final state and 2). the stack must be empty. This criterion can be reached by minimizing the error:

$$Error = 1/2[(1 - S_o(l))^2 + L(l)^2] \tag{1}$$

where $S_o(l)$ is the activation of an $OutputNeuron$ with its target value for legal strings as 1.0 and $L(l)$ is the stack length, all after a string of length l has been presented as input a character at a time. For illegal strings, the error function is modified as:

$$Error = S_o(l) - L(l) \quad if \ (S_o(l) - L(l)) > 0.0 \tag{2}$$

otherwise $Error = 0.0$. Equation (2) reflects the criterion that, for an illegal pattern we require either the final state $S_o(l) = 0.0$ or the stack length $L(l)$ to be greater than 1.0.

Stack Control

The analog stack is external to the network and is manipulated by the action neuron with continuous activation values. Since the activation of the action neuron is continuous valued, the pushing and popping is also continuous. Associated with each element on the stack is an analog value. An example of the stack would be the one shown in *Table 1.* It has 0.4 of a stacked over 0.5 of b and so on. Operations on the stack are determined by the activation of *Action Neuron*, S_a. The value of S_a is allowed to vary between +1 and −1. The operations will be described as follows:

PUSH: If the activation of $ActionNeuron$, S_a is significantly positive the action taken is *push.* In our simulations we performed *push* when the magnitude of $S_a > 0.1$. In case of *push* the current input is pushed on the stack and its value is determined by the magnitude of the activation of $ActionNeuron$. Therefore, for the stack shown in *Table 1*, $S_a = 0.6$ and the current input is c, then, after the operation, the stack would appear as shown in *Table 2.*

c	.6
a	.4
b	.5
c	.8
..	..

Table 2: *After pushing* 0.6 *of* c *onto stack shown in Table 1.*

a	.1
b	.5
c	.8
..	..

Table 3: *After poping* 0.9 *from the stack in Table 2.*

POP: If activation of $ActionNeuron$ is sufficiently negative, the action taken is *pop.* In this case, quantities stored on the stack are removed up to a depth denoted by the magnitude of S_a. Therefore, for the stack in *Table 2* and $S_a = -0.9$, after the *pop* operation stack would appear as shown in *Table 3.* For our simulations we performed *pop* if $S_a < -0.1$.

READING from the stack: At every time step (or with processing of every element of the input string), the information on *top of the stack* has to be updated every time an action is taken. This is done as follows. All the elements on the top of the stack up to a *depth* of 1.0 (i.e., all the symbols whose quantities add up to 1.0 from the top) are considered. Then their individual quantities on the stack are used as the corresponding activations of the *Read Neurons* in the next time step. For example, the *Read* information of the stack shown in *Table 3* would be $R_a = 0.1$; $R_b = 0.5$, $R_c = 0.4$ if we consider only three input symbols. It should be noted that our goal is to train the network to take the correct actions, and as training proceeds all magnitudes of S_a should approach 1 or 0. Hence, the quantities of symbol pushed and popped on the stack would also approach 1. Thus, after training, a specific *reading* of the stack should contain only *one* symbol and the performance of the analog stack should approximate that of a discrete one.

NO OPERATION: If the magnitude of S_a is significantly small, no operation is taken. For our simulations we performed a *no-operation* if $-0.1 < S_a < 0.1$.

Training of the NNPDA

The activation of *State Neurons* (and *Action Neuron*) may be written as

$$S(t+1) = F(S(t), I(t), R(t); W) \tag{3}$$

where I is the activation of the *Input Neurons* and R is the activation of the *Read Neuron* and W is the weight matrix of the network. We use a localized representation for Input and Read symbols (thus, a symbol is uniquely represented by a vector which has only one 1 and all other elements 0). We now describe the different forms equation (3) take for different orders of the *State, Read and Input Neurons.*

For *First Order*, let V(t) represent a concatenation of vectors $I(t)$, $R(t)$ and $S(t)$, i.e., $V(t) = I(t) \oplus R(t) \oplus S(t)$. Then equation (3) becomes

$$S_i(t+1) = g(\sum W_{ij} V_j(t)) \tag{4}$$

For *Second Order*, let V(t) represent concatenation of vectors $I(t)$ and $R(t)$, i.e., $V(t) = I(t) \oplus R(t)$. Equation (3) becomes

$$S_i(t+1) = g(\sum\sum W_{ijk} S_j V_k(t)) \tag{5}$$

For *Third Order* equation (3) becomes

$$S_i(t+1) = g(\sum\sum\sum W_{ijkl} S_j(t) I_k(t) R_l(t)) \tag{6}$$

where $g(x) = 1/(1+exp(-x))$.

At the end of each input sequence of alphabets $a_0, a_1, a_2, ...a_{l-1}$, a distinct symbol called the end-marker is presented to the network. The activation of the *Output Neuron* at this point is compared with the *Target*. The end symbol is useful because there may be more than one final state and we want to accept a string whenever the string reaches *some* final state. The end symbol facilitates computation by effectively constructing an extra hidden layer. Adjusting the weights connected to the end symbol neuron (since the input has a local representation, only one input neuron turns on to represent a symbol) corresponds to the training of a *super*-final state.

There are two *coupled* functions that the network needs to learn in the process of training: the state transition function and the stack manipulation function. During training, input sequences are presented one at a time and activations are allowed to propagate until the end of the string is reached. Once the end is reached the *Target* is matched with the *Output Neuron* and weights are updated in accordance with the learning rule. The learning rule used in the NNPDA is a significantly enhanced extension to *Real Time Recurrent Learning* [Williams 89].

For the First-order network, using the objective function defined by equation (1) and (2) in a gradient-descent weight update expression $\Delta W_{ij} = -\eta \partial Error/\partial W_{ij}$, the weight update rule becomes

$$\Delta W_{ij} = \begin{cases} \eta((Target - S_o(l))\partial S_o(l)/\partial W_{ij} - \\ \quad L(l)\partial L(l)/\partial W_{ij}) \quad \textit{for equation 1} \\ -\eta(\partial S_o(l)/\partial W_{ij} - \partial L(l)/\partial W_{ij}) \\ \quad\quad \textit{for equation 2} \end{cases} \tag{7}$$

where η is the learning rate. Then, $\partial S_o(l)/\partial W_{ij}$ can be calculated from the following recurrence relation by setting $\partial S_m(0)/\partial W_{ij} = 0.0$.

$$\partial S_m(t+1)/\partial W_{ij} = g'(\delta_{mi} V_j(t) + \sum W_{mn} \partial S_n(t)/\partial W_{ij} + \sum W_{mn} \partial R_n(t)/\partial W_{ij}) \tag{8}$$

where $\delta_{mi} = 1\ if\ m = i$, $g' = d(g(x))/dx$.

How do we obtain $\partial R(t)/\partial W_{ij}$? Since the current stack reading depends on its entire history, no simple recurrence relation can be found. However, the following approximation appears valid. It may be noted that we are able to differentiate R only because the stack is continuous. Also, after the network has been trained sufficiently and action values are large (> 0.5), each reading may not contain much information of the past. We obtain an approximate value of $\partial R(t)/\partial W_{ij}$ as follows:

$$\partial R(t)/\partial W_{ij} = (\partial R(t)/\partial S_a(t))(\partial S_a(t)/\partial W_{ij})$$

where $S_a(t)$ is the activation of the *Action Neuron.*

During *push* and *pop*, any incremental (or decremental) change of ΔS_a in S_a would cause an increase (or decrease) of R in the top of the stack with the same amount. Therefore,

$$\partial R_i/\partial S_a = 1$$

if R_i corresponds to the symbol on top of the stack. Also, since the total reading length (equal to 1) is fixed, any incremental (or decremental) change of ΔS_a in S_a would also cause a decrease (or increase) of R in the bottom of the stack. Hence,

$$\partial R_i/\partial S_a = -1$$

if R_i corresponds to the symbol at the bottom of the stack. It may be noted that, these are only first order approximations with the assumption that the network has been trained sufficiently so that actions are large in magnitude (close to 1.0).

Therefore $\partial R_m(t)/\partial W_{ij}$ may be approximated as:

$$\partial R_m(t)/\partial W_{ij} \approx (\delta_{mr_1} - \delta_{mr_2})\partial S_a(t)/\partial W_{ij}) \tag{9}$$

where r_1 and r_2 are the indices of the symbols on top and bottom of the stack respectively, and $\delta_{mr_i} = 1$ if $m = r_i$. Having defined $\partial R(t)/\partial W_{ij}$ and assuming all partial derivatives at $time = 0$ to be 0, $\partial S_m(l)/\partial W_{ij}$ can be evaluated, where l is the length of the input string being processed.

Since the stack length $L(t)$ may be recursively evaluated by

$$L(t+1) = L(t) + S_a(t) \tag{10}$$

the second partial derivative, $\partial L(l)/\partial W_{ij}$, in equation (7) may be expressed as

$$\partial L(t+1)/\partial W_{ij} = \partial L(t)/\partial W_{ij} + \partial S_a(t)/\partial W_{ij} \tag{11}$$

For an initial condition let $\partial L(0)/\partial W_{ij} = 0.0$, then $\partial L(l)/\partial W_{ij}$ can be evaluated by the above recursion. Therefore, by imposing the "on-line" learning algorithm, the derivatives of the weights are propagated forward using the recursive formula and the final correction ΔW_{ij} is made at the end, after one whole input string has been presented. The learning rules for second and third order networks are exactly the same in nature but vary in the type of interconnections or the W matrix.

To determine the time complexity of the learning algorithm, let S and I be respectively the number of fully-connected recurrent and input neurons and l the length of the input string. Then the number of operations required per time step is of the order $l * (S+1)^2 * (S+R) * (I+S+R)$ for a first-order recurrent network (primarily dominated by the computation of the partial derivatives in equation (8)) The 1 in $S+1$ takes into account the action neuron. Similarly a second and a third order network require respectively $l * S^2 * (S+1)^2 * (I+R)^2$ and $l * S^2 * (S+1)^2 * I^2 * R^2$. Note that for large S, the complexity goes as $O(S^4)$.

Learning with Hints

Our training sets contained both positive and negative strings. One problem with training on incorrect strings is that, once a character in the string is reached that forces the string to a *reject state*, no further information is gained by processing the rest of the string. For example, if we are training the network on language $a^n b^n$ and we come across a string that begins with *aaaaba...*, no matter what follows the last a in the string, it is unnecessary to parse and train the network on rest of the string any further. In order to incorporate this idea we have introduced the concept of a *Dead State*.

During training, we assumed that there is a *teacher* or an *oracle* who has some knowledge of the grammar and is able to identify the points on the strings (of negative examples) that takes the strings to a *reject state*. When such a point is reached in the input string, further processing of the string is stopped and the network is trained so that one designated *State Neuron* called the *Dead State Neuron* is "on". To accommodate the idea of a *Dead State* in the learning rule, the following change is made: if the network is being trained on illegal strings that end up in a *Dead State* then the length $L(l)$ in the error function in equation (1) is ignored and simply becomes $Error = 1/2(Target - S_o(l))^2$. Since such strings have an illegal sequence, they cannot be a prefix to any legal string. Therefore at this point we do not care about the length of the stack.

For strings that are either legal or illegal but do not go to a dead state (an example of such a string would be a prefix of a legal strings, that ends prematurely); the objective function remains the same as described earlier in equation (1) and equation (2). *Hints* in this form made learning faster, helped in learning of exact pushdown automata and made better generalizations. For certain languages, these *hints* actually made learning possible. There are methods for inserting hints (rules) directly into recurrent neural networks [Omlin 92]; it would be interesting to see the effect of using these methods in training a NNPDA.

Simulations

The training data consisted of sequence of strings generated in *alphabetical order* from the input alphabet set. *Incremental, real-time learning* was used to train the NNPDA. In other words, the length of the strings in the training set was increased in steps, gradually as the network learned the smaller ones. At the beginning of each run the weights were initialized with a set of random values chosen between [-1.0, 1.0]. Training began with the shortest possible strings (of length *one*).

Once the network learned to recognize the strings in the current training set, longer strings (of length one more than the longest string in the current set) were added to the training set. Longer strings were added when either of the two criteria was satisfied: (1) a threshold number of epochs were completed, (2) network learned to recognize all strings in the training set before completing the threshold number of epochs. Epochs here imply one pass over the training set. A training set was considered to be successfully learned when all the strings in the set were recognized correctly. In general, for every language trained, this threshold was varied until the performance (in terms of total number of epochs needed for training) could not be further increased. For most simulations, the threshold for the number of epochs ranged between 20 and 40.

If the correct stack actions are learned by the NNPDA, then adding longer strings would not increase the error. This was used to estimate an upper bound for the maximum length of training strings to be used. The maximum length of the strings required for training was usually limited to *ten*. For simple languages like $a^n b^n$, training strings of length up to *six* were sufficient to train the NNPDA. For a particular length, since the number of positive strings was much smaller than the number of possible negative strings, a positive string of the same length was placed every third string in the training set. Thus, a small set of positive strings were repeated many times in the training set. Once the network was trained, the actions and states were quantized so as to *extract* a perfect pushdown automaton. This *extracted pushdown automaton* can recognize strings of arbitrary length. For a discussion of this extraction method, see [Sun 90] and [Giles 90] and, more recently, for finite state automata [Giles 92a].

The same simulation criteria and initial conditions described above were used for training NNPDA of various orders. A comparative performance of the networks of *first, second* and *third* orders in terms of number of iterations required, generalization capability and number of neurons are shown in *Tables 4, 5 and 6.* The values in the tables were typical ones obtained in our simulations; changing the initial conditions resulted in values of similar orders of magnitude. These tables show statistics for the *minimal machines* learned.

Conclusions

A neural network pushdown automaton (NNPDA) was constructed by connecting a recurrent neural network state controller to an external stack memory through a joint error function. This NNPDA was shown to be capable of learning a range of small, but interesting, deterministic context-free (DCF) grammars. A *continuous* external stack was constructed that permitted the successful use of continuous optimization methods (gradient-descent). The NNPDA learned to make efficient use of this stack. When it was trained on regular languages, e.g. (*single parity*, where the odd or even occurrence of a single symbol is checked for acceptance), the network learns the state transitions without making use of the stack. However, a language like parity could have been learned using a stack, that is, it could have used the stack by *pushing* a symbol on every odd occurrence of a character and *popping* the stack on every even occurrence. But the NNPDA error function apparently allows the network to selectively avoid using the stack when the language can be learned without it.

Simulations varying the order of the recurrent network showed that, in general, the higher the order of the net, the easier it was to learn grammars. (For some grammars, higher order proved to be a necessity for successful training!) However, it proved possible to learn a simple DCF Language such as the *parenthesis matching* grammar by using only *first-order* networks. We also observed that the stack was able to learn to change its stack actions. For example, in learning the language $a^n b^n c b^m a^m$, the stack had to learn to push a's and push b's when it saw an a and then reverse that process. Third order networks do not necessarily perform much better than second order networks. One possible explanation is that in the higher order networks the increase in the degrees of freedom slows down convergence. Of course the network has only learned small DCF grammars; larger grammars should be much more difficult. However, the NNPDA was able to learn how to efficiently control and use *an external stack* while at the same time learning its neural network state machine controller.

Acknowledgement

The authors would like to acknowledge helpful and useful discussions with M. Goudreau, C. Miller, M. Mozer, C. Omlin, H. Siegelmann, P. Smolensky and D. Touretzky.

References

[Allen 90] Allen, R.B., 1990. Connectionist Language Users. *Connection Science* 2(4): p 279.

[Elman 90] Elman, J.L., 1990. Finding Structure in Time. *Cognitive Science* 14:p. 179.

[Giles 90] Giles, C.L.; Sun, G.Z.; Chen, H.H.; Lee, Y.C.; Chen, D., 1990. Higher Order Recurrent Networks & Grammatical Inference. *Advances in Neural Information Systems 2*, D.S. Touretzky (ed), Morgan Kaufmann, San Mateo, Ca:p. 380.

[Giles 92a] Giles, C.L.; Miller, C.B.; Chen, D.; Chen, H.H.; Sun, G.Z.; Lee, Y.C., 1992. Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks. *Neural Computation* 4(3):p. 393.

[Giles 92b] Giles, C.L.; Miller, C.B.; Chen, D.; Sun, G.Z.; Chen, H.H.; Lee, Y.C., 1992. Extracting and Learning an *Unknown* Grammar with Recurrent Neural Networks. *Advances in Neural Information Systems 4*, J.E. Moody, S.J. Hanson, R.P. Lippmann (eds), Morgan Kaufmann, San Mateo, Ca.

[Miclet 90] Miclet, L., 1990. Grammatical Inference, *Syntactic and Structural Pattern Recognition; Theory and Applications*, H. Bunke and A. Sanfeliu (eds), World Scientific, Singapore, Ch 9.

[Mozer 90] Mozer, M.C.; Bachrach, J., 1990. Discovering the Structure of a Reactive Environment by Exploration. *Neural Computation* 2(4):p. 447.

[Omlin 92] Omlin, C.W.; Giles, C.L., 1992. Training Second-Order Recurrent Neural Networks Using Hints. Proceedings of the Ninth International Conference on Machine Learning, D. Sleeman and P. Edwards (eds). Morgan Kaufmann, San Mateo, Ca.

[Pollack 90] Pollack, J.B., 1990. Recursive Distributed Representations. *J. of Artificial Intelligence* 46:p. 77.

[Pollack 91] Pollack, J. B. 1991. The Induction of Dynamical Recognizers. *Machine Learning* 7:p. 227.

[Servan-Schreiber 91] Servan-Schreiber, D.; Cleeremans, A.; McClelland, J.L., 1991. Graded State Machine: The Representation of Temporal Contingencies in Simple Recurrent Networks. *Machine Learning* 7:p. 161.

[Sun 90] Sun, G Z.; Chen, H.H.; Giles, C.L.; Lee, Y.C.; Chen, D., 1990. Neural Networks with External Memory Stack that Learn Context-Free Grammars from Examples. Proceedings of the Conference on Information Science and Systems, Vol. II: p. 649. Princeton University, Princeton, NJ: Conference on Information Science and Systems, Inc.

[Watrous 92] Watrous, R.L.; Kuhn, G.M., 1992. Induction of Finite-State Languages Using Second-Order Recurrent Networks, *Neural Computation* 4(3).

[Williams 89] Williams, R.J.; Zipser, D., 1989. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* 1(2):p. 270.

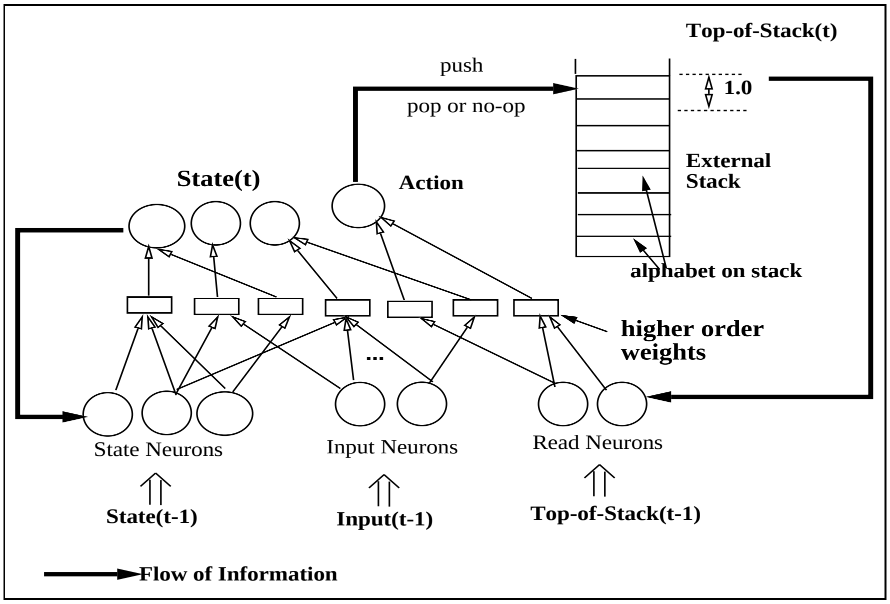

Figure 1: *The figure shows the architecture of a third-order NNPDA. Each weight relates the product of Input(t-1), State(t-1) and Top-of-Stack information to the State(t). Depending on the activation of the Action Neuron, stack action (namely, push, pop or nooperation) is taken and the Top-of-Stack (i.e. value of Read Neurons) is updated.*

Order	parenthesis		$a^n b^n$		$a^n b^n c b^m a^m$		$a^{n+m} b^n c^m$	
of NN	hints	w/o hints	hints	w/o hints	hints	w/o hints	hints	w/o hints
1st	50–100	***	300–500	***	***	***	***	***
2nd	50–80	80–100	150–300	300	500	***	200-250	***
3rd	50–80	50–80	150–250	150–250	150	***	150–250	***

Table 4: *Iterations required by first, second and third order networks to learn various languages with and without hints and under same initial conditions, namely, same initial learning rate, same initial value of state neurons, same random number and same input set ("***" in the table implies that the simulation did not converge).*

Order	parenthesis		$a^n b^n$		$a^n b^n c b^m a^m$		$a^{n+m} b^n c^m$	
of NN	hints	w/o hints	hints	w/o hints	hints	w/o hints	hints	w/o hints
1st	0.0	***	8.9	***	***	***	***	***
2nd	0.0	3.07	0.0	2.67	5.56	***	0.0	***
3rd	0.0	0.0	0.0	1.03	3.98	***	0.0	***

Table 5: *Generalization (in % error on all possible strings up to length 15, starting from length 1, that is, with 65534 strings).*

Order	parenthesis		$a^n b^n$		$a^n b^n c b^m a^m$		$a^{n+m} b^n c^m$	
of NN	hints	w/o hints	hints	w/o hints	hints	w/o hints	hints	w/o hints
1st	3+1	***	3+1	***	***	***	***	***
2nd	1+1	2	1+1	3	1+1	***	1+2	***
3rd	1+1	2	1+1	2	1+1	***	1+1	***

Table 6: *Minimal number of State Neurons required to learn the languages in various orders (for the simulations with hints one neuron was required explicitly for dead state and hence the "+1"s).*