Training a neural network to count: A genetic approach

Bo Yuan

Supervisor: A/Prof Janet Wiles

School of Information Technology and Electrical Engineering
The University of Queensland

Abstract

Recurrent neural networks (RNNs) are capable of learning languages. However, the learning performance is not satisfactory. Recent studies have revealed that the main reason lies in the complex relationship between the weightspace and the dynamics of the RNN. As a result, traditional gradient-based algorithms such as backpropagation through time (BPTT) tend to encounter difficulty. This project tries to solve this problem by considering the application of Genetic Algorithms (GAs) in RNNs training. We first show that a GA can also be used as the training algorithm of a RNN on a simple task. Furthermore, we propose a handcrafted GA that is more suitable for learning the context-free a^nb^n language. The performance of this GA on learning the context-sensitive $a^nb^nc^n$ language is also investigated. We conclude that the GA is a promising training algorithm for RNNs on language learning tasks and more work needs to be done to further improve its performance.

I. Introduction

The broad context of this project is the application of Genetic Algorithms (GAs) in recurrent neural networks (RNNs). The study contains two parts. The first is to show that the GA can be employed in the training of RNNs on some standard tasks such as the Exclusive-OR task and the counting task and the second is to investigate some possible ways to further improve its performance.

RNNs have been shown to be capable of recognizing regular languages from example (Elman 1990). Wiles and Elman (1995) showed how networks could be trained to learn the context-free $a^n\ b^n$ language by backpropagation through time (BPTT). Later studies have revealed that backpropagations tend to encounter instabilities when training on the a^nb^n task (Tonkes and Wiles 1997). Bodén, Wiles, Tonkes and Blair (1999) claimed that the reason lies in the relationship between the dynamics of the network and the weightspace and the dynamics required for the solution lie in a region of weightspace close to a bifurcation point where small changes in weights may result in radically different network behaviour. Furthermore, the error gradient information in this region is highly irregular, which makes it difficult for a gradient-based learning algorithm such as BPTT. Tonkes, Blair and Wiles (1998) suggested a kind of evolutionary hill-climbing algorithm with an error measure that more closely resembles the performance measure. It was reported that the evolutionary algorithm found some different solutions to those found by BPTT. Chalup and Blair (1999) extended this evolutionary approach to the task of predicting the context-sensitive $a^n\ b^n\ c^n$ language.

In this project, we will use another evolutionary approach: Genetic Algorithms as the training algorithm. The work reported here includes some empirical results and dynamics analysis of RNNs. We first show that an ordinary GA is successful in a classical task: the Exclusive-OR problem (Elman 1990). Our subsequent simulations show that a specifically designed GA is also capable of finishing the learning task for the context-free $a^n b^n$ language and may find some interesting solutions. However, this GA still has some difficulty in learning the context-sensitive language. So, the future work will be in the direction of designing an appropriate GA for learning the context-sensitive language.

II. Some Preliminary Simulations

2.1) Simulation One: Exclusive-OR

2.1.1 Issues

In this simulation, we are concerned with the following questions:

A. Can a GA be used to train a RNN on a relatively simple task?
B. How is the GA's performance on this kind of task?

The first question investigates the possibility of using GAs in RNNs training. In other words, GAs should be able to find a solution according to the task object. The second question investigates the performance of GAs compared with traditional training algorithms such as BPTT. Hopefully, GAs should demonstrate good performance in *efficiency*, *reliability*, *accuracy* and the like. Here, we defined *efficiency* as that the GA should be able to find a solution within reasonable time or generations and *reliability* as that the networks (solutions) found should also have good performance on the test set. Finally, *accuracy* was defined by the error between the expected output and the real network output.

2.1.2 Simulation Details

This problem was introduced in the research work of Elman (1990) to study the internal representation of time in RNNs. The training set was a sequence of bits in which the first and the second bits were XOR-ed to produce the third; the fourth and fifth were XOR-ed to give the sixth and so on. This input stream was presented to a RNN shown in Figure 1. The network had one input unit, two hidden units, two context units, providing recurrent connections from the hidden units and one output unit. The task was, at each time point, to predict the next bit in the sequence. In RNNs, the output of networks is based not only on the current input but also on the network's previous state, which is passed back to the hidden unit(s) by the context unit(s). Certainly, it is only possible to predict certain bits (i.e., the network can predict the third one based on the values of the previous two bits). In our simulation, we randomly generated a training set containing 3000 binary bits according to the rule described above. Figure 2 is a sample of the input-output mapping. The first line is the input stream and the second line is the expected output. In the input stream, the first bit (0) and the second bit (1) produce 1 in the third bit through XOR and similarly the fourth bit (0) and the fifth bit (0) produce another 0 in the sixth bit. Given this input stream, a successful network should be able to predict 1 at the second stage, which means the third input should be 1 according to the rule of XOR. However, the output of the network is not deterministic at those stages marked by '?'. That is, when the network receives the first bit, the chance that the next bit will be 1 or 0 is fifty-fifty. We employed an ordinary GA with standard Roulette Wheel selection and two-point crossover. The population size, crossover rate and mutation rate were set to 100, 0.8 and 0.05 respectively. We used 11 real value numbers to represent the network's parameters. The fitness of each individual was evaluated by the average of the absolute value of the difference between the expected output and the real output (predication) on each bit. During evolution, new offspring generated through recombination would replace old individuals with low fitness, keeping the best 20% individuals unchanged. In order to avoid premature convergence, we used asymmetric crossover. That is, we exchanged gene sequences on different parts of the two selected parents. Since we also used mutation, the elitism mechanism was used to make sure that the best individual in current population would be copied to the population of the next generation. Furthermore, we also used catastrophe, randomising individuals with low fitness after every five generations and keeping the best 20% individuals unchanged. Through these efforts, we were able to keep the genetic diversity of the population and find good solutions.

2.1.3 Results

We conducted several simulations by allowing this GA to run for 50 generations and recorded the best solution at the end of evolution. Each time, we could find a solution. A solution was defined as a network that could correctly predict all the predictable bits in the training set. Figure 3 is the graph of a successful network's error on those predictable bits. The network error was calculated by the absolute value of the difference between the network's predication and the real input on each bit. We can see that the error is less than 0.015, which is better than the result by using BPTT, which is around 0.1 (Elman, 1990). In order to fully test this network's performance, we generated a test set containing 3000 bits. We found that this network's performance on the test set was very similar to that on the training set, which means this network had good reliability. Now, we can say that the performance of the GA in this task is satisfactory.

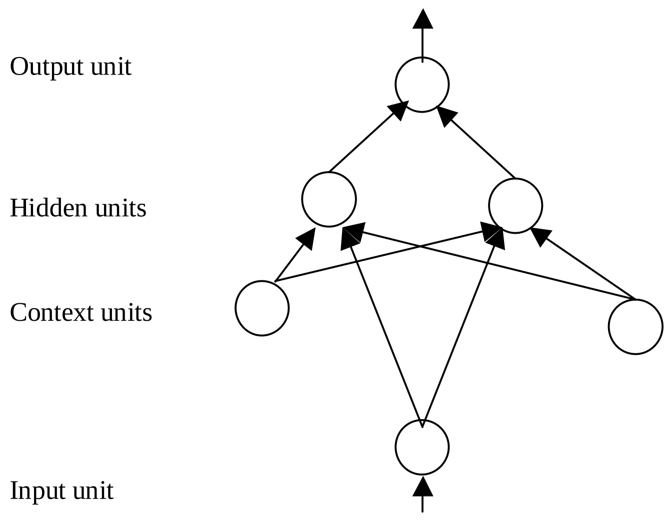

Figure 1: A recurrent neural network with one input unit, two hidden units, two context units and one output unit. The recurrent connections are realized by copying the values of the hidden units to the context units and the output in the next time step is calculated by the values of the context units and the value of the input unit at the next time step.

Input: 0 1 1 0 0 0 1 0 1 1 1 0…
Output: ? 1 ? ? 0 ? ? 1 ? ? 0 ?…

Figure 2: A sample of the input-output mapping of the XOR task. The first raw is the input stream and the second raw is the corresponding expected outputs. At each time stage, the network should be able to correctly predict the next input if it is predictable (i.e., the third bit in each 3-bit input group). Positions not predicable are marked with '?'.

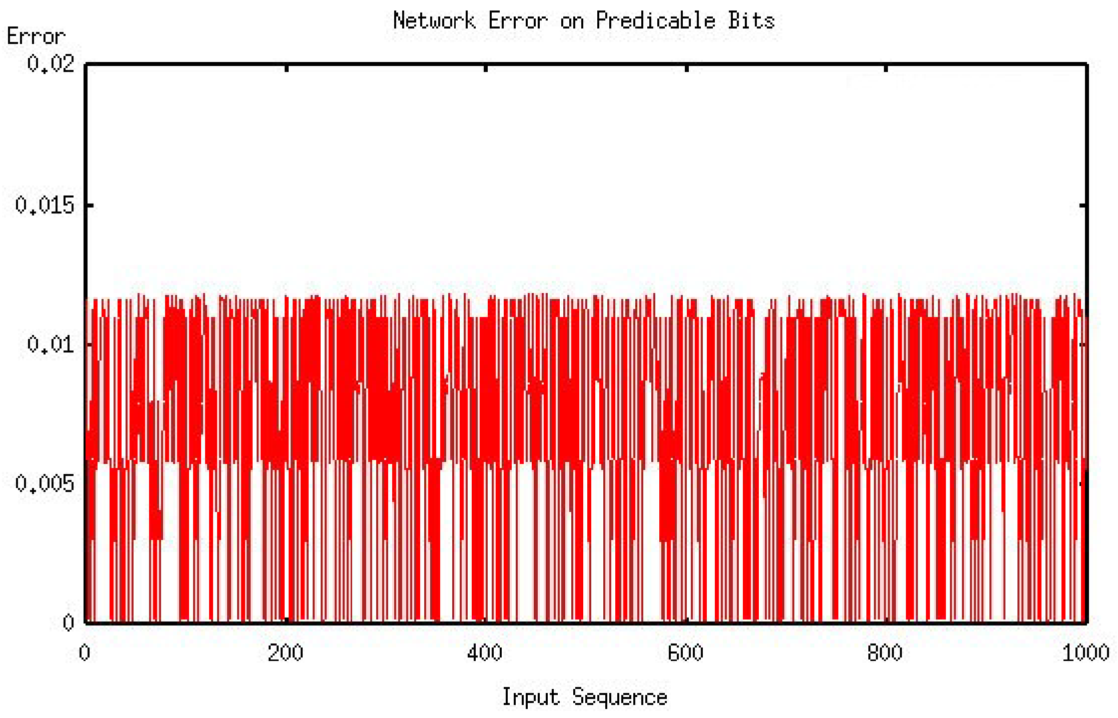

Figure 3: Network's error on those predictable bits in the training set. The error measure is the absolute value of the difference between the expected output (next input) and the real output (predication). It is less than 0.015, which is better than the result by using BPTT.

2.1.4 Analysis

We first present a set of parameters of that network:

	Input Unit	Context Unit1	Context Unit2	Threshold
Hidden Unit1:	8.599200	-5.791498	-3.890805	7.037873
Hidden Unit2:	-6.224250	6.879177	-9.509262	4.600055

	Hidden Unit1	Hidden Unit2	Threshold
Output Unit:	9.757683	6.960967	3.007599

Figure 4: Parameters of a successful recurrent network on the XOR task. The upper part shows the weights between the two hidden units and the input unit and the context units and thresholds of the two hidden units. The lower part shows the weights between the output unit and the two hidden units and the threshold of the output unit.

Next, we will analyse the dynamics of the hidden units under different situations to investigate how the network processes the input sequence. In our simulations, the two context units were set to zero before processing the input sequence and they were never reset. The activation function used in the simulation was a slightly scaled sigmoid function:

f (input)=1/(1+exp(-1.5*input))

As described before, the output of the network is decided by both the current input and the previous states of the hidden units, which were saved to the two context units. So, the network can be interpreted as a dynamical system. In this XOR task, there are totally four input patterns: "0 0 0", "0 1 1", "1 0 1" and "1 1 0". Here we take the second pattern: "0 1 1" as a sample to analyse the state change of units in the network.

Input Pattern: 0 1 1

- When the network receives the first bit (0):

Input (HiddenUnit1)= 0*8.599200+0*(-5.791498)+0*(-3.890805)-7.037873= -7.037873
Input (HiddenUnit2)= 0*(-6.224250)+0*6.879177+0*(-9.509262)-4.600055= -4.600055

Activation (HiddenUnit1)=f (-7.037873)= 2.6015045470049e-005≈ 0
Activation (HiddenUnit2)=f (-4.600055)= 0.0010066878485297≈ 0.001

Value (ContextUnit1)= Activation (HiddenUnit1) ≈ 0
Value (ContextUnit2)= Activation (HiddenUnit2) ≈ 0.001

- When the network receives the second bit (1):

Input (HiddenUnit1)=1*8.599200+ 0*(-5.791498)+0.001*(-3.890805)- 7.037873≈ 1.56
Input (HiddenUnit2)=1*(-6.224250)+0*6.879177+0.001*(-9.509262)- 4.600055≈ -10.8

Activation (HiddenUnit1)=f (1.56) ≈ 0.92
Activation (HiddenUnit2)=f (-10.8) ≈ 0

Value (ContextUnit1) = Activation (HiddenUnit1) ≈ 0.92
Value (ContextUnit2) = Activation (HiddenUnit2) ≈ 0

Input (OutputUnit) = 0.92*9.757683+0*6.960967-3.007599 ≈ 5.9
Activation (OutputUnit) = f (5.9) ≈ 0.9998

- When the network receives the third bit (1):

Input (HiddenUnit1) = 1*8.599200+0.92*(-5.791498)+0*(-3.890805)-7.037873≈ -3.72
Input (HiddenUnit2) = 1*(-6.224250)+0.92*6.879177+0*-9.509262-4.600055≈ -4.55

Activation (HiddenUnit1) = f (-3.72) ≈ 0.0038
Activation (HiddenUnit2) = f (-4.55) ≈ 0.0011

Value (ContextUnit1) = Activation (HiddenUnit1) = 0.0038
Value (ContextUnit2) = Activation (HiddenUnit2) = 0.0011

From the analysis above, we can see that when the network received the fist bit (0), the states of the context units were nearly kept unchanged. However, when the network received the second bit (1), the activation of the two hidden units were 0.92 and 0 respectively, giving 0.9998 as the output, which matched the ideal output 1 very well. When the network received the third bit (1), the states of the two context units returned to a position (0.0038, 0.0011) very close the their initial values (0, 0), getting ready to process the next 3-bit input group. Here, we would like to point out that in order for the trained network to have good reliability, the state(s) of context unit(s) should be able to automatically return to their initial value(s) before processing new strings.

Finally, we will compare the network's error on predicable bits and unpredictable bits.

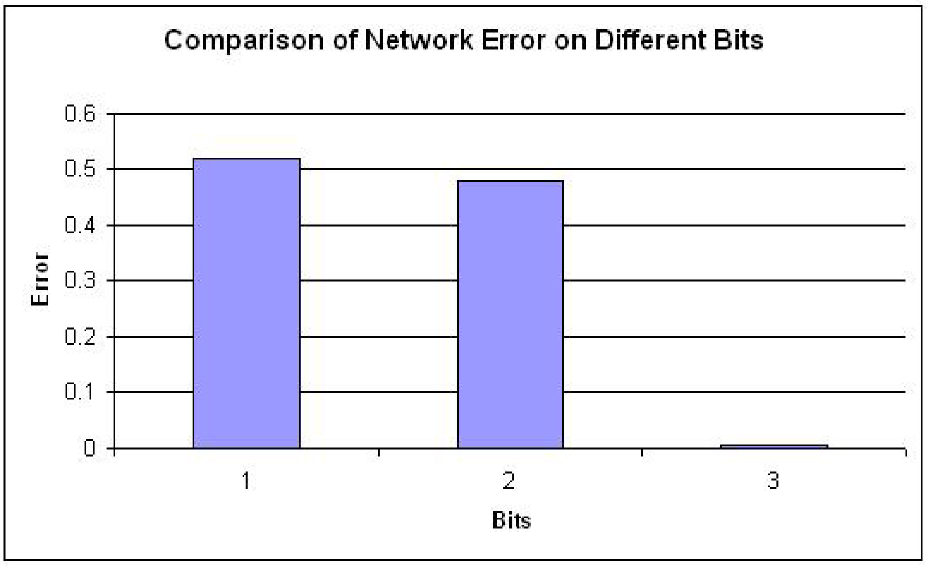

Figure 5: Network's error on different bits. The three columns from left to right represent the network's error on the first, the second and the third bits of the 3-bit input group respectively. Results are averaged over 3000 bits. The error is measured by the mean of the absolute value of the difference between the expected output and the real output.

As described before, the first bit and the second bit of each 3-bit input group in the training set were randomly generated and thus are not predictable. That is, the network's outputs on these bits are not reliable and its error should be approximately in the middle of the range of the output. In fact, from Figure 5, we can see that network's error on the first and second bits are around 0.5, which is roughly in the middle of [0, 1]. As a contrast, when it comes to the third bit, the network's performance is very well. The average error is about 0.007, which means the network has correctly learned the XOR problem.

2.1.5 Summary

In this simulation, we demonstrated how a GA could be used to train a RNN on the XOR task. Although XOR is a relatively simple task, it is a good starting point for us to investigate the possibility and potential of applying GAs in RNNs on other tasks. From the simulation results presented above, we are confident to say that the GA is also capable of training RNNs and may have certain advantages over BPTT. Through further analysis, we also have a good understanding of how the network works.

2) Simulation Two: Context-Free Language a^n b^n

2.2.1 Issues

In this simulation, we try to answer two questions:

(1) Can a GA be used in learning the a^n b^n language, which is a relatively difficult task?
(2) What is the GA's performance on this kind of task?

Since we have demonstrated that a GA could be used in training a RNN on the XOR task, in principle, we can expect that the GA can also be used as the training algorithm on other similar tasks. One of the obvious advantages of GAs over BPTT is that during training GAs may keep the current best solution(s) to the next generation and thus will not lose it. As a contrast, training a network by BPTT after it had found a solution often resulted in the network losing the solution (Wiles and Elman, 1995). However, the GA's performance on the task of context-free language learning cannot be easily predicated. Although the GA has proved to be a kind of excellent global optimisation algorithm and been used in a wide range of areas, there is still a lot of tuning work for each specific problem (No Free Lunch Theorem). In order to find an appropriate GA for a RNN, we need to answer at least the following questions: 1) What kind of diversity maintenance method should we use? 2) What kind of fitness function should we use? 3) What kind of training set should we use? 4) How to assess network's reliability? In this simulation, our purpose was to investigate the possibility of using a GA as the training algorithm and have some understanding about how difficult this task is for the GA. Furthermore, we also wanted to be able to give some suggestions towards how to improve the performance of the GA.

2.2.2 Simulation Details

a^n b^n is one of the simplest context-free languages (Wiles and Elman 1995). It consists of strings of some number of *a*s followed by the same number of *b*s. The number of *a*s is not deterministic but the number of *b*s should match the number of *a*s. The network's task was to take a symbol as its input and to predict the next input. In order to finish this kind of predication task, the network in Figure 1 was extended to provide for the 2-bit input: there were two input units, two hidden units, two context units and two output units. The two input units were set to (0 1) for the '*a*' input and (1 0) for the '*b*' input. As to output, if the activation of the first output unit is less than 0.5 and the activation of the second output unit is greater than 0.5, we say the output is '*a*'; when the activation of the first output unit is greater than 0.5 and the activation of the second output unit is less than 0.5, we say the output is '*b*'. The context units were initialised to zero before processing the first string and were never reset during processing subsequent strings. The training set consisted of 100 randomly generated strings, containing a total of 1034 *a*s and *b*s. These strings conformed to the form a^n b^n, with n ranging from 2 to 12 (meaning length varied from 4 to 24). The input stream and the expected outputs are as the sample in Figure 6.

```
Input:  aaaaabbbbbaaabbbaaaabbbb…
Output: aaaabbbbbaaabbbaaaabbbba…
             *****   ***     ****
```

Figure 6: A sample of the input-output mapping for the a^n b^n language. The sequence shown is $a^5b^5a^3b^3a^4b^4$. Positions in which predication is possible are marked with '*'. During evolution, we only took the network's error on these positions into account.

Since when the input is '*a*', the next input may be either '*a*' or '*b*', so it is not predicable. After the first '*b*' is encountered, the network should be able to predict n-1 *b*s and an '*a*' at the nth step, indicating the beginning of a new string (i.e., n is the number of *a*s). When calculating network error we only took those positions marked with '*' into consideration. That is, during training, we did not care the network's output when the input was '*a*'.

Again, we defined some criteria for our simulation. *Efficiency* means that the training algorithm should be able to find a solution in each run within reasonable generations. That is, the algorithm should not get stuck at local optima but should still be able to converge to a solution. A solution was defined as a set of parameters which corresponding network can correctly predict all the strings in the training set. *Reliability* means that the network should also correctly predict all the strings in the test set. A test set was generated similarly to the training set. For reliability, the test set contains randomly generated strings with the same n range as that of the training set. For example, if the training set contains strings with $2 \leq n \leq 12$, the test set will also contain strings with $2 \leq n \leq 12$. The reason that we employed a large test set is that the states of context units may not be able to return to their exact initial values after processing a string and this error will have some not deterministic influence on the network's behaviour on subsequent strings. We have found that sometimes a network can correctly predict a string of certain length in a certain position of the input stream but may make mistake on the same string in another position. If we use a small test set containing only one copy of each kind of string, we may not be able to test the reliability of the network thoroughly. Another important criterion is *generalization*. A network was considered to be able to generalize (has learned the language) if that network could correctly predict strings longer than the maximum string in the training set.

The network was trained by a standard genetic algorithm using roulette wheel selection. Each individual consisted of 16 real value parameters, representing each weight and threshold by one real value number between –15 and +15. The population size was set to100. The crossover rate and mutation rate were set to 0.8 and 0.05 respectively. During evolution, we kept the current best solution to the next generation and after every five generations, if the improvement of the network's performance was less than a small predefined threshold, we will re-initialise the worst 50 percent individuals in the population. The activation function was a slightly scaled sigmoid function:

$f(x) = 1/(1+\exp(-0.5*x))$

The error measure was the mean square error (MSE). We didn't take the network's error into account when it received '*a*' as input. The reason is if we calculated the network's error at each position, the network often learned the $a^* b^*$ language to minimize error. Each time, we allowed the simulaton to run for 500 generations and recorded the best solution at the end of evolution. We found that the MSE of the best solution was often in the range from 0.015 to 0.03. However, a low MSE didn't necessarily mean a good solution. The MSE of the best solution found so far was 0.0233. After 500 generations, this network learned to predict all but n=2 strings in the training set. It could also predict strings with n=13 and 14, demonstrating limited generalisation ability. The network's reliability was evaluated by a test set including strings with n from 2 to 12. Sometimes, it still made mistakes on some short strings, even those that it had learned in the training set. So, the network was not reliable. Furthermore, the network always predicted '*b*' until the input is the last '*b*' in a string as the input. Although this result was different from those found by other researchers (Rodriguez, Wiles and Elman 1999), it was still acceptable.

2.2.3 Analysis

We first present the parameters of the best solution found so far:

	Input Unit1	Input Unit2	Context Unit1	Context Unit2	Threshold
Hidden Unit1	-1.113773	-4.398785	-12.745903	-12.574694	-5.700705
Hidden Unit 2	-14.834284	4.633869	-0.144200	5.295114	7.903531

	Hidden Unit1	Hidden Unit2	Threshold
Output Unit1	14.890133	14.732658	1.642964
Output Unit2	-14.967956	-14.747307	-1.828822

Figure 7: Parameters of a successful network on the a^n b^n task. The upper part shows the weights between the two hidden units and the two input units and the two context units and the thresholds of the two hidden units. The lower part shows the weights between the two output units and the two hidden units and the thresholds of the two output units.

We are interested in the dynamics of the hidden units under different conditions. Each hidden unit has several kinds of inputs: input units (0 1 for '*a*' and 1 0 for '*b*'), thresholds and self-recurrence. When the input is held constant (for example, a sequence of *a*s or a sequence of *b*s), the only thing that changes is the recurrent activation. Here, we take a string with n=12 as the instance to observe the changes happened in the hidden units.

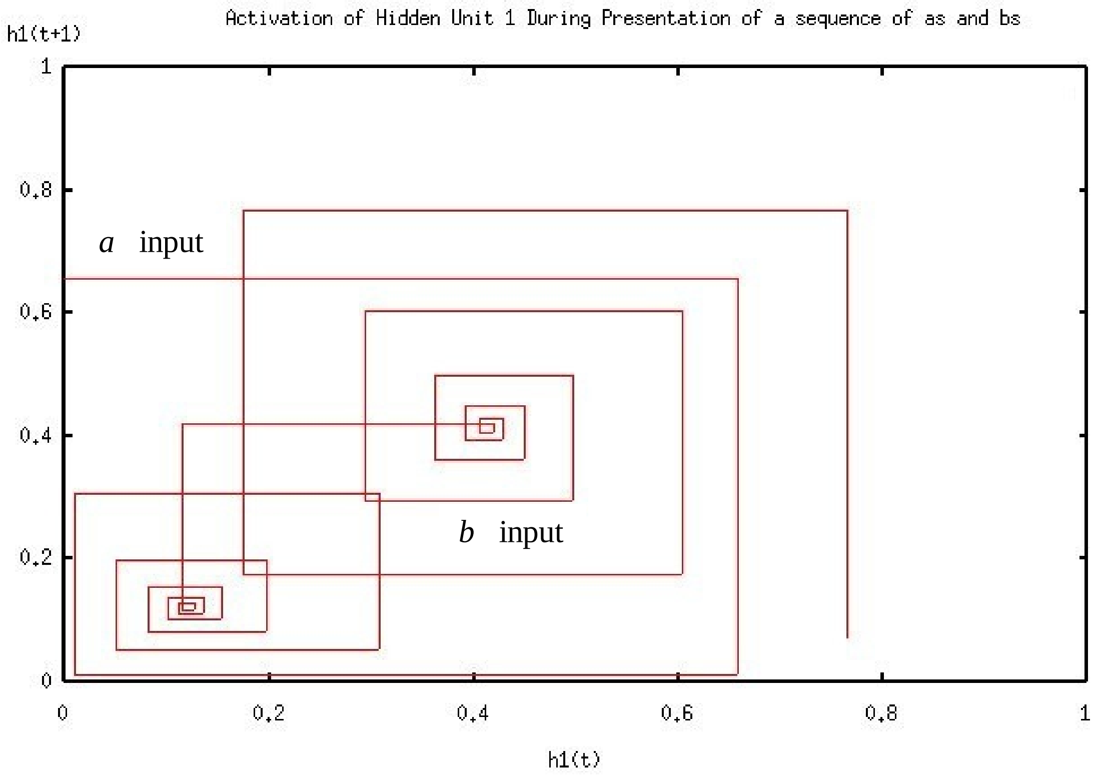

Figure 8: Dynamic behaviour of hidden unit 1 during presentation of 12 *a*s and 12 *b*s. The X-axis represents its state at time t while the Y-axis represents its state at time t+1.This is a typical oscillating behaviour: converging oscillations followed by diverging oscillations. When the input is '*a*', it converges inward; when the input changes to '*b*', it jumps to another fix point and diverges outward.

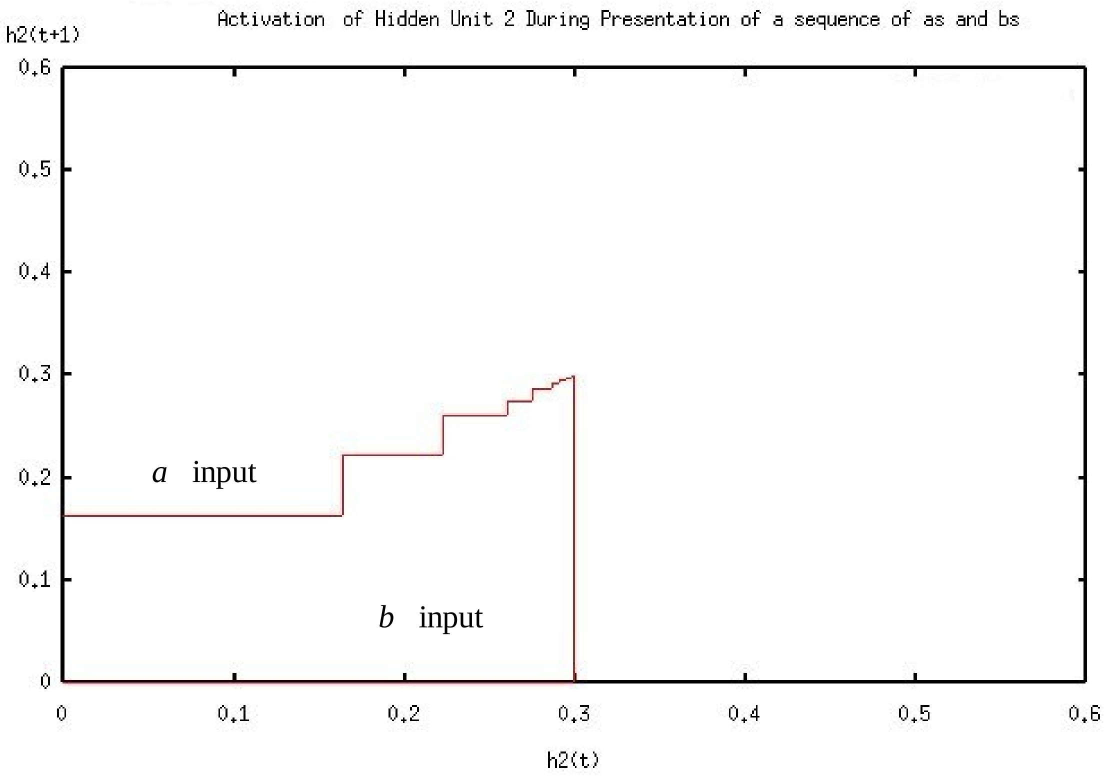

Figure 9: Dynamic behaviour of hidden unit 2 during presentation of 12 *a*s and 12 *b*s. The X-axis represents its state at time t while the Y-axis represents its state at time t+1. When the input is '*a*', its state gradually increases; while the input changes to '*b*', it jumps to a level close to zero and remains at that value.

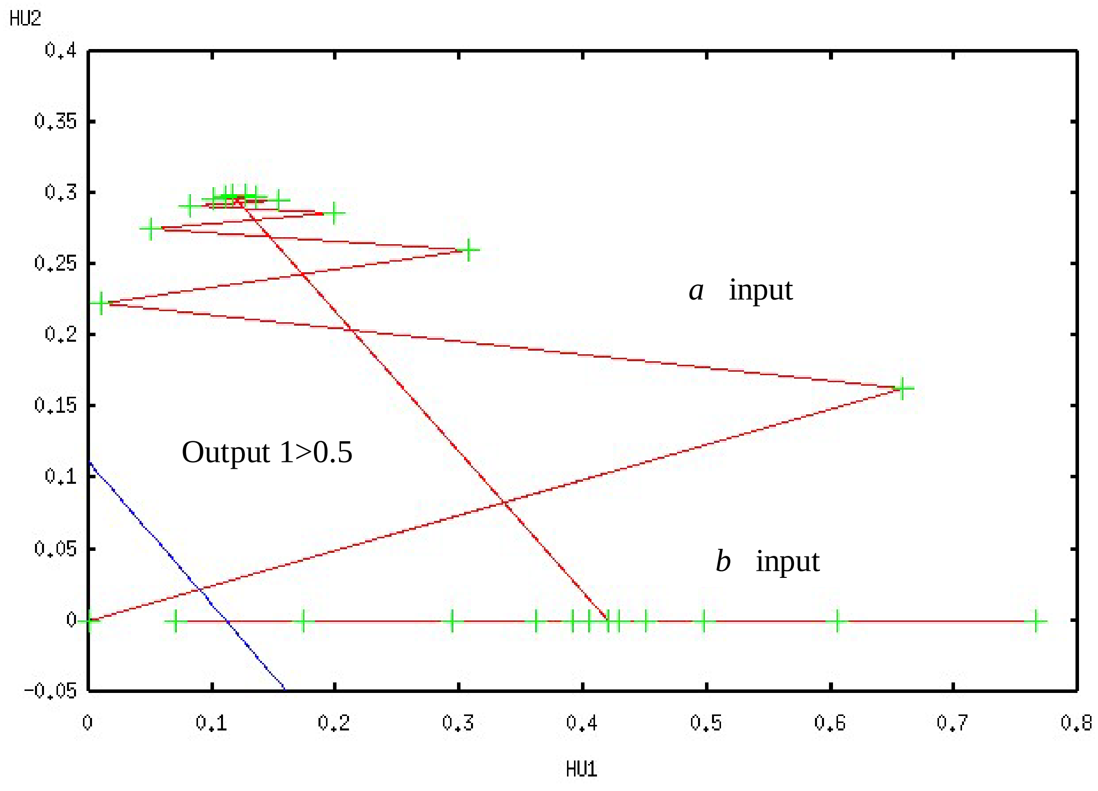

Figure 10: Hidden (context) units trajectories with decision boundary of output unit 1 (the inclined line at the left-bottom corner). The X-axis represents hidden unit 1 while the Y-axis represents hidden (context) unit 2. When the two hidden units are in the area above the decision boundary, the activation of hidden unit 1 is less than 0.5 and vice versa.versa.

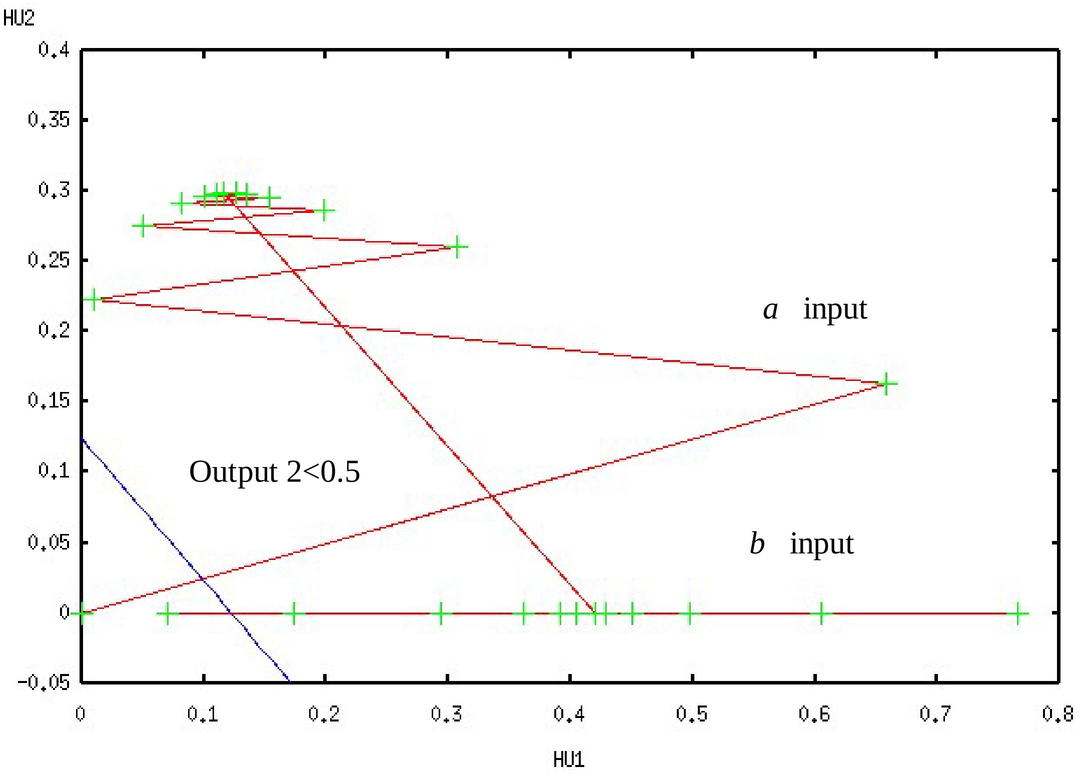

Figure 11: Hidden (context) units trajectories with decision boundary of output unit 2 (the inclined line at the left-bottom corner). The X-axis represents hidden unit 1 while the Y-axis represents hidden (context) unit 2. When the two hidden units are in the area above the decision boundary, the activation of hidden unit 2 is less than 0.5 and vice versa.

Figure 8 shows the the oscillations of the first hidden unit: It oscilates back and forth above and below the middle fixed point. When the input is a string of *a*s, it converges inward. When the input is a string of *b*s, it jumps to another fix point and diverges out for as long as corresponds to the number of *a*s. As a contrast, there is no interesting pattern in the dynamical behaviour of the second hidden unit (see Figure 9). The activation of the second hidden unit gradually increases when the input is '*a*'. When the input is '*b*', the activation of the seond hidden unit is held at the same level around 0.000011. After processing a complete string, the states of the two hidden (context) units will return to a position very close to the initial values of the two context units, ready for a new string. Figure 10 and Figure 11 are hidden (context) units trajectories and output hyperplane for the two output units respectively. When the input is '*a*' or '*b*' except the last '*b*', the two hidden units are always in the area above the decision boundaries. As a result, the activation of output unit 1 is greater than 0.5 while the activation of output unit 2 is less than 0.5. According to our definition before, the output of the network is '*b*'. Similarly, when the input is the last '*b*' in a string, both hidden units are in the area under the decision boundaries, giving '*a*' as the output. Furthermore, we can see that in this network "counting" of both the *a*s and *b*s is performed by the same hidden unit (hidden unit 1), with the other unit (hidden unit 2) used to "switch mode". This kind of soultion has also been found by the evolutionary hill-climbing algorithm but has not been observed by BPTT (Tokens, Blair and Wiles 1998).

Since only one hidden unit is active when processing the input stream, it is pretty straightforward to consider a question: whether a RNN with only one hidden unit will work? To investigate the possibility of using only one hidden unit for the a^n b^n task**,** we conducted additional simulations to address this question.

2.2.4 Additional simulation: one hidden unit

This simulation was similar to the previous one except that there was only one hidden unit (see Figure 12). We found that it was difficult to train such a network using GAs. Only one network learned all the strings in the training set with n<7. Through analysis of the network structure, we found that it is possible to reduce some reductant parameters.

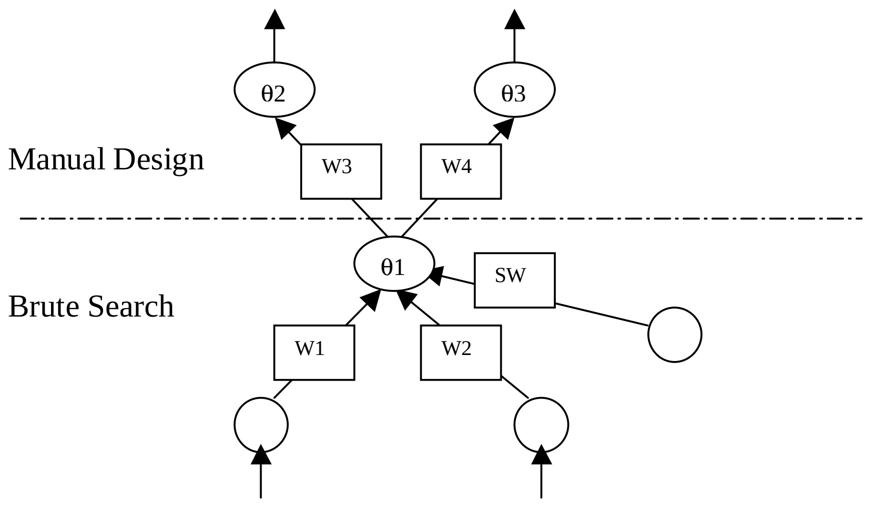

Figure 12: A recurrent network with one hidden unit. W_1, W_2, W_3, W_4 and SW represent weights and θ_1, θ_2, θ_3 represent thresholds.

InputofHidden (t) and Hidden (t) represent the input/output of the hidden unit at time t:

'*a*' input (0 1): InputofHidden(t+1)=0*W_1+1*W_2+SW*Hidden(t)-θ_1
'*b*' input (1 0): InputofHidden(t+1)=1*W_1+0*W_2+SW*Hidden(t)-θ_1

So, we can use W_2' to represent $W_2-\theta_1$ and W_1' to represent $W_1-\theta_1$, deleting θ_1.

The training strategy is: finding a set of parameters (W_1', W_2', SW) through brute search so that there exists a decision boundary (threshold) x when the input is '*b*' except the last '*b*', the output of the hidden unit is greater than it; otherwise the output of the hidden unit is less than it. After we find such a set of parameters and the corresponding threshold, it is easy to manually design other parameters.

For example, suppose x=0.2314, then we can set W_3=10, θ_2=2.314, W_4=-10, θ_3=-2.314. Certainly, we may have many different sets of W_3, W_4, θ_2 and θ_3 as long as they meet the following requirements (Activation means the output of the hidden unit):

For Activation > x

$$\begin{cases} W_3\text{*Activation-}\theta_2>0 \\ W_4\text{*Activation-}\theta_3<0 \end{cases} \quad \text{Output '}b\text{'}$$

For Activation < x

$$\begin{cases} W3\text{*Activation-}\theta_2<0 \\ W4\text{*Activation-}\theta_3>0 \end{cases} \quad \text{Output '}a\text{'}$$

The meaning of the above equations is obvious. Since according to the training strategy, the output of the hidden unit is greater than x when the input is '*b*' except the last '*b*', the output of the first output unit should be greater than 0.5 and the output of the second output unit should be less than 0.5, predicting '*b*' as the next input. Similarly, when the input is the last '*b*', the output of the hidden unit is less than x and the network will predict '*a*'.

The parameters of the best network found in our simulations are:

W1'= -3.8 W2'= -3.2 SW= 7.0 x= 0.279183

The initial value of the context unit was set to 0.25 and the activation function was:

f (x) =1/(1+exp (-0.6*x))

This network successfully predicted all the strings in the training set ($2 \le n \le 12$). In order to investigate how the network works, we generated a test set containing one copy of each kind of string with n from 2 to 12. Those 12 strings were presented in increasing order.

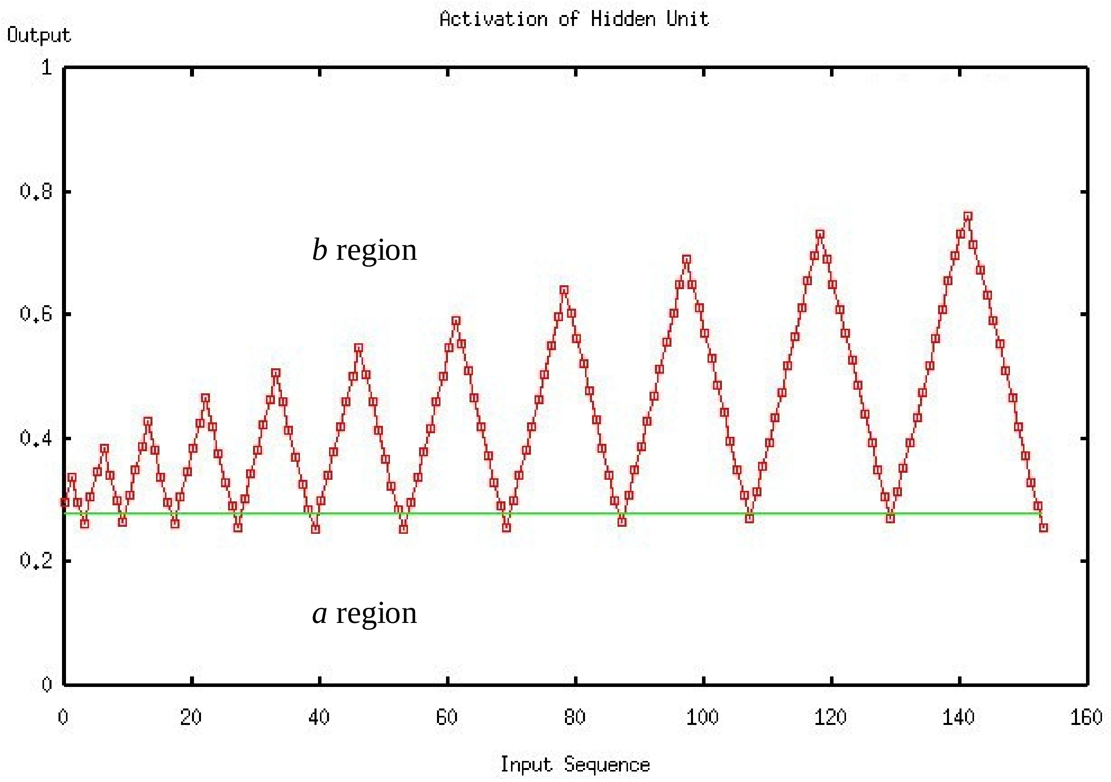

Figure 13: Activation of the hidden unit during presentation of a sequence of *a*s and *b*s. The input sequence contains one copy of each kind of string with n from 2 to 12, presented in increasing order. The horizontal line is the decision boundary (threshold). We can see that this network is a monotonic solution.

From Figure 13 we can see that this network is a monotonic solution. When the input is '*a*', the state of the hidden unit increases gradually and when the input is '*b*', the state of the hidden unit decreases gradually. Furthermore, only when the input is the last '*b*' of the current string, will the state of the hidden unit be less than the decision boundary (i.e., the horizontal line in the graph), predicting '*a*'. Otherwise, the network will always predict '*b*'.

2.5 Summary and some suggestions

Our preliminary work shows that the GA is also capable of training a RNN to learn the $a^n b^n$ language. Furthermore, we also found that even a RNN with only one hidden unit can also be trained by a GA to finish the predication task. However, the overall training task is still difficult for ordinary GAs. Here are some issues arising from our simulations:

1. It has been observed that the error landscape of RNNs is very complex and there are strong dependencies between weights (Bodén, Wiles, Tonkes and Blair 1999). Although the GA is an excellent optimisation algorithm, it still needs a lot of tuning.

2. What is a good training set? Generally, there are two kinds of training sets: static set and incremental set. Static set contains a large number of randomly arranged strings with varying length. It is obviously time-consuming for training and another question is: if the network makes an error at one position, this error may influence the rest strings. Even if a network is actually capable of processing strings with length up to N, it may also make mistakes on strings with length no more than N if we insert some strings longer than N in the training set. That is, during training, if a network makes a mistake on a certain string, the performance of this network on rest strings is not reliable because the network may not be able to come back to its initial state before processing a new string. The disadvantage of the incremental training set is that the trained network may be sequence-sensitive, having poor performance on the test set.

3. Error measure. Because the performance of networks is evaluated by the maximum length of the string that can be correctly predicted, MSE is not reasonable to be a primary error measure. From our simulations, we also found that an individual with low MSE doesn't necessarily mean a good solution. In fact, during training, we cannot know the network's actual performance by simply observing its MSE. Instead, Number of Correct Strings (NCS) may be suitable (Tonkes, Blair and Wiles, 1998).

Our suggestions are: i) using Tournament Selection instead of Roulette Wheel Selection; ii) using an incremental training set containing only one copy of each kind of string presented in increasing order; iii) using NCS as the primary error measure and MSE as the secondary error measure. By doing so, the training process would be expected to be less time-consuming and the best individual in the population is more likely to be the best network.

III. A successful GA for the $a^n b^n$ Language

3.1 Issues

Now, we have had a good understanding of how the network works and how difficult it is for GAs to train a RNN to learn the $a^n b^n$ Language. Based on the analysis made before, we are going to implement a GA that would be expected to have better performance on this training task.

3.2 Simulation Details

The network architecture was the same as that described in previous simulations: two input units, two hidden units, two context units and two output units. However, there were also some differences. First, we used Tournament Selection instead of Roulette Wheel Selection in order to maintain better genetic diversity and avoid premature convergence. Second, we used ATC (Yuan, 2002) to further improve the genetic diversity in the population. Third, we used a small training set containing one copy of each kind of string with n from 1 to 12. That is, the training set contained totally 12 strings, 156 *a*s and *b*s. These strings were presented in increasing order according to their length (i.e., $abaabbaaabbb\ldots a^{11}b^{11}a^{12}b^{12}$).

Fourth, the mutation rate was set to zero. Fifth, we used binary string instead of real value number to represent each parameter. Each parameter was represented by 15 binary bits with 5 bits for the integer part and 10 for the floating part. Sixth, we used NCS combined with SSE (sum square error) as the fitness measure instead of only SSE. NCS was the primary error measure while SSE was a secondary error measure. In our simulations, NCS was defined as the number of strings that can be correctly predicted, from n=1 up to the longest string that can be correctly predicted. For example, if NCS=6, that means the network can correctly predict the consecutive strings from *ab* to a^6b^6 and the network made incorrect prediction in a^7b^7. SSE was defined by the network's error on strings that have been learned plus the network's error on the first incorrect string. In our example, that means SSE is equal to the network's error on the first six strings plus the network's error on the seventh string, which is a^7b^7. The reason for doing so is to speed up the process of evaluation. As we analysed before, after the network made incorrect predication in a certain string, its performance on subsequent strings may be unreliable and thus in our simulations, we simply discarded those strings. As a result, we can save a lot of computation resource and the algorithm can run much faster, especially at the beginning of evolution because at that time the value of NCS is often relatively small. In the mean time, we also only calculated network's error when the current input is '*b*', keeping consistency with our previous simulations. The two context units were initialised to zero and were never reset. The activation function for all units was the standard sigmoid function. The population size was set to 400 and the crossover rate was set to 0.8.

In Tournament Selection, we randomly selected two individuals and compared their fitness. We first compared the value of NCS and if their values were equal, we would further compare their SSE to decide which one was the winner and then put the winner into a crossover pool. Next, we would perform crossover among those individuals in the crossover pool. Totally, we would select 160 pairs of individuals (i.e., 400*0.8*0.5=160). Each pair of individuals would produce two offspring through ATC and they would be put into a buffer. After 320 offspring were produced, we would add the best individual in the original population to the buffer and randomly selected some individuals from the crossover pool to fill in the buffer until it was full. At last, the buffer would replace the original population to be the new population.

The most important part is the evaluation function. Figure 14 is the flowchart of the process of fitness calculation of an individual. At the beginning, the two context units are set to zero. *Success* and *Error* represent NCS and SSE respectively and are also set to zero. A *flag* is used to indicate whether the network has made incorrect predication. After the network receives an input, the activations of the two hidden units will be calculated and the states of the two context units will also be updated. If the input is '*a*', the network will continue to receive the next input in the input stream. Otherwise, we will calculate the network's output and calculate the error between the expected output and the real output. This error is then added to *Error*. If the output (predication) is different from the actual next input, we set *Flag* to 1, indicating an incorrect predication. If the input is the last character of a certain string and *Flag*=0, which means the network has correctly predicated all predictable characters in that string, we will increment *Success* by 1 and the program will begin to evaluate the network on the next string unless the end of the whole training set is reached. If *Flag*=1, we will not change the value of Success and terminate the process of evaluation. In a word, the process of evaluation will be continued only if the network has correctly predicted all characters in the previous strings. After evaluation, the NCS of the individual will be set to *Success* and the SSE of the individual will be set to *Error*.

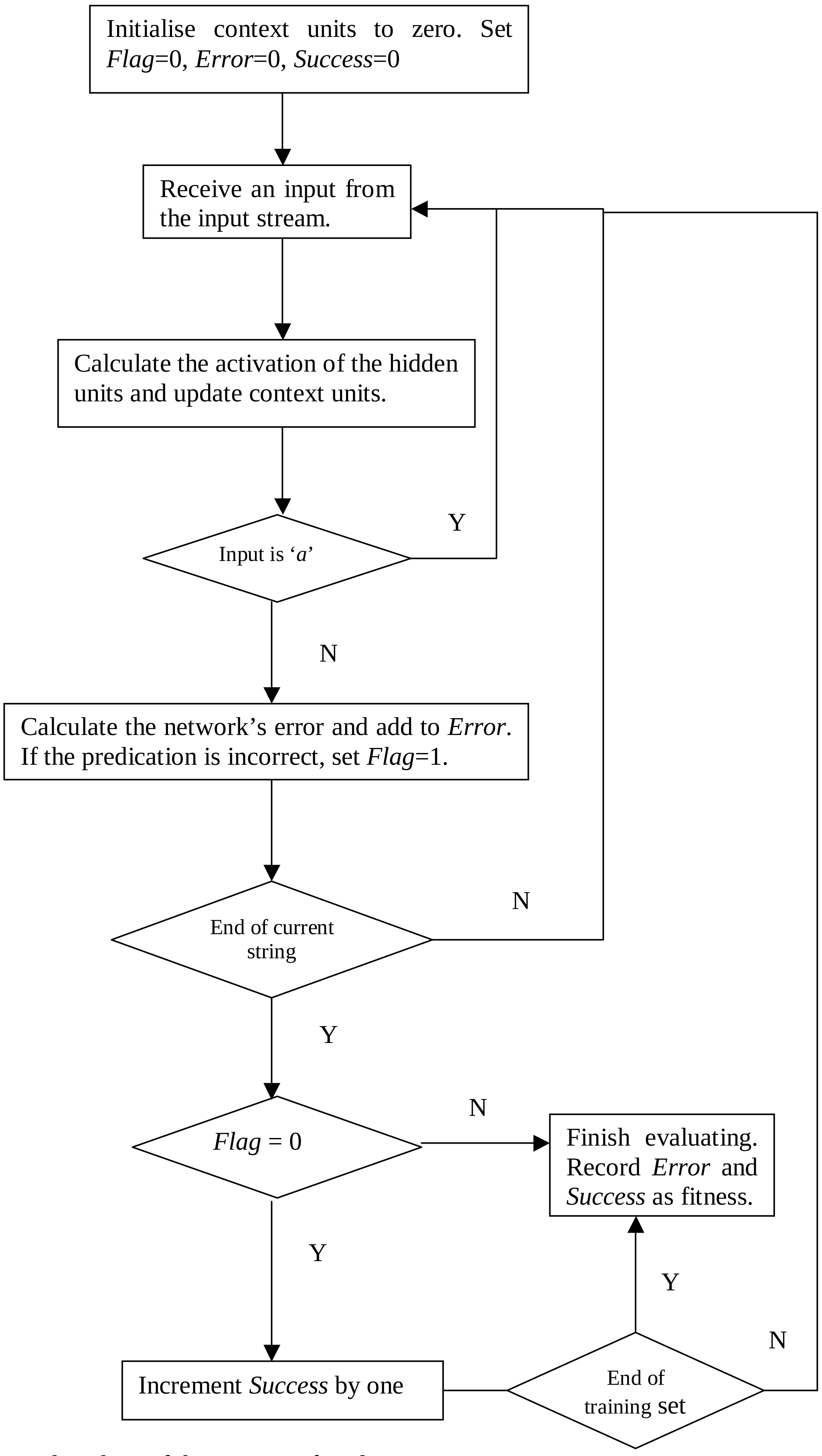
Initialise context units to zero. Set *Flag*=0, *Error*=0, *Success*=0
Receive an input from the input stream.
Calculate the activation of the hidden units and update context units.
Input is '*a*'
Y
N
Calculate the network's error and add to *Error*. If the predication is incorrect, set *Flag*=1.
End of current string
N
Y
Flag = 0
N
Finish evaluating. Record *Error* and *Success* as fitness.
Y
Increment *Success* by one
Y
End of training set
N

Figure 14: Flowchart of the process of evaluation.

3.3 Results and Analysis

We ran the simulations for 10 times. Each time, we allowed the program to run for 2000 generations and recorded the best solution at the end of evolution. Out of the 10 networks, 7 networks successfully predicted all the strings in the training set ($1 \leq n \leq 12$), 2 networks predicted strings with n up to 11 and one network predicted strings with n up to 10. At this stage, we can say that our algorithm can find a solution with high efficiency (i.e., 70%). Since the training set was small, the process of evolution was not as time-consuming as before. Usually, the evolution containing 2000 generations could be finished in less than 90 seconds in PIII 800 PC platform. In order to test the reliability of those solutions we have found, we generated a test set containing 100 strings with n from 1 to 12. Out of the 7 solutions, 5 also successfully predicted all the strings in this test set. That is, although we employed a small incremental training set, it is still quite likely to find a reliable solution. Furthermore, out of the 5 networks, 2 networks were also capable of processing string with n=13. The most interesting thing is that the network with best generalization ability is not from the 5 'reliable' networks but from the 2 'unreliable' networks. It can process strings with n up to19 but made incorrect predications on some strings in the test set.

Next, we will analyse two reliable networks: Network 1 and Network 2 (see Figure 15). Figure 16 shows Network 1's hidden (context) units trajectories with the decision boundary. The decision boundary is where the output of the first output unit is equal to that of the second output unit (i.e., in our simulations, if the output of the first output unit is greater than that of the second one, the prediction is '*b*' and if the output of the first output unit is less that that of the second one, the predication is '*a*'). We can see that this solution is similar to those found in previous simulations, always predicting '*b*' except given the last '*b*' as the input. After processing a string, the states of the two context units will return to a position very close their initial values (0, 0). By contrast, Network 2 demonstrated different behaviour. From Figure 17 we can see that it always predicts '*a*' given '*a*' except the first '*a*' as the input and predicts '*b*' given '*b*' except the last '*b*' as the input.

	Input Unit1	Input Unit2	Context Unit1	Context Unit2	Threshold
Hidden Unit1	1.471680	-1.716797	-11.467773	-3.289063	-1.005859
Hidden Unit2	-7.991211	4.281250	-7.828125	-8.557617	-7.121094

	Hidden Unit1	Hidden Unit2	Threshold
Output Unit1	15.151367	10.558594	1.484375
Output Unit2	-15.715820	-14.998047	-1.375000

(**Network 1**)

	Input Unit1	Input Unit2	Context Unit1	Context Unit2	Threshold
Hidden Unit1	-0.230469	-2.937500	-14.633789	-12.654297	-2.077148
Hidden Unit2	-0.198242	13.810547	-1.018555	-1.039063	14.777344

	Hidden Unit1	Hidden Unit2	Threshold
Output Unit1	15.991211	-2.788086	1.000000
Output Unit2	-15.997070	0.003906	-1.133789

(**Network 2**)

Figure 15: Parameters of the two reliable networks on the $a^n b^n$ language.

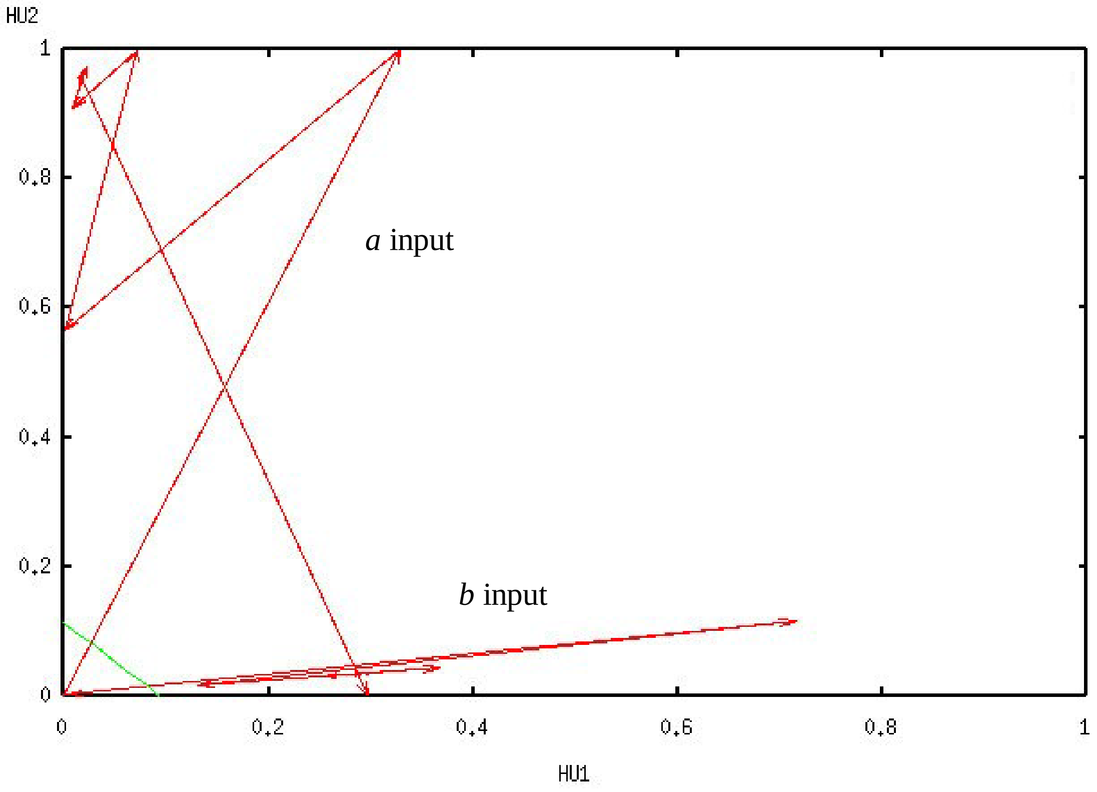

Figure 16: Hidden (context) units trajectories for n=12 and the decision boundary of the output (Network1). The short line at the left-bottom corner is the decision boundary. The left side indicates an ‘*a*’ predication and the right side indicates a ‘*b*’ predication.

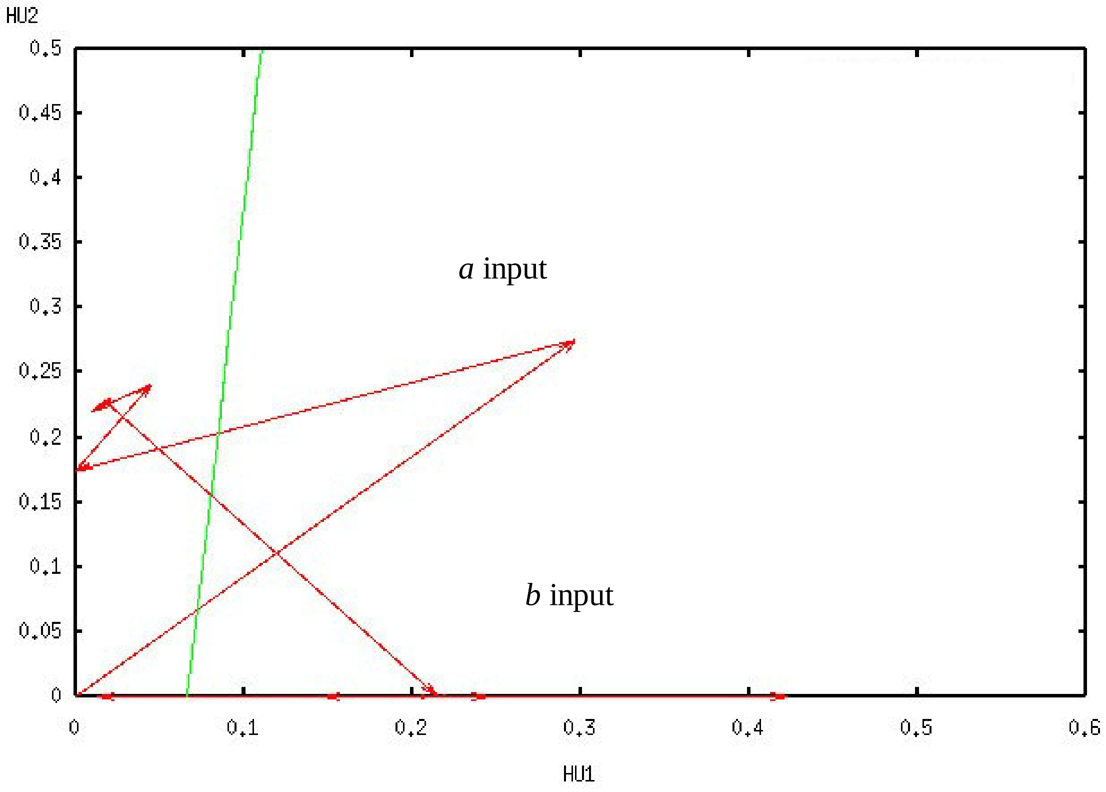

Figure 17: Hidden (context) units trajectories for n=12 and the decision boundary of the output (Network2). The nearly vertical line is the decision boundary. The left side indicates an ‘*a*’ predication and the right side indicates a ‘*b*’ predication.

IV. Conclusion and Further Work

The major motivation of this project was to investigate the possibility of training RNNs on language learning tasks by GAs instead of traditionally used BPTT, which tends to encounter instabilities during training. We began by investigating the question “Does the GA can be used to train a RNN?” From the simulation on the XOR task, we found that a GA is also capable of training a RNN and may have some kinds of advantages. Next, we conducted simulations on the a^nb^n language. We found that GAs can also be used in training RNNs on this task but the performance was not satisfactory. Based on the analysis of simulation results, we proposed a specifically designed GA with incremental training set and dual error measures. We can show that the simple context-free language a^nb^n can be learned by a SRN with this kind of GA as the training algorithm. The GA proposed in this paper demonstrated high efficiency in finding a solution. Furthermore, the solutions found also had good reliability and some of them were also capable of generalisation. At this point, we prefer not to thoroughly compare the performance of the GA with that of BPTT because designing a GA for a specific problem is still a work of art while BPTT is already a mature algorithm with solid and well-understood mathematical background. We simply want to show that GAs can also accomplish a training task that was previously finished by BPTT. The open question left here is how to further improve the GA’s performance so that the solutions can be found more quickly and have better generalisation ability. After all, the solution having best generalisation ability found so far is unfortunately not reliable on the test set, which means we still need to do some work to get a perfect combination of efficiency, reliability and generalization.

The work presented here is far from a complete story. There is an obvious extension for the $a^nb^nc^n$ language, which is a simple context-sensitive language. Previous research has revealed that a successful network should use some hidden units to count a^nb^n and some hidden units to count b^nc^n (Rodriguez, Wiles and Elman 1999). In order words, the $a^nb^nc^n$ language consists of both ‘counting up’ and ‘counting down’ letters and the solution is to oscillate in two dimensions, one for counting up and one for counting down (Bodén and Wiles 2000). In fact, we have conducted some trials on this context-sensitive language using the GA descried in this paper. We tried both SRN and SCN (Bodén and Wiles 2000) but still had no positive results. The training algorithm often got stuck at early stages with n=3 or 4. Obviously, more tuning work needs to be done for learning the $a^nb^nc^n$ language. One possible explanation of the difficulty we are facing is that the error landscape of the RNN on this kind of task is extremely complex and good solutions are often far from each other. In fact, from the simulations on the a^nb^n language we can see that the solutions found were often quite different. Darwen (2002) claimed that for this kind of problem in which best solutions do not have many common building blocks, crossover does not work well. Instead, the mutation-only approach may work better.

At last, it is necessary for us to pay some attention to one question: why did the solutions found by the GA for a^nb^n always employ only one active hidden unit? That is, in our solutions, counting of both *a*s and *b*s was performed by the same hidden unit with another one used to switch to another fixed point and change oscillation mode (i.e., from converging to diverging) when receiving the first ‘*b*’ input. Maybe there is some kind of inherent bias in the training algorithm.

Acknowledgement

Throughout the course of this project, my supervisor A/Prof Janet Wiles has helped guide me in the right direction and pushed towards reaching my desired goal. Without her helpful suggestions and comments, I would not have been able to achieve the results that can be seen in this thesis.

References

Elman, J.L. (1990) Finding structure in time. *Cognitive Science*, 14:179-211

Wiles, J. and Elman, J.L. (1995) Learning to count without a counter: a case study of dynamics and activation landscapes in recurrent neural networks. In *Proceedings of the 17th Annual Conference of the Cognitive Science Society* (Mahwah, NJ: Lawrence Erlbaum), pp.482-487.

Tonkes, B. & Wiles, J. (1997). Learning a context-free task with a recurrent neural network: An analysis of stability. In *Proceedings of the Fourth Biennial Conference of the Australasian Cognitive Science Society (OzCogSci97).*

Tonkes, B., Blair, A. and Wiles, J. (1998) Inductive bias in context-free language learning. *Proceedings of the Ninth Australian Conference on Neural Networks (ACNN'98)*, Brisbane, pp.52-56.

Rodriguez, P., Wiles, J. and Elman, J.L. (1999) A recurrent neural network that learns to count. *Connection Science,* 11(1): pp.5-40

Chalup, S. and Blair, A.D. (1999) Hill climbing in recurrent neural networks for learning the $a^n b^n c^n$ language. In *Proceedings of the 6th International Conference on Neural Information Processing*, Perth, pp. 508-513.

Bodén, M., Wiles, J., Tonkes, B. and Blair, A. (1999) Learning to predict a context-free language: analysis of dynamics in recurrent hidden units. In *Proceedings of the International Conference on Artificial Neural Networks,* Edinburgh, IEE, pp. 359-364.

Bodén, M. and Wiles, J. (2000) Contxt-free and context-sensitive dynamics in recurrent neural networks. *Connection Science*, Vol.12, No. ¾, 2000, pp. 197-210.

Yuan, B. (2002) Deterministic crowding, recombination and self-similarity. In *Proceedings of Congress on Evolutionary Computation (CEC2002)*, pp. 1516-1521, Honolulu, U.S.A.

Darwen, P.J. (2002) Search landscape of a realistic single-machine scheduling task: Peaks with big differences. In *Proceedings of Congress on Evolutionary Computation (CEC2002)*, pp. 1191-1196, Honolulu, U.S.A.