

# Notes on chapter 4 of Sieglemann's *Neural Networks and Analog Computation*.

---

Marshall Abrams © 2020

See [algrand/sieglemann4.clj](https://algrand/sieglemann4.clj).

---

## Lemma 4.1.2, stated on page 63

---

- The statement about the value of  $x_r$  is misleading. It suggests that there is a node value that will consist of a bunch of zeros, followed by the encoding string, followed by more zeros, but that's not what the algorithm in the proof provides. Rather, what you get is that node  $x_{16}$  contains zero during each step until the specified step (but see below about the count), and then in the specified step it contains the encoding string, and then after that it goes back to zero henceforth.

Note two off-by-one-ish errors in Lemma 4.1.2:

- The network needed in the proof has 17 nodes (numbered 0 through 16), not 16.
- The number of zero steps before the encoding appears is two less than what's stated. You can either change  $n$  to  $n - 1$ , or change 5 to 3. (This becomes clear when you run the algorithm. Yes, I'm taking into account that the step numbering is 1-based.)

So the actual number of ticks at which zero appears in  $x_{16}$  before the circuit encoding appears is

$$2(n - 1) + \sum_{i=1}^n l(C_i) + 5$$

or

$$2n + \sum_{i=1}^n l(C_i) + 3$$

where  $n$  is the number of 1's in  $u$ , and  $l(C_i)$  is the length of the  $i$ th encoding (not including the preceding or following 8). So the desired circuit encoding appears at step

$$2n + \sum_{i=1}^n l(C_i) + 4$$

## Circuit-retrieval algorithm

---

## Miscellaneous preliminary notes:

This is the core of the proof of Lemma 4.1.2.

I'll refer to the step in which  $u$  appears as tick 1 or step 1. So steps are 1-based, while matrix and vector indexes will be 0-based.

All elements of the state vector (of length 17) start as zero.

I formulate the state update algorithm using matrices and vectors, in terms of Equation (2.2), p. 19:

$$x_i(t+1) = \sigma \left( \sum_{j=1}^N a_{ij}x_j(t) + \sum_{j=1}^N b_{ij}u_j(t) + c \right)$$

where  $x_i(t)$  is the  $i$  th element of state vector  $x$  at time  $t$ . Or using matrices and vectors:

$$x(t+1) = \sigma(ax + bu + c)$$

where  $a$  is a 17x17 matrix,  $x$ ,  $b$ , and  $c$  are column vectors of length 17, and  $u$  is a column vector of length 1. The preceding equation is implemented by `next-state`.

$\hat{c}$  ("C" with a hat over it in the text), which encodes a series of Boolean circuits (pp. 61–62), is the col-9-to-row-10 weight.

## How the algorithm works:

1.  $u$  is added only on the first tick. Not before, not after. That is, you can have zeros as inputs before that point, but when  $u$  shows up, it does so only for one tick. It's replaced with zero after the first tick. Nothing happens until  $u$  shows up.
2. So  $x_9$  is nonzero only on the first tick. That causes  $\hat{c}$  to be placed into  $x_{10}$  on the second tick (since  $x_0$  through  $x_8$  are still zero). This is the *only* time that  $\hat{c}$  as a weight is used directly (since it's the only time that  $x_9$  is nonzero). After that, information from  $\hat{c}$  shuttles around in the  $x_0$  through  $x_8$  nodes, and  $x_{10}$ . (Pieces of  $\hat{c}$  get into  $x_{10}$  not because of the  $\hat{c}$  weight on  $x_9$ , but because  $x_{10}$  is a linear combination of  $x_0$  through  $x_8$ .)
3.  $x_{10}$  and  $x_0$  through  $x_8$  implement the lambda-tilde shift operator, but they do a bit more than that in the network. At each subsequent step,  $x_0$  through  $x_8$  will contain either:
  - a. A number that contains the left-shift of the decimal part of what's left of  $\hat{c}$ . This will be in some in  $x_j$  for  $j$  in  $\{0,2,4,6,8\}$ . Specifically, this shifted portion of  $\hat{c}$  will be in the  $x_j$  such that  $j$  is the integer part of the shifted string—i.e.  $j$  is what used to be the first digit. The new value in this  $x_j$  will be the next value of  $x_{10}$ .

b. 1 in  $x_i$  for  $i < j$  (see a above). There is always an even number of these 1's, so on the next tick, they will sum to zero in the linear combination that produces  $x_{10}$ .

c. 0 in  $x_i$  for  $i > j$ .

4.  $x_{11}$  **is used to build the possible output circuit encoding.** It reconstructs digits from c-hat using the 1's in  $x_0$ – $x_7$ , and places them on the front of  $x_{11}$ :

- 2 times the 4-part sum in  $x_{11}$  is the digit that was just stripped off in calculating  $x_j$  (see 3a above). This might not be obvious: the count of 1's below the newly shifted float (derived from c-hat) is equal to the digit that was shifted off. [You might think you could instead just sum all of the 1's. But you have to choose input nodes that will work every time, and then you'd accidentally add in the node with a shifted c-hat float in it. The trick to the algorithm here is based on the fact that the new piece of c-hat is always in an even-numbered node, so if you only use odd-numbered nodes, you're guaranteed to miss that node containing the shifted c-hat. Since there is always an even number of 1's below the shifted c-hat node, doubling the odd nodes gives you the same count as if you counted all of the 1's. (And the registers in  $x_0$ – $x_8$  above the c-hat float are zero.) So  $2(x_1 + x_3 + x_5 + x_7)$  gives you the digit that was just shifted off.]
- $x_{11}$  is used to build an output encoding string. We will place a new digit on the front of a right-shifted  $x_{12}$ . This has the old version of the partially constructed circuit encoding. We right-shift using  $(1/9)x_{12}$  in order to make space for the newly reconstructed digit that will be pushed onto this string. This new digit is  $2(x_1 + x_3 + x_5 + x_7)$  [see preceding item]. This is an integer, so we need to right-shift it to put it into the output string:  $1/9$  times  $2(x_1 + x_3 + x_5 + x_7)$  right-shifts the integer into the first decimal place, after which it can be added to  $(1/9)x_{12}$ .

(This is why the encoding in c-hat has to be backwards: you're pulling digits off the stack that came from c-hat, and then pushing them onto the stack in  $x_{12}$  and  $x_{11}$ , so the order gets reversed.)

5. **Counting to find the desired circuit that's indicated by the number of 1's in  $u$ :**

- We mentioned above that  $u$  is placed in  $x_9$  to start the process, copying c-hat to  $x_{10}$ .  $u$  is also placed in  $x_{13}$  on tick 1. (At that point  $x_{14}$  and  $x_{15}$  are zero, so contribute nothing.) Again,  $u$  as a value coming from the input stream will never appear again after this tick.
- This value is then copied to  $x_{14}$  or  $x_{15}$ , but not both:
  - If  $x_7 = 0$  (i.e. the last digit shifted off the c-hat string is not 8),  $x_{13}$  is simply copied to  $x_{15}$ , which is then copied back to  $x_{13}$ , and so on. (Like vamping in music.) Meanwhile,  $x_{14}$  stays zero, since 2 is greater than a left-shifted  $u$ -string.
  - But when  $x_7$  is 1 (i.e. when the last digit shifted off of the c-hat string is 8),  $x_{15}$  becomes zero (since  $u$  or any part of it is always  $< 1$ ), and now  $x_{14}$  is in effect  $\sigma(2x_{13} - 1)$ . There the 1 simply strips off the 1 that is newly shifted into the integer position by  $2\times$ . (Remember,  $u$  consists of a series of 1's indicating how many circuits to count.)
  - The resulting value will then be copied back to  $x_{13}$ . This is how we increment the counter by stripping a digit from the  $u$  string when an 8 is encountered in c-hat. [Note that at the beginning, c-hat is copied to  $x_{10}$  in tick 2, is stripped in  $x_8$  in tick 3, so it's in tick 4 that the first 1 is stripped from  $u$ , placing the new version  $x_{14}$ .]
- The preceding assumed that some 1's remain from  $u$ . When all that's left are zeros,  $x_{14}$  and  $x_{15}$ , and hence  $x_{13}$ , will be forced to zero.

**6. The relationship between counting through the 1's in  $u$ , and output encoding string construction:**

Although the  $x_{11}$  code always tries to go through the process of constructing an encoding string, shuttling partially constructed versions back and forth between  $x_{11}$  and  $x_{12}$ , *this doesn't happen while there are still 1's left from  $u$ .*

That's because  $2x_{13}$  is subtracted in the  $x_{11}$  formula. As long as there is at least a single 1 left from  $u$ , in  $x_{13}$ ,  $2x_{13}$  left-shifts that 1 into the integer place. Since the partially constructed output would always be  $< 1$  (because of the  $1/9$ 's),  $x_{11}$  remains zero until all of the 1's are gone.

Note that it's the 8 *before* the encoding string that causes a 1 to be shifted. So if  $u = 0.111$ , the final 1 will be shifted off just as the network is starting to look at the third encoding. Since  $x_{13}$  will remain zero now, the encoding can now be copied, piece by piece and in reverse, into  $x_{11}$  and  $x_{12}$ .

**7. Putting the output on  $x_{16}$ :**

Finally, all along, if there was a partially constructed output in  $x_{12}$ , it was copied to the formula for  $x_{16}$ . However, since that formula contains  $-1$ , the output string, which is always  $< 1$ , is turned into zero by *sigma*. It's only when, now that a nonzero output string is in  $x_{12}$ , *and then*  $x_7$  contains 1 because another 8 was found in the  $\hat{c}$  remains, that the  $-1$  in the  $x_{16}$  formula is counteracted, and the now fully reconstructed encoding string will appear in  $x_{16}$ .

**8. Coda:**

After the answer has been placed in  $x_{16}$  at step  $2n + \sum_{i=1}^n l(C_i) + 4$ ,  $x_{16}$  goes back to zero because now  $x_7$  is not 1, since the most recently removed digit is not 8.

However, *the process continues*. As written, the algorithm doesn't stop. And since there are no more 1's left from  $u$ , new circuit encodings will periodically appear in  $x_{16}$ . Or rather, what happens is that new circuit encodings are concatenated onto the ones already seen so far, separated by 8's. It's pretty useless. (If you run the algorithm, you'll eventually get to states that are all zeros, but that's apparently just because you only included a finite number of circuits in  $\hat{c}$ , and they've run out.)

Of course, it would be easy to stop the process after the first encoding is reported. This isn't a flaw in the proof.