Learning a deterministic finite automaton with a recurrent neural network

Laura Firoiu[†] **(lfiroiu@cs.umass.edu)**
Tim Oates[†] **(oates@cs.umass.edu)**
Paul R. Cohen[†] **(cohen@cs.umass.edu)**
[†] Computer Science Department, LGRC, University of Massachusetts, Box 34610
Amherst, MA 01003-4610

Abstract

We consider the problem of learning a finite automaton with recurrent neural networks, given a training set of sentences in a language. We train Elman recurrent neural networks on the prediction task and study experimentally what these networks learn. We found that the network tends to encode an approximation of the minimum automaton that accepts only the sentences in the training set.

1 Introduction

1.1 The problem of inducing a deterministic finite automaton (*DFA*)

The interest in *DFA* inference is partly induced from the larger goal of explaining how humans learn the grammar rules of their native language. There have been debates on whether children learn in an unsupervised mode, just by listening to other language speakers, or if they have innate knowledge of language. Therefore, it is an interesting problem to see what can be learned just by “listening to others”, that is, from a set of grammatically correct sentences. While the complex syntactic rules of natural language cannot be encoded efficiently as regular grammar productions, fragments of language can be represented by finite automata. Thus this representation which is preferred due to its simplicity, is adequate for some small languages.

1.2 Deterministic Finite Automata(*DFA*) and Recurrent Neural Networks (*RNN*)

In spite of the low complexity (uniformity) of the representation, the problem of learning finite automata turned out to be quite difficult. Different kinds of finite automata induction problems have been studied:

- Gold ([5]) showed that if the language is not known to be finite, then grammar induction from even infinite sequences of sentences in the language is not always possible.
- Gold ([6]) showed that the problem of finding the smallest automaton consistent with a finite set of accepted and rejected strings from the language is NP-complete; Pitt and Warmuth ([11]) showed that even a *DFA* which is at most polynomially larger than the optimum solution is still hard to find; Kearns and Valiant ([7]) showed that the learning problem remains difficult even when a different hypothesis space is searched.

In this work we study the problem using simple recurrent neural networks to induce a regular grammar that is consistent with a *training set* of sentences from the language, i.e. from positive evidence only. This problem is different from the problem of finding the minimum *DFA* consistent with a set of positive and negative examples. The difference comes from the absence of negative examples and explicit assumption about the sentences that are not in the language. This absence renders the problem under-specified and may lead to one of the two extreme assumptions:

- If it is assumed that negative examples do not exist then the language consists of all the sentences over the alphabet, so the minimum *DFA* that represents the language is the automaton with just one state.
- If it is assumed that the negative examples are all the sentences that are not in the training set then the language is completely defined by the training set and there is a polynomial time algorithm on the sample size that builds a finite automaton that accepts exactly this language. We call this automaton the *training set DFA* . This *DFA* has a unique state for each prefix of the sentences in the training set and can be minimized in polynomial time.

The problem resembles the conditions of the Gold's theorem on the impossibility of language learning from positive evidence, with the provision that the sequence of examples cannot be infinite.

Recurrent neural networks and deterministic finite automata have similar behaviors. They are both state devices and their state functions have the same form : $state(t) = f(state(t-1), input\_symbol(t))$. It has been shown (see [13]) that there is an immediate encoding of a *DFA* with n states and m input symbols into a simple recurrent neural network with $m \times n$ state units. Conversely, a *DFA* can be easily extracted from such a *RNN* . Neural networks have the attractive property of being trainable devices, due to the backpropagation algorithm and its variants. Given their representational and learning capabilities, recurrent neural networks are a natural choice for the tasks of *DFA* encoding or induction.

Due to this property, different network architectures, for example second order networks (with multiplicative units) have been used for the task of *DFA* induction. For output-less automata, the network can be trained either with the word following the current input (*the prediction task*) as in [3] , or if both positive and negative examples are present, with a target encoding the membership in the language of the current string as in [4] . For automata with output, the natural choice of the target is the output symbol of the current transition.

As in [1]and [3] we use an Elman recurrent network trained on the prediction task to induce a *DFA* , and are interested in understanding what automaton is learned by the network. In section 2 we describe in more detail the setting of the learning task, the experiments and results. In section 3 we observe that the network learns an approximation of the minimum *DFA* of the training set and conclude that this network architecture and training regime are biased towards the extreme case where the training set is the entire language.

2 Experiments

2.1 The grammar and the languages

We wrote a small context free grammar (CFG), that generates natural-sounding sentences, similar with the one used in [3] . Some of the constraints imposed syntactically by this grammar are subject-verb and noun phrase-relative clause number agreement and transitive/intransitive verb distinctions.

From this CFG we obtained a regular grammar by expanding the start symbol with all possible productions, up to an arbitrary depth in the derivation trees. The resulting regular grammar can generate only a subset of the original language. Furthermore, the regular grammar is used to generate sentences with bounded length. A fragment of the initial context free grammar is given in figure 1. The sub-languages we generated from this grammar are:

- “elm_r1_d4” has 56 sentences like “Mary sees .”, “the girls see .”, “John and John walk .”
- “elm_r2_d5” has 512 sentences and includes the previous language. Some of its sentences are “ Mary feeds the girl .”, “Mary chases Mary and Mary .”, “the cat sees the girl .”, “the boy who walks sees .”.

S	→	NP_hs VP_hs .	S	→	NP_hp VP_hp .
S	→	NP_as VP_as .	S	→	NP_ap VP_ap .
NP_hs	→	the {boy, girl}	NP_hs	→	{John, Mary}
NP_hp	→	the {boys, girls} RC_hp	RC_hp	→	who VP_hp
VP_hp	→	chase OBJ	VP_hp	→	feed OBJ
OBJ	→	NP_hs	OBJ	→	NP_hp

Figure 1: Part of the CFG that generates the sub-languages used for training the network.

Some of the above sentences are unusual in natural language. A constraint like “chase” is a verb that requires a direct object could be easily incorporated in the context-free grammar rules by creating a separate variable for this verb category and then adding the corresponding productions, with at most a double increase in the number of the existing productions. On the other hand, a constraint that avoids sentences like “Mary and Mary ...” cannot be expressed without an exponential increase of the number of context free productions. For this reason, we decided to allow such sentences in the languages we used as training sets.

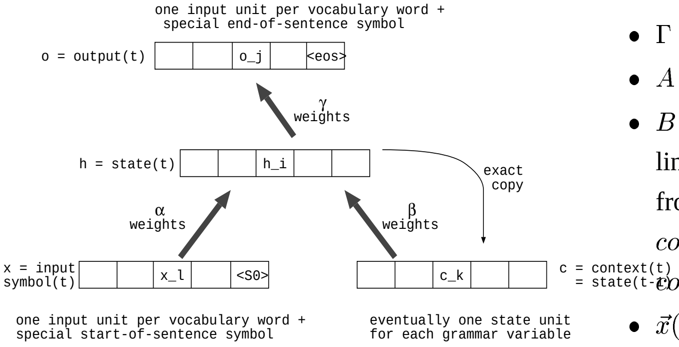
2.2 Network Architecture and Training

We use Elman recurrent neural networks, with the architecture and notations as in figure 2. The activation function of the hidden and output nodes is the sigmoid $\sigma(x) = 1/(1 + exp(-x))$.

The network is trained with the backpropagation through time algorithm (see [14]) and with the cross-entropy error function.

All sentences in the training set are presented to the network, one word at each time step. There is one input unit for each word of the alphabet. An extra symbol, “start_of_sentence”, marks the beginning of each sentence. This extra symbol plays the role of the starting state of the automaton. Because the network state is set to 0 at each beginning of sentence, this additional input unit provides the initial context. The (approximately) periodic reset and the short sentences help avoid the state instability mentioned by Kolen in [8] . Thus, we avoid the solution proposed by Giles in [10] , that relies on large weights and biases, because we noticed that the network learns better when it starts with small weights. The network is trained for the prediction task: the word following the input word in the current sentence represents the target output. There is also one output unit for each word in the alphabet, and a special marker “end_of_sentence”.

The cost to be optimized is the cross-entropy error function. As described in [12] , the cross-entropy function forces the network to learn the probability distribution over the next words, conditioned on the input symbol and the network: $P(x(t+1) \mid N, x(t))$. The network weights are updated on-line, with the backpropagation through time algorithm. We noticed that for input sequences of more than one hundred words, batch training yields large update



one input unit per vocabulary word +
special end-of-sentence symbol
o = output(t)
o_j
<eos>
γ
weights
h = state(t)
h_i
exact
copy
α
weights
β
weights
x = input
symbol(t)
x_l
<S0>
c_k
c = context(t)
= state(t-1)
one input unit per vocabulary word +
special start-of-sentence symbol
eventually one state unit
for each grammar variable

- n = the number of hidden units = the number of context units.
- m = the number of input units = the number of output units.
- $\Gamma = \{\gamma_{i,j}\}$ are the hidden to output weights.
- $A = \{\alpha_{l,i}\}$ are the input to hidden weights.
- $B = \{\beta_{k,i}\}$ are the weights of the recurrent links,
  from context to hidden units, where $context(t) = state(t-1), t \geq 1$ and $context(0) = 0$.
- $\vec{x}(t)$ is the input vector at time $t \geq 1$
- $\vec{h}(t)$ is the network state at time $t \geq 1$
- $\vec{o}(t)$ is the output vector at time $t \geq 1$
- $\vec{c}(t) = \vec{h}(t-1)$ is the context at time $t, t \geq 1$

Figure 2: Elman network architecture and notation.

values of the weights, thus leading quickly to saturation and no further learning. A learning rate of .1 and momentum of .9 gave the best results in terms of network convergence. For each training set, we chose the number of hidden units to be approximately equal to the number of states in the associated *DFA* . The weight update equations for this architecture and training regime are given in appendix A.

2.3 *DFA* extraction

One method used so far for extracting a *DFA* from a *RNN* , for example see [4] , is to assume that the network states whose values are close in R^n form well separated clusters that represent the automaton states. We rely on the same assumption and obtain the *DFA* states by hierarchically clustering the network states into a binary tree and then by merging the siblings, provided they satisfy a distributional criterion. This additional criterion is necessary for two reasons: first, to stop the merging from reaching the root of the tree, thus yielding a one-state automaton, and second to correct eventual “network errors”, that is states that have close values but cannot encode equivalent *DFA* states. This criterion tests for the similarity of the distributions over the state classes that follow and precede the two classes. The initial state classes are formed by the identical network states. We use a G statistic, which has a χ^2 distribution (see [2]) to test if there is a statistically significant difference in the two probability distributions. At the beginning of merging, because there are too many degrees of freedom (the number of states), and each state is usually preceded and followed by only one state, respectively, the G statistic is not effective. But as merging proceeds, irrespective of the distributional criterion, the number of states in each class increases, while the number of degrees of freedom decreases and the G statistic becomes effective and eventually stops the process.

Alternatively, we can view each word occurrence in a regular language as being indexed by the pair of states between which it transits. The language alphabet becomes the set of terminal (word) instances that occur between two states of its automaton. The language with the set of symbols thus modified becomes a *Szilard* language of a regular grammar. As described in [9] , this kind of languages has a polynomial inference algorithm that induces the grammar from a set of positive examples. Obviously, any regular language can be "Szilard-ified" in this manner. If we consider the network states at times $t-1$ and t as the additional index on the current input symbol, $x(t)$ we can treat the unknown target language as Szilard:

- represent the word instance $x(t)$ as the concatenation of the previous and current network states $\langle \vec{c}(t), \vec{h}(t) \rangle$
- cluster the word instances, based on this representation
- consider each cluster a set of terminals that appear only on transitions between the same pair of states (the Szilard property) and apply the inference algorithm to construct the *DFA* ; conflate the occurrences of the same word in a cluster into the original terminal.

The resulting *DFA* does not necessarily represent a Szilard language of a regular grammar, because instances of the same terminal can appear on different transitions. This merging technique obtains the "Szilard-ified" version of an unknown regular language that contains all the sentences in the training set.

2.4 Results

We trained the network several times with each of the chosen sub-languages. Three methods were used for *DFA* extraction:

- direct mapping of clusters of network states to the *DFA* states; this automaton is noted the *state DFA* .
- direct mapping of clusters of the concatenation of the previous and current network states to the *DFA* states; this method tries to avoid "network errors": two network states are further distinguished by part of the paths that lead to them, namely by their previous states; this has the effect of a more conservative merging, that eventually avoids clustering together two states that get close values accidentally (states that are not actually equivalent in the *training set DFA* . this automaton is noted the *prev_state DFA* .
- constructing the Szilard version of the unknown target language, as described in section 2.3; this automaton is noted the *Szilard DFA* .

The results in table 1 compare the induced automata for each of the training sets with the corresponding minimized *training set DFA* .

It can be noticed in table 1 that while for the smaller language the *training set DFA* is almost always recovered, for the larger language the resulting automaton generates many more sentences than there are in the training set. Some of these sentences, like "the boy feeds Mary and John ." are in the language of the original CFG and represent correct generalizations. Other sentences are plainly incorrect, like "John lives walks lives lives hears.", which is generated by a cycle in the induced *DFA* .

3 What DFA is extracted from the trained network ?

If the network always starts in the same state at the beginning of the sentence, then the *RNN* will reach the same state for all unique prefixes of the sentences in the training set. For example, the prefix "a b" from the sentences "a b c ." and "a b d ." will lead the network to the same state.

$$s_0 \xrightarrow{a} s_1 = f(a, s_0) \xrightarrow{b} s_2 = f(b, s_1)$$

language	*training set DFA*	*state DFA*	*prev_state DFA*	*Szilard DFA*
elm_r1_d4	8 states 56 sent.	8 states 0 extra sent.	8 states 0 extra sent.	7 states 32 extra sent. (all correct generalizations) : “the boy and John hear .” “the boy and Mary hear .” ⋮
		8 states 0 extra sent.	8 states 0 extra sent.	9 states 0 extra sent.
elm_r2_d5	22 states 512 sent.	23 states 849 extra sent. :	21 states 298 extra sent. :	23 states 4068 extra sent. :
		18 states 1188 extra sent. : “John and John feed the boy .” “John chases Mary and John walk .” “John lives walks lives lives hears .” ⋮	20 states 260 extra sent. : “the boy feeds Mary and John .” “the cat lives the boy . “ “John and Mary .” ⋮	21 states 1560 extra sent. : “John and Mary and John see .” “John and John who walk see .” “the boys who live hear Mary .” ⋮

Table 1: The original and induced automata for the two increasingly complex sub-languages of the CFG grammar. The “extra sent.” in the induced automata is the number of sentences accepted by these *DFA* , which are not in the corresponding training set. The sentences were generated by imposing a limit of $d+1$ words on the sentence length, where d is the maximum sentence length in the training set.

Because the network states belong in principle to a continuous space, it is not likely that two states corresponding to two different prefixes will be identical, unless the weights are adapted to encode precisely such an equality. So, for random weights, we can assume a one-to-one mapping between the set of sentence prefixes and the network states, and name each state with its associated prefix.

It follows that the *RNN* will construct the prefix tree of the training set, as it can be seen in figure 3.

This implies that the network tries to learn the probability distribution over the next words conditioned on the current prefix :

$$P(x(t+1) \mid N, x(t)) = P(x(t+1) \mid h_w, x(t)) = P(x(t+1) \mid h_w)$$

For the simple example in figure 3 the network states are: $\{h_0, h_a, h_{ab}, h_{ac}, h_{ab.}, h_{ac.}\}$. The states in a finite automaton are also defined by the strings of words that label the paths that lead to them from the start state. For example, in figure 3, state q_1 is defined by “a”, while q_2 is defined by the set $\{$“a b”, “a c”$\}$. It follows that the network states can be partitioned in groups corresponding to the states of the training set DFA. For the example in figure 3 the network states $\{h_{ab}, h_{ac}\}$ correspond to the *DFA* state q_2.

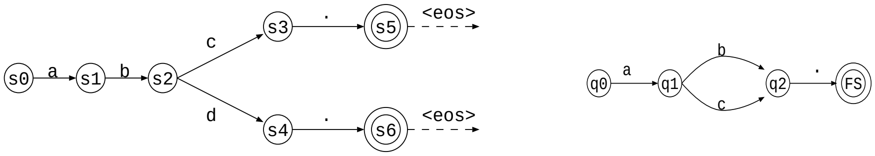


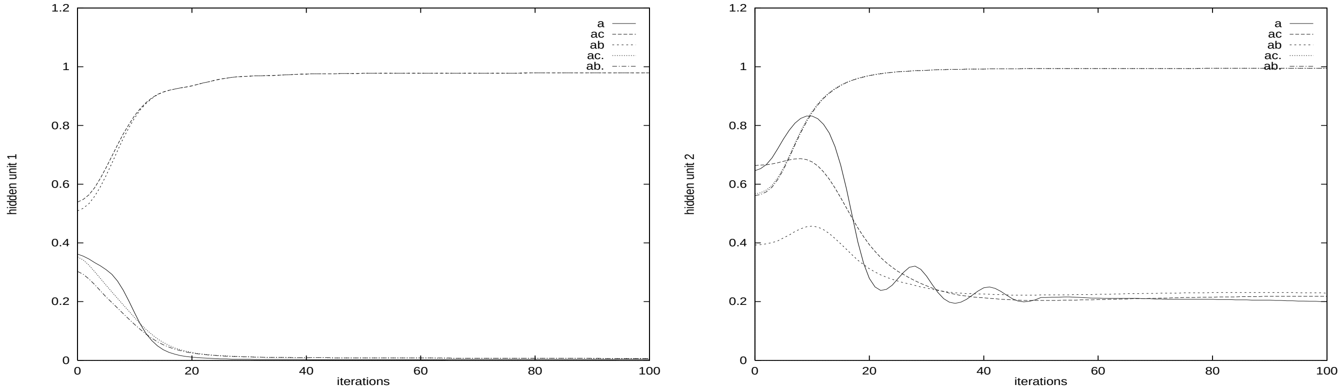
Figure 3: The prefix tree of the training set $\{$“a b .”, “a b .”$\}$ and the *DFA* that represents it.

Due to the cross-entropy error function, the training process will adapt the network weights such that each state will describe the probability distribution over the next words. For a prefix w, this constraint is described by the equation $\sigma(\Gamma * h_w) = P(next\_word \mid w)$. Because the sigmoid function σ is injective, if two strings w_1 and w_2 have the same distributions over the next words, then the network will ideally assign two states h_{w_1} and h_{w_2} such that $\Gamma * h_{w_1} \approx \Gamma * h_{w_2}$. It follows that a possible solution is $h_{w_1} \approx h_{w_2}$. So, the network will eventually assign similar values for the states associated with prefixes that can be followed by the same words. This approximate merging is reminiscent of the classical *DFA* minimization algorithm. The difference is that while the *DFA* minimization algorithm, according to the definition of equivalent states, considers the entire paths from each state to the final state, the network is trained only with the words immediately following the two states and thus has a different criterion for merging. This criterion, i.e. the probability distribution over the next words, is weaker than the finite automaton state equivalence criterion, because it relies on less information. It follows that it can lead to merges of states which are not equivalent in the *DFA* of the training set and thus to generalizations beyond this set.

For some training sets, for example like that in fig. 3, the two criteria are equivalent: the first word of the path following a state uniquely identifies the path. For such cases, the network training process can converge to an approximate encoding of the training set *DFA* : the network states $\{h_{w_i}, h_{w_j}, \ldots\}$ corresponding to one *DFA* state might converge to values close in metric space. On the other hand, because the backpropagation algorithm may find only a local minimum of the error function, it can happen that not all the states with the same probability distribution over the next words are assigned similar values. Thus, states that are equivalent in the training set *DFA* might not get close values and thus are not merged. These phenomena can be seen in the examples in figures 4 and 5.

For the training set $\{$“a b .”, “a c .”$\}$, it can be seen in figure 4 how the encodings in the hidden layer of the states

$\{h_{ab}, h_{ac}\}$ and $\{h_{ab.}, h_{ac.}\}$ converge towards the same values, respectively.



Evolution of the first hidden unit during the first 100 iterations.

Evolution of the second hidden unit during the first 100 iterations.

The values of the hidden units after 20000 iterations, for all the prefixes in the training set:

prefix:	"a"	"ac"	"ac."	"a"	"ab"	"ab."
hidden unit 1	0.054	1.000	0.001	0.054	1.000	0.001
hidden unit 2	0.062	0.123	1.000	0.062	0.123	1.000

Figure 4: Convergence of the network states to an encoding of the states of the *DFA* that represents the training set for a network with two states.

For the small network in in figure 4, the evolution of states can be traced by examining the weight update equations. In order to explain, for example, how the values of the first hidden unit for the prefix states h_{ab} and h_{ac}, which have the same probability distribution over the next words, vary together in time, as in figure 4, we look at the weights $\alpha_{c,1}$ and $\alpha_{b,1}$. The prefix states occur at time steps 2 and 4. Because these states are preceded by the same state h_a , the difference in their activation values is given only by the difference of the two weights. It follows that the two weights, $\alpha_{c,1}$ and $\alpha_{b,1}$ must also have the same variation in time. From the fact that the network is reset at time 3 and from the update equations in section A it follows that the updates of these weights for one sweep in the training set, are :

$$\triangle\alpha_{c,1} \propto \underbrace{h_1(2)[1-h_1(2)]}_{positive}\{infl_{h_1}(2) + \underbrace{\sum_{a=1}^{n}\beta_{1,a}h_a(3)[1-h_a(3)]infl_{h_a}(3)]\}}_{part2}$$

$$\Delta\alpha_{b,1} \propto \underbrace{h_1(4)[1-h_1(4)]}_{positive}\{infl_{h_1}(4) + \underbrace{\sum_{a=1}^{n}\beta_{1,a}h_a(5)[1-h_a(5)]infl_{h_a}(5)]\}}_{part2}$$

If we consider that at the beginning of the training the weights are small, the second term can be ignored in both equations. Thus, the more important terms are $infl_{h_1}(2)$ and $infl_{h_1}(4)$. Because the two prefix states are followed by the same symbol, '.', the two terms become :

- $infl_{h_1}(2) = \gamma_{1,.} - \sum_{j=1}^{m}\gamma_{1,j}o_j(2)$
- $infl_{h_1}(4) = \gamma_{1,.} - \sum_{j=1}^{m}\gamma_{1,j}o_j(4)$

The values of the output units for the two states are:

- $h_{``ab''} : o_j(2) = \sigma(\sum_{i=1}^{n} \gamma_{i,j}\sigma(\alpha_{c,i} + const(B, context)))$
- $h_{``ac''} : o_j(4) = \sigma(\sum_{i=1}^{n} \gamma_{i,j}\sigma(\alpha_{'b',i} + const(B, context)))$

From the above formulas it follows that $o_j(2)$ and $o_j(4)$ cannot be far apart, because the weights $\alpha_{c,i}$ and $\alpha_{'b',i}$, which are the only non-equal quantities in the above formulas, are small. Furthermore, the differences in the output values induced by these weights are reduced by the double application of the sigmoid function. It follows that eventually, the two values $infl_{h_1}(2)$ and $infl_{h_1}(4)$ will have the same sign, and thus the weights $\alpha_{c,1}$ and $\alpha_{b,1}$ will change in the same direction. Once this happens, these weights will influence the output in the same direction and the process continues.

For larger training sets and networks, the evolution of states, weights and output values becomes even more difficult to follow, but it can be watched in experiments. For the language "elm_r2_d5" we look at three sets of prefixes, { "the cats who hear", "the boys who see" }, {"the cats who hear hear", "the boys who see see"} and {"Mary", "the boy"}, each set being characterized by a different probability distribution. The evolution of two hidden units, arbitrarily chosen, in the states associated with each prefix is depicted in figure 5. It can be noticed that the states in the same set did not all converge to close values. For example, the states "the boy" and "Mary" get different representations for those two units, despite their common next word probability distribution. On the other hand, the states of the first two sets do get similar values although they should be distinguished.

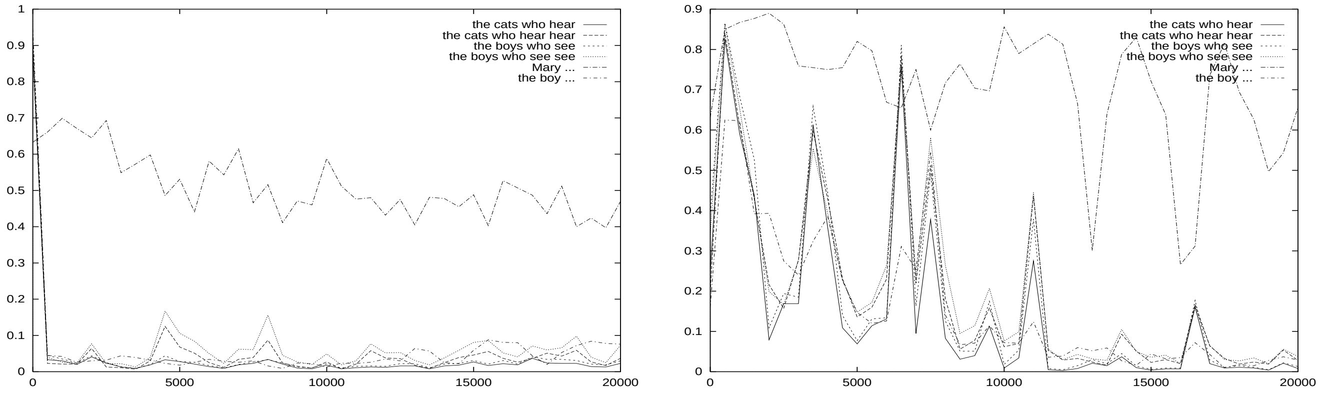


Figure 5: Evolution of hidden units 9 and 17 of the states associated with prefixes { "the cats who hear", "the boys who see" }, {"the cats who hear hear", "the boys who see see"} and {"Mary", "the boy"}.

4 Conclusions

We addressed In the experiments conducted, simple recurrent neural networks were trained with sets of sentences from a regular language. The results indicate that the network is biased towards encoding the minimum automaton of the training set. For a very small network, an informal analysis of the learning process supports this finding. This bias is more evident for smaller training sets and networks. For more complex training sets, the network might not be able to correctly minimize the *training set DFA* and these errors may lead sometimes to correct generalizations.

A The update equations for batch training for backpropagation through time and cross-entropy cost function

The following formulas use the notations and apply to the network described in section 2.2. As in [12] , the cost function for a set of T input-output vector pairs $\{\langle x(t), d(t)\rangle, 1 \leq t \leq T\}$ is:

$$C = \prod_{t=1}^{T} C_t = \prod_{t=1}^{T} P(\vec{d}(t) \mid \vec{x}(t), N(t)).$$

If the output units are seen as encoding the probability $o_j(t) = P(d_j(t) = 1 \mid \vec{x}(t), N(t))$ then the logarithm of the cost function is: $lnC = \sum_{t=1}^{T}\sum_{j=1}^{m} d_j(t)\ln o_j(t) + [1 - d_j(t)]\ln[1 - o_j(t)]$. It can be noticed that for the prediction task, $d(t), d(t+1), \ldots$ are independent, given the network state, because this state encodes the prefix at each time step. The partial derivative of the C_t component of this function to a parameter η is: $\frac{\partial C_t}{\partial \eta} = \sum_{j=1}^{m}[d_j(t) - o_j(t)] * \frac{\partial net_{o_j}(t)}{\partial \eta}$, where $net_{o_j}(t)$ is the net activation of output unit j.

The weight changes are : $\Delta\eta \propto \sum_{t=1}^{T} \frac{\partial C_t}{\partial \eta}$, with $\eta \in \{\gamma_{i,j}, \alpha_{l,i}, \beta_{k,i}\}$.

By applying the chain rule as in [14] , the weight changes at a time step t are:

- The weights from hidden to output units: $\frac{\partial C_t}{\partial \gamma_{i,j}} = [d_j(t) - o_j(t)]h_i(t)$
- Let $infl_{h_i}(t) = \sum_{j}^{m} \gamma_{i,j}[d_j(t) - o_j(t)]$. Let $\delta_{i,a}$ be the function that is 1 when $a = i$ and 0 otherwise.
- The weights from input to hidden units: $\frac{\partial C_t}{\partial \alpha_{l,i}} = \sum_{a=1}^{n} h_a(t)[1 - h_a(t)]infl_{h_a}(t)\frac{\partial net_{h_a}(t)}{\partial \alpha_{l,i}}$
- The recurrent weights from context to hidden units: $\frac{\partial C_t}{\partial \beta_{k,i}} = \sum_{a=1}^{n} h_a(t)[1 - h_a(t)]infl_{h_a}(t)\frac{\partial net_{h_a}(t)}{\partial \beta_{k,i}}$

The derivatives of the net activations $\{net_a\}$ are defined iteratively:

- $t = 0 : h_a(0) = 0$ and $\frac{\partial net_{h_a}}{\partial \eta} = 0$
- $\frac{\partial net_{h_a}(t)}{\partial \alpha_{l,i}} = \delta_{i,a}x_l(t) + \sum_{b=1}^{n} \beta_{b,a}h_b(t)[1 - h_b(t)]\frac{\partial net_{h_b}(t-1)}{\partial \alpha_{l,i}}$
- $\frac{\partial net_{h_a}(t)}{\partial \beta_{k,i}} = \delta_{i,a}k_k(t-1) + \sum_{b=1}^{n} \beta_{b,a}h_b(t)[1 - h_b(t)]\frac{\partial net_{h_b}(t-1)}{\partial \beta_{k,i}}$

From the above equations it follows that:

- $t = 1 : \frac{\partial net_{h_a}(t)}{\partial \beta_{k,i}} = 0$, $\frac{\partial net_{h_a}(t)}{\partial \alpha_{l,i}} = \delta_{i,a}x_l(1)$
- $t = 2 : \frac{\partial net_{h_a}(t)}{\partial \beta_{k,i}} = \delta_{i,a}h_k(1)$

  $\frac{\partial net_{h_a}(t)}{\partial \alpha_{l,i}} = \delta_{i,a}x_l(2) + \beta_{i,a}h_i(1)[1 - h_i(1)]x_l(1)$

- $t = 3: \dfrac{\partial net_{h_a}(t)}{\partial \beta_{k,i}} = \delta_{i,a} h_k(2) + \beta_{i,a} h_i(1)[1 - h_i(1)] h_k(1)$

  $\frac{\partial net_{h_a}(t)}{\partial \alpha_{l,i}} = \delta_{i,a} x_l(3) + \beta_{i,a} h_i(2)[1 - h_i(2)] x_l(2) + \sum_{b=1}^{n} \beta_{b,a} h_b(2)[1 - h_b(2)] \beta_{i,b} h_i(1)[1 - h_i(1)] x_l(1)$

References

[1] A. Cleeremans, D. Servan-Schreiber, and J.L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1:372–381, 1989.

[2] P. R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, 1995.

[3] J. L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 1992.

[4] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee. Extracting and learning an *unknown* grammar with recurrent neural networks. In *Advances in Neural Information Processing Systems 4*. 1992.

[5] E. M. Gold. Language identification in the limit. *Information and control*, 10:447–474, 1967.

[6] E. M. Gold. Complexity of automaton identification from given data. *Information and control*, 37:302–420, 1978.

[7] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95, 1994.

[8] John F. Kolen. Fool's gold: Extracting finite state machines from recurrent network dynamics. In *Advances in Neural Information Processing Systems 6*, 1994.

[9] E. Makinen. Inferring regular languages by merging nonterminals. Technical Report A-1997-6, Department of Computer Science, University of Tampere, 1997.

[10] Christian W. Omlin and C. Lee Giles. Constructing deterministic finite-state automata in recurrent neural networks. Technical report, Computer Science Department, Rensselaer Polytechnic Institute, 1994.

[11] Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM*, 40(1):95–142, 1993.

[12] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin. Backpropagation: The basic theory. In *Backpropagation: Theory, architectures, and applications*. Erlbaum, 1993.

[13] H. T Siegelmann. *Theoretical Foundations of Recurrent Neural Networks*. PhD thesis, Rutgers, 1992.

[14] P. N. Werbos. *The roots of backpropagation*. John Wiley & Sons, Inc., 1994.