Advanced Topics in Machine Learning: Meta-Learning

Rahul Siripurapu

1 Introduction

Sepp Hochreiter et al, in their 2001 paper "Learning to learn using gradient descent" conduct an experiment where they successfully train an LSTM network to learn how to learn. They demonstrate this by feeding the network sequences of input-output pairs obtained from a randomly sampled function from a distribution of functions and show that the LSTM quickly adjusts its output to match the function targets much faster than could be expected from gradient descent. In the following pages, a process for replicating these results is detailed, along with an analysis of the mechanism by which this "Faster than gradient descent learning" happens.

2 Motivation

The benefits of a Meta-learning algorithm are quite obvious. What is not obvious, are the constraints under which it is possible. In "Fixed weight networks can learn" the authors show how learning can happen in a fixed weight recurrent network. A fixed weight recurrent network can in theory, simulate another neural network "a subordinate network" which learns the input-output mappings. The parameters of this subordinate network would be encoded in the cell states of the original network. Intuitively, one may expect that such a recurrent network which is trained by gradient descent could learn how to better train the subordinate learning algorithm. This is precisely what Hochreiter et al show to a limited extent. Further it is considered that LSTMs are turing complete and hence should be able to do something akin to simulating an LSTM within an LSTM and so on albeit at the cost of significantly more recurrent steps or network complexity. This justifies the expectation that, given a proper training setup, the LSTM could be trained to learn how to learn by simulating a subordinate network within itself and figuring out how to update it. It would be an interesting experiment to figure out how to train an LSTM to simulate a subordinate LSTM and learn how to update it. But this is currently beyond the scope of this work.

3 Experimental Setup

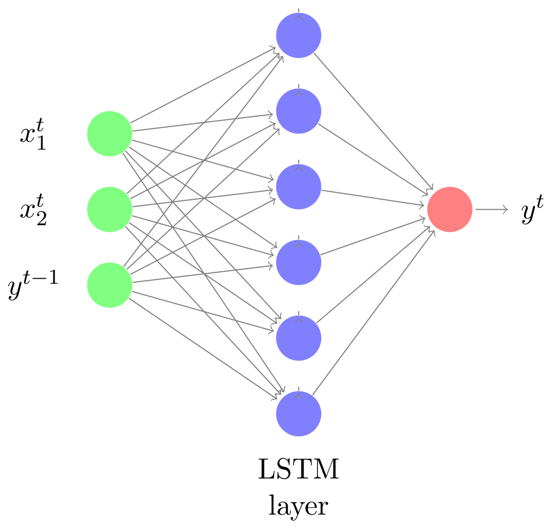
The most crucial element that determines what the network actually learns to do is the training setup. In Meta-learning the training setup consists of a distribution of tasks rather than a single task. Indeed, to learn how to learn better it would be important to be able to measure your performance on a set of different tasks. Once the network is able to learn new tasks quickly, we can expect that it would perform similarly on an unseen test set of tasks. Whether it can show similar performance on a test set drawn from a different distribution than the training task distribution would depend on the generality of the subordinate learning algorithm. This begs the question, why would a network shown a limited distribution of tasks learn a fully general subordinate learning algorithm rather than cheat by fitting only the training distribution? Indeed, the training curriculum is the important factor.

In the original paper, the following distributions of functions are considered:

Table 1: Training distributions

Function	Definition
Boolean-14	Set of binary boolean functions excluding XOR,XNOR
Boolean-16	Set of all 16 binary boolean functions
Semi-Linear	$0.5\,(1.0 + \tanh\,(w_1x_1 + w_2x_2 + w_3))$
	$w_1, w_2 \in [-1, 1]$
Quadratic	$ax_1^2 + bx_2^2 + cx_1x_2 + dx_1 + ex_2 + f;$
	a , . . . , f $\in [-1, 1]$; $output(scaled) \in [0.2, 0.8]$

The following network architecture succeeds for the first three classes:



The architecture used by Hochreiter et al. uses an earlier version of LSTMs that did not have a forget gate. In the following experiments however, the forget gate is used since the LSTM learns better with it.

As seen in the figure, the inputs consist of the current x_1 and x_2 along with the output from the previous time step y^{t-1}. First a random function is picked from the chosen training function distribution. A sequence of random inputs are generated along with the corresponding outputs of the function. This is then fed to the network for training. The output is used as a target to train the network using Mean Squared Error as done in the original paper. The 1 step delayed outputs are also provided as an input to the network because we want it to use the correct output to train its subordinate learning algorithm. As pointed out in the original paper, we cannot feed the current output itself since it is difficult to ensure that the network doesn't cheat by mapping it directly to the output.
Further recent training methods like Adam and mini-batch training have been used to speed up the learning process significantly.

4 Boolean functions

The network easily learns both the B-14 and B-16 functions successfully. It produces the characteristic learning curves seen in the original paper:

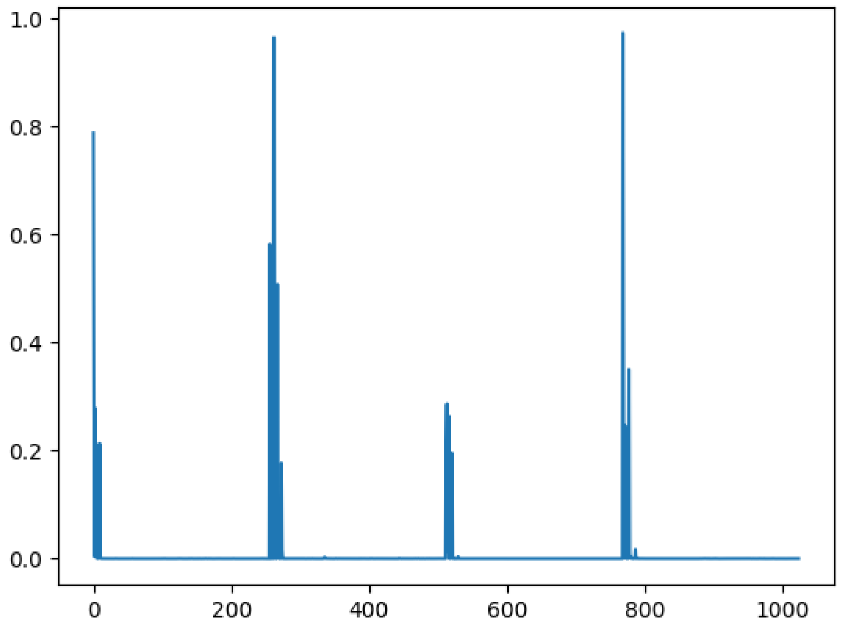


Figure 1: Training loss curve for the B16 experiment with mini-batch size 4

The figure above shows the training loss curve for one batch of 4 sequences using for training the B16 network. The peaks represent the points where the function changes.

Within 2-3 steps the network learns the new function and the loss goes down.

The paper does not detail the dataset partitions and whether even a separate test set was used which is very critical. It seems unlikely that the test set functions used were distinct from the training. Since it is important to have a distinct test set to ensure that "learning" which is distinct from "memorizing" happens, for the B14 experiment, the training was done only on the first 11 functions using the same sequence length of 64 but with a batch size of 16, with the Adam Optimizer. Following is the training loss curve and it attains a loss of 0.034 in terms of Mean Squared Error as in the original paper(0.033):

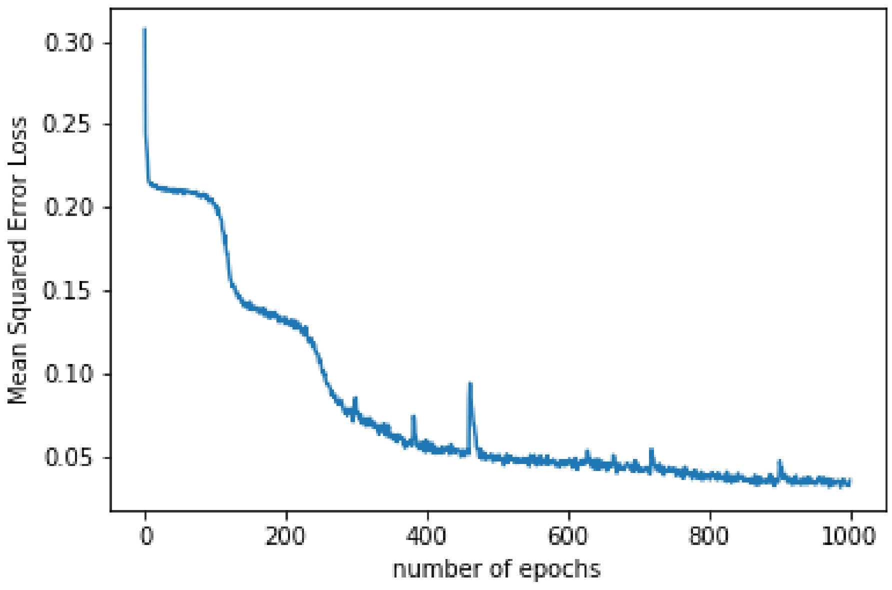


Figure 2: Training loss vs Epochs for the B14 experiment with mini-batch size 16

The network achieves an accuracy of 96% which exactly translates to getting 3-4 answers wrong in the 64-long training sequence per function. However this scheme does not help it generalize enough to perform equally well on the 3 held out functions, where it only achieves an accuracy of 86% and a significantly higher loss of 0.09. The same training scheme worked well for the B16 experiment, where 13 functions were used for training and longer sequences of 256, producing the following training loss curve:

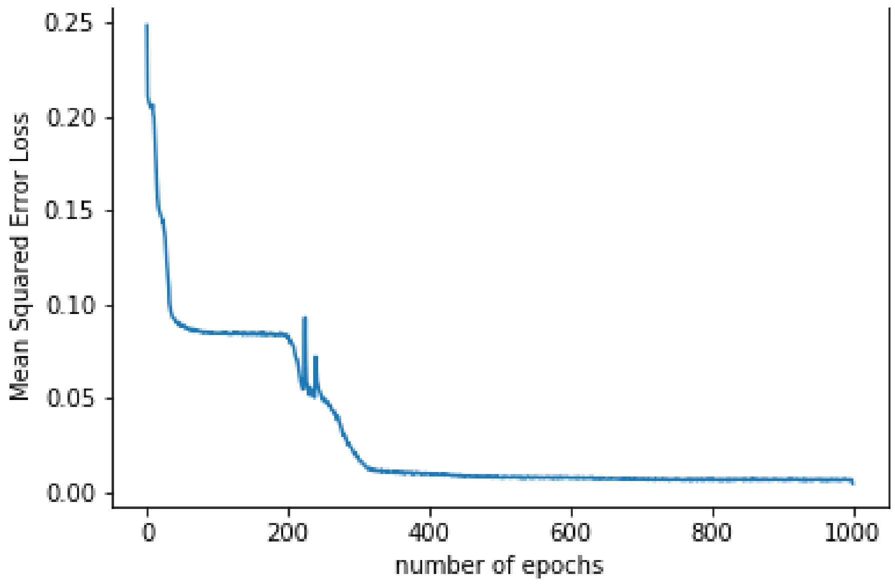


Figure 3: Training loss vs Epochs for the B16 experiment with mini-batch size 4

This time however it learns something very general. It achieves a loss of 0.0064 compared to the loss of 0.0055 in the paper and an accuracy of 99.3% on both the training as well as the test set! This corresponds to getting 2 answers wrong on average per sequence. Following is the loss curve on the test set:

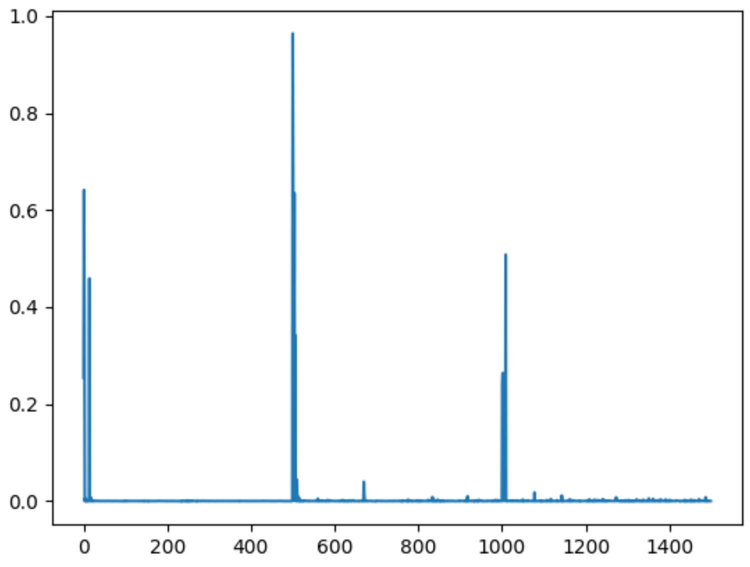


Figure 4: Test loss curve for the B16 experiment on the three test functions

So the network has certainly learned "how to learn" since it is able to quickly learn three new functions that it has never seen before. The above graph provides significant insight into what the network is doing!

5 Understanding the B16 network through surgery

Ideally never absorb information without predicting it first. Then you can update both 1) your knowledge but also 2) your generative model. **-Andrej Karpathy**

The fact that it takes just 2-3 examples to learn suggests that it may have learned one of the most fundamental skills a student learns while learning to learn: Memorization! But how can the network learn to memorize something that it has not seen yet? The correct output is provided in the subsequent time step and the gradients only flow backwards and cannot reach the input weights for information provided in the future! The solution here is to carry the problem into the future where the answer is available! It would still need to look a few more steps into the future for the gradient based reasoning to figure out what relation the correctly matched input output pairs have to do with the loss it obtains in the future

It is not detailed in the original paper how the figure six was arrived at for selecting the network size but a network of 6 LSTM units can implement a simple memorization strategy. 2 units to remember the past input, and 4 units to store the correct outputs corresponding to the 4 possible input combinations. The 2 units always forget their current states and learn the current input which is made available in the next time step. This input from the past is used to determine which of the 4 neurons is responsible for memorizing this input's correct output which is now available as an input. The selected neuron forgets its state and stores the correct answer provided. With this strategy the network should be able to immediately learn any new function.

To verify that this is indeed what the network is doing, we will need to perform a surgery. The simplest verifiable fact should be that two of the units are never used to output because their function is only to memorize the input.
The following are the weights and the bias rounded to 2 decimals from the output layer:

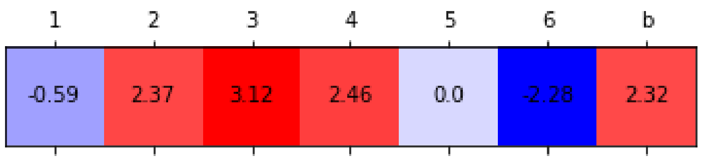
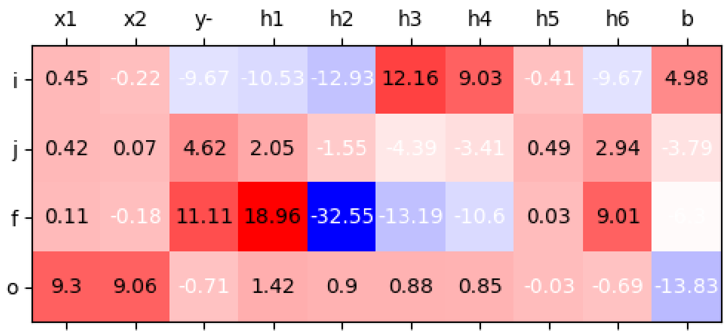


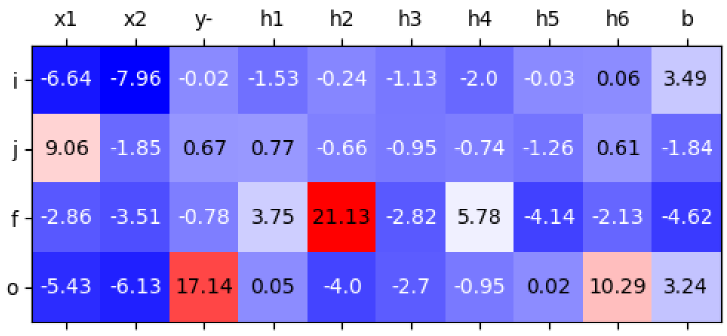
Figure 5: Output layer weights visualized

Exactly as expected the first and the fifth neuron have weights equal or close to zero (when compared with the bias). Both of them are individually incapable of overcoming the bias (the outputs from the network are limited to [-1,1] because we have a tanh activation). This verifies our first hunch. We can try to verify other similar patterns in the weight matrices. One such pattern is clear by looking at the first two weights of the output gates for the following four neurons. The rows correspond to the four LSTM gates and the columns correspond to the three inputs x1,x2 and y(t-1) and the 6 recurrent inputs along with the bias at the end.



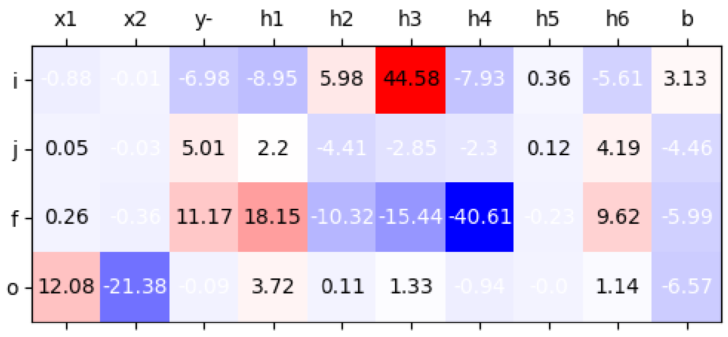
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	0.45	-0.22	-9.67	-10.53	-12.93	12.16	9.03	-0.41	-9.67	4.98
j	0.42	0.07	4.62	2.05	-1.55	-4.39	-3.41	0.49	2.94	-3.79
f	0.11	-0.18	11.11	18.96	-32.55	-13.19	-10.6	0.03	9.01	-6.3
o	9.3	9.06	-0.71	1.42	0.9	0.88	0.85	-0.03	-0.69	-13.83

Figure 6: Weights of Neuron 2



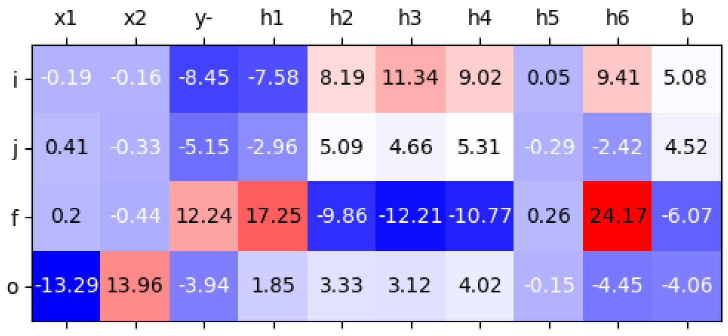
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-6.64	-7.96	-0.02	-1.53	-0.24	-1.13	-2.0	-0.03	0.06	3.49
j	9.06	-1.85	0.67	0.77	-0.66	-0.95	-0.74	-1.26	0.61	-1.84
f	-2.86	-3.51	-0.78	3.75	21.13	-2.82	5.78	-4.14	-2.13	-4.62
o	-5.43	-6.13	17.14	0.05	-4.0	-2.7	-0.95	0.02	10.29	3.24

Figure 7: Weights of Neuron 3



	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-0.88	-0.01	-6.98	-8.95	5.98	44.58	-7.93	0.36	-5.61	3.13
j	0.05	-0.03	5.01	2.2	-4.41	-2.85	-2.3	0.12	4.19	-4.46
f	0.26	-0.36	11.17	18.15	-10.32	-15.44	-40.61	-0.23	9.62	-5.99
o	12.08	-21.38	-0.09	3.72	0.11	1.33	-0.94	-0.0	1.14	-6.57

Figure 8: Weights of Neuron 4



	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-0.19	-0.16	-8.45	-7.58	8.19	11.34	9.02	0.05	9.41	5.08
j	0.41	-0.33	-5.15	-2.96	5.09	4.66	5.31	-0.29	-2.42	4.52
f	0.2	-0.44	12.24	17.25	-9.86	-12.21	-10.77	0.26	24.17	-6.07
o	-13.29	13.96	-3.94	1.85	3.33	3.12	4.02	-0.15	-4.45	-4.06

Figure 9: Weights of Neuron 6

After analyzing more such heat maps of the weight matrices and the activations, the following is an outline of the algorithm used. The cell state of neuron 2 remembers the value of the output for the input [1,1]. Neuron 2 stores -1 when the output should be 0 and 0 when the output should be 1. Neuron 4 remembers the output for the input [1,0] and again stores -1 for 0 and 0 for 1. Neuron 6 tracks the output for [0,1] and this one however, as we can see from the output weight matrix, has a negative value, and thus stores 1 for 0 and 0 for 1. As seen from the bias, the output is always 1 and activation of any of these neurons helps overcome the bias to pull the network output down to zero. But that is without counting the final combination [0,0]. Neuron 5 tracks the output for [0,0], stores -1 for 1 and 1 for 0 but however it's output never makes it directly out, rather Neuron 3 recognizes the input [0,0] and based on the stored value in Neuron 5 outputs -1 for 0 and 0 for 1. This now perfectly explains the output weights. The mechanism was not as perfect as the initial guess but close enough. The complete set of heatmaps is attached in the Appendix.

6 Semi-linear functions

In this experiment, the network must fit a significantly more complicated semi-linear function which looks like this:

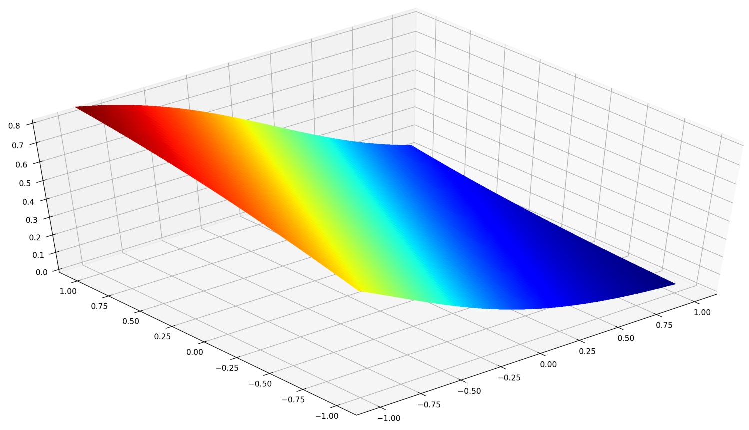


Figure 10: Semilinear function

The same network works successfully even on this task. The following is the network loss for the 2000th epoch:

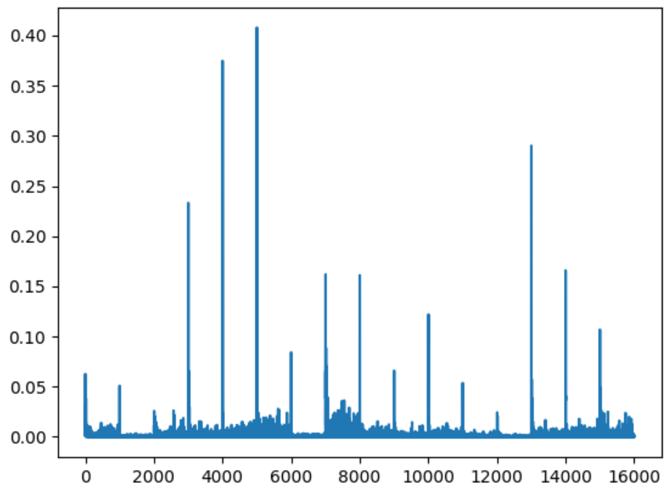


Figure 11: Semilinear loss curve for batch size 16 and sequence length 1000

Following is the training curve:

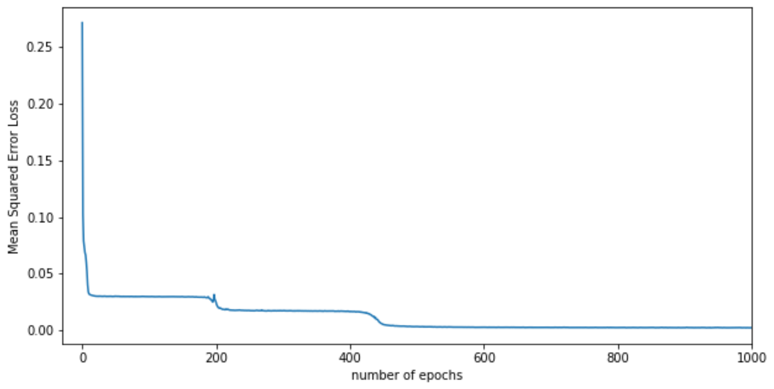


Figure 12: Semilinear training loss vs epochs

This time however the convergence is very quick using a batch size of 16 and sequence length of 1000 and it converges to 0.0017 within 1000 epochs itself, whereas in the paper it is trained till 5000 epochs to get 0.0020.

It is not obvious that even this can be solved using a method similar to the previous one, but it turns out it can. The function used here can be seen as a continuous analogue to the boolean function. A similar analysis can be done for this network as well but has been skipped due to constraints in time.

7 Quadratic Functions

The architecture used by the authors is not clear for this case. The paper states they use a network with 6 LSTM units as above but with 12 memory cells followed by a standard hidden layer of 40 neurons whose output is fed back as input to the first layer. They state that this network has 5373 weights. A similar architecture using 6 LSTM units in the first layer and 40 LSTM units in the second layer achieves an average Loss of 0.0005 within just 3000 epochs compared to the 25000 used in the paper. Further analysis is needed to understand the significance of these results.

8 Conclusion

Following are the conclusions that can be drawn based on the above experiments and a brief survey of the literature in this field.

What is clear from these experiments is that this network does learn how to learn, but only in a limited sense. The B16 network successfully learns how to memorize answers which is perhaps the best strategy for that scenario. However the evidence is not sufficient to say that the network simulates another network in its cell states and we cannot easily see the network's quick adaptation as coming from updates being made to the parameters of a subordinate network. One analogy could be to an agent that runs Newton's method for finding roots. Whenever the function changes it will converge to the new roots quickly, but one would be hard pressed to call it a "learning algorithm". In that same sense "memorization" can be seen as a learning algorithm.

The reason why the learned algorithm performs better than gradient descent is simply because gradient descent executes a circuit search for the best circuit that can solve any general problem whereas these algorithms which are learned are very specific to the task distributions they were trained on. For example, the priors that help the Boolean-16 network achieve optimal learning are the facts that there are only 4 different input combinations and each has a 1 or 0 output. By observing the weight matrices we can see that the network is specifically exploiting these two facts.

It is clear that gradient descent can be used to train an LSTM to implement complex algorithms and it is plausible that similar networks can be trained to learn specific algorithms for any function or even a distribution of functions but at the cost of significantly increasing the complexity of the network and the limitations would merely be the limitations of gradient descent itself as an optimizer. As seen in the case of the Boolean 16 function set, one may speculate that as long as the network is more complex than the task at hand, there are many weight combinations that may achieve the same result making it much easier for gradient descent to converge to some good minimum. But if the task is so complex that out of all the possible weight combinations, only a very tiny fraction of weights is able to perform the task reasonably well, then gradient descent would not work.

9 References

1. Learning to learn using gradient descent, Hochreiter et al. 2001
2. Meta-learning with backpropagation, Younger et al. 2001
3. Long Short Term Memory, Hochreiter and Schmidhuber, 1997
4. Learning to learn by gradient descent by gradient descent, Andrychowicz et al. 2016
5. Fixed-weight networks can learn, Cotter et al. 1990
6. Fixed-weight online learning, Younger et al. 1999

10 Appendix

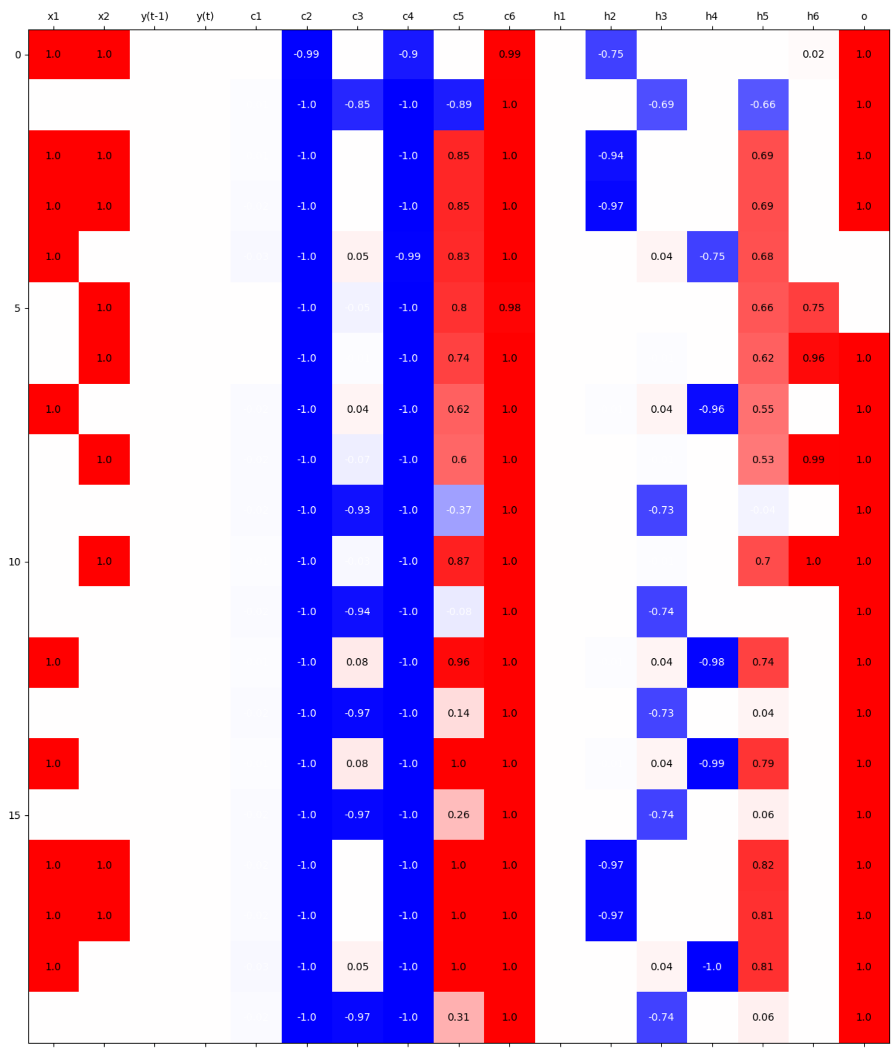


Figure 13: Boolean function 0

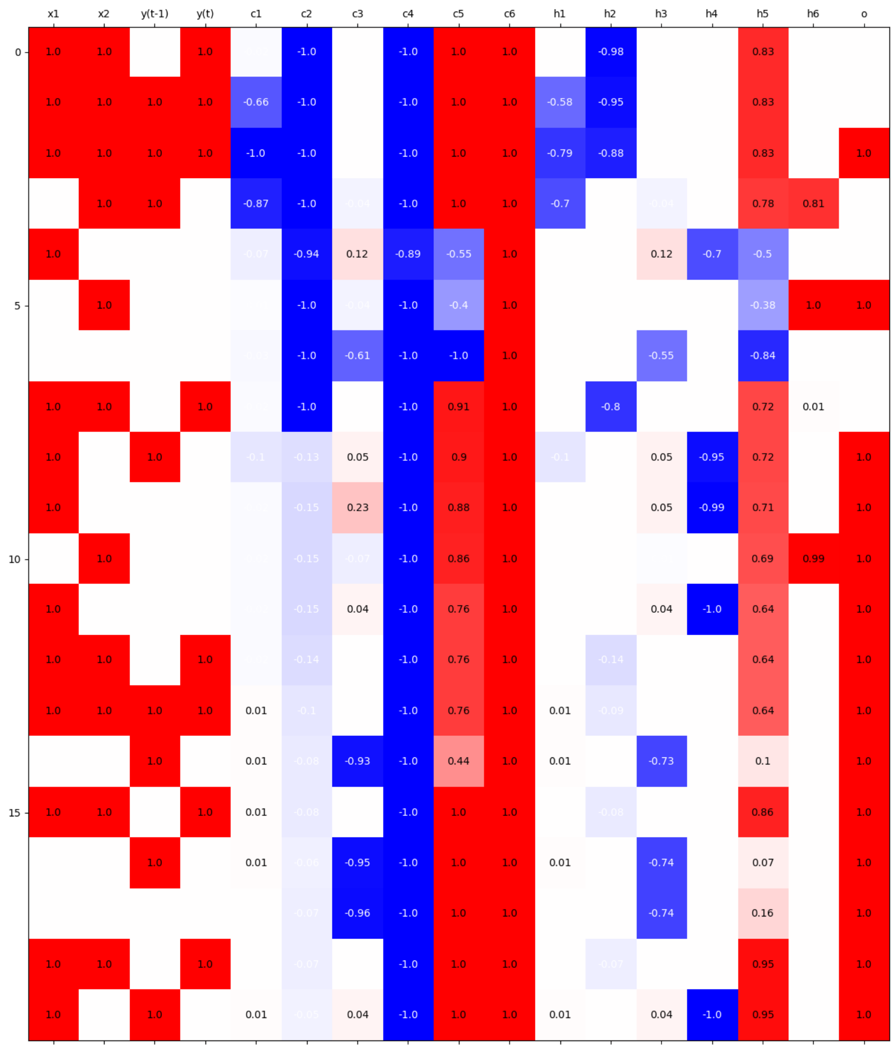


Figure 14: Boolean function 1

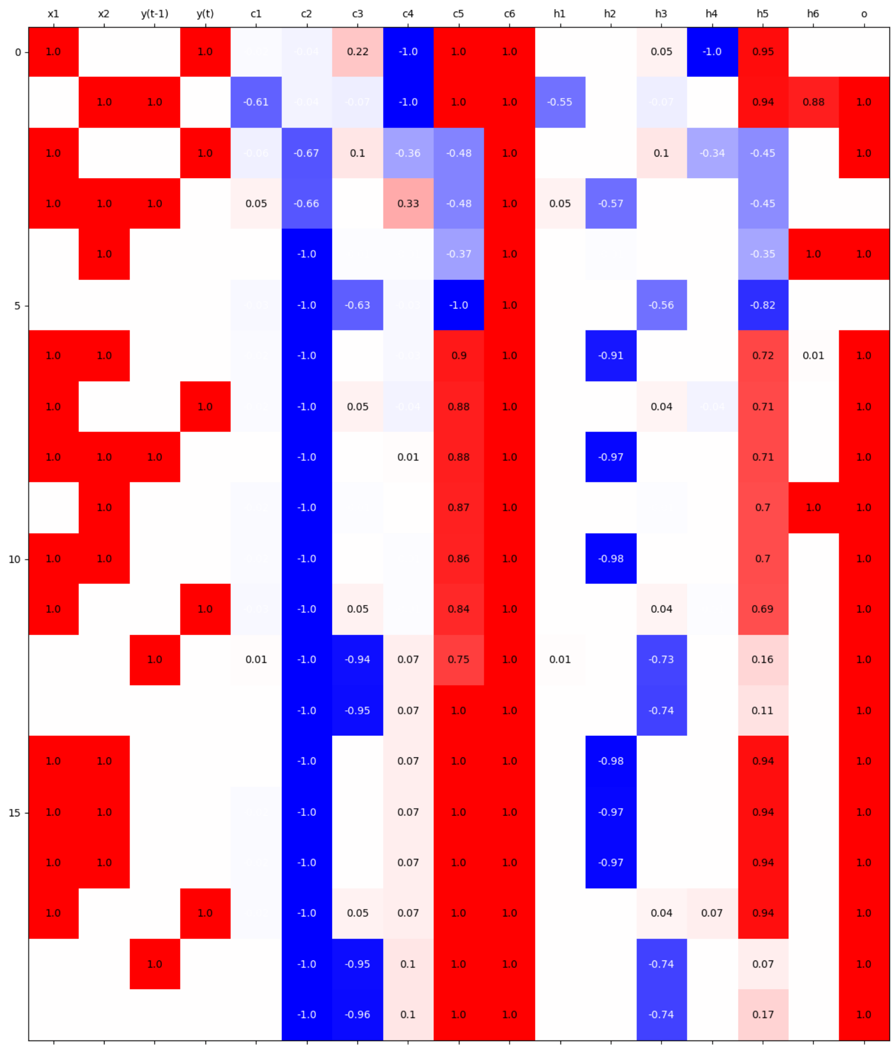


Figure 15: Boolean function 2

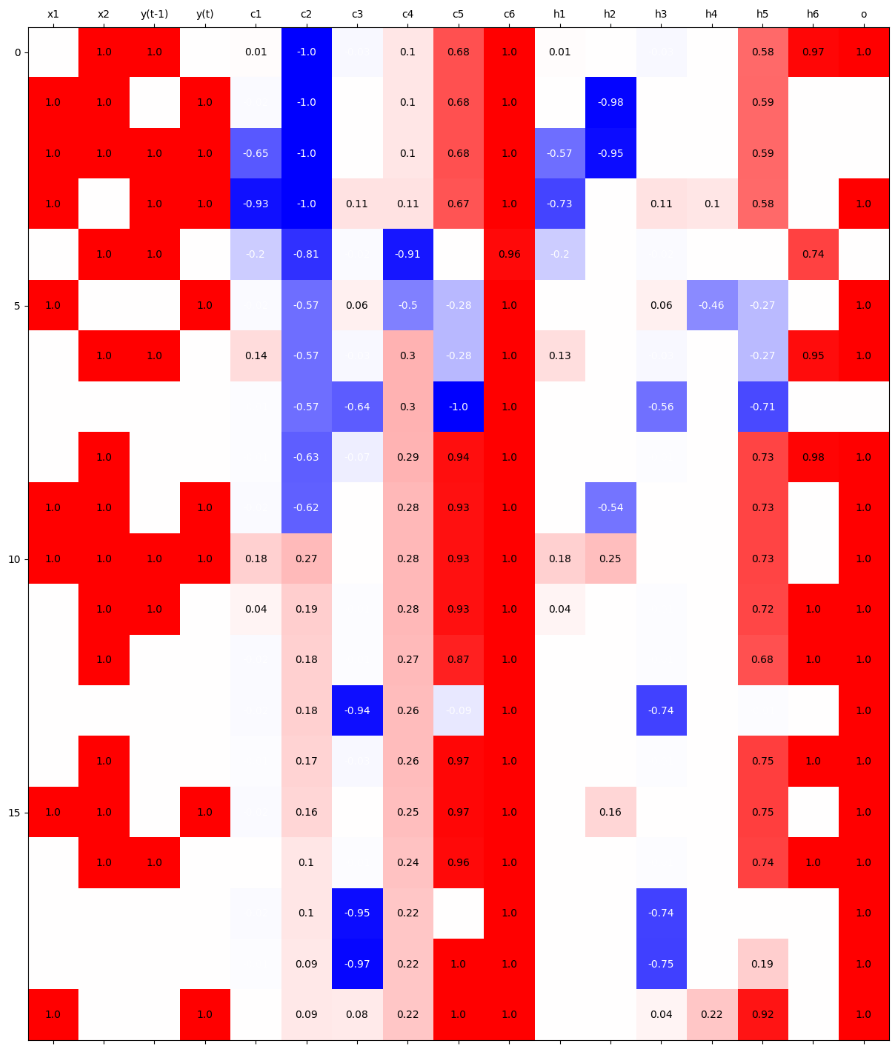


Figure 16: Boolean function 3

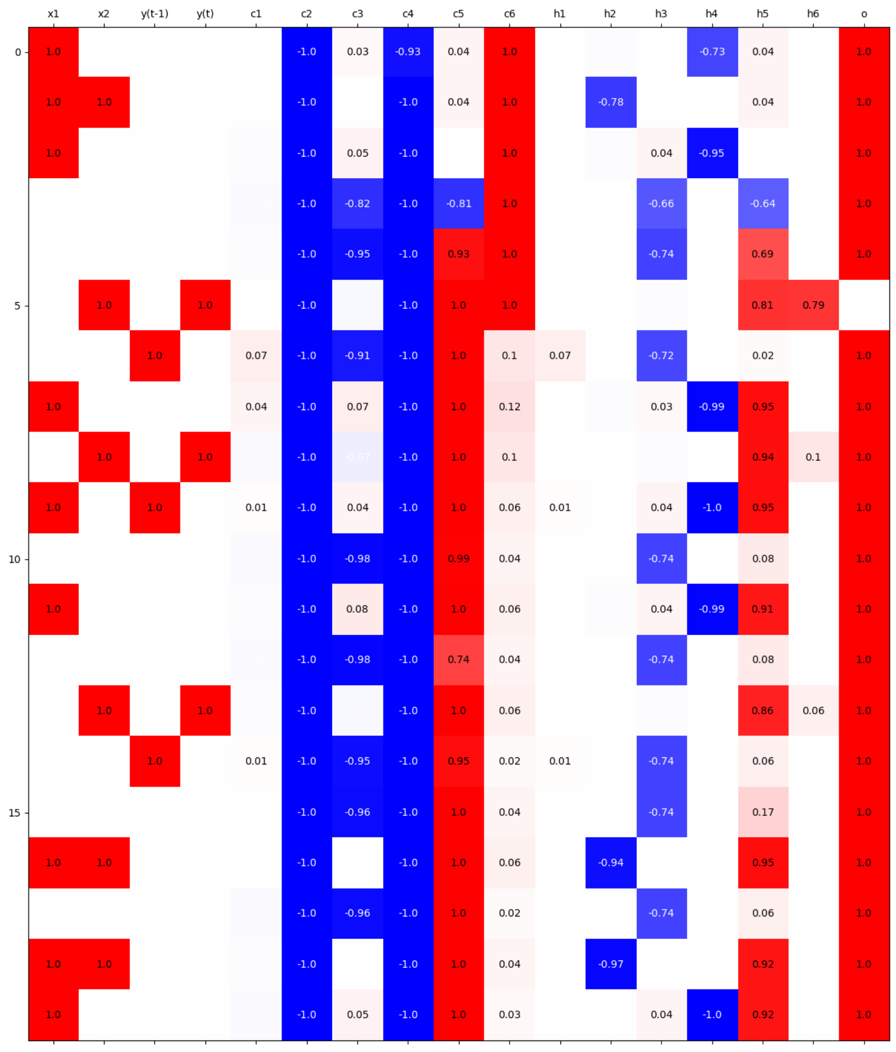


Figure 17: Boolean function 4

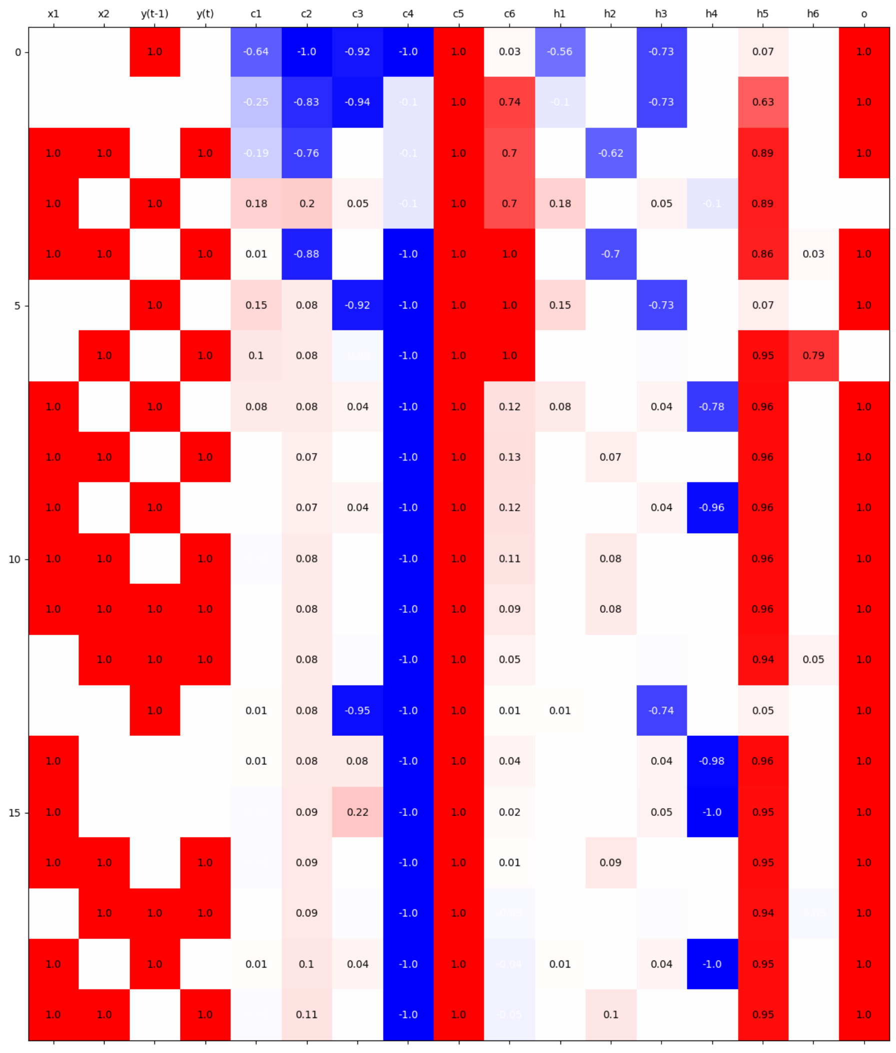


Figure 18: Boolean function 5

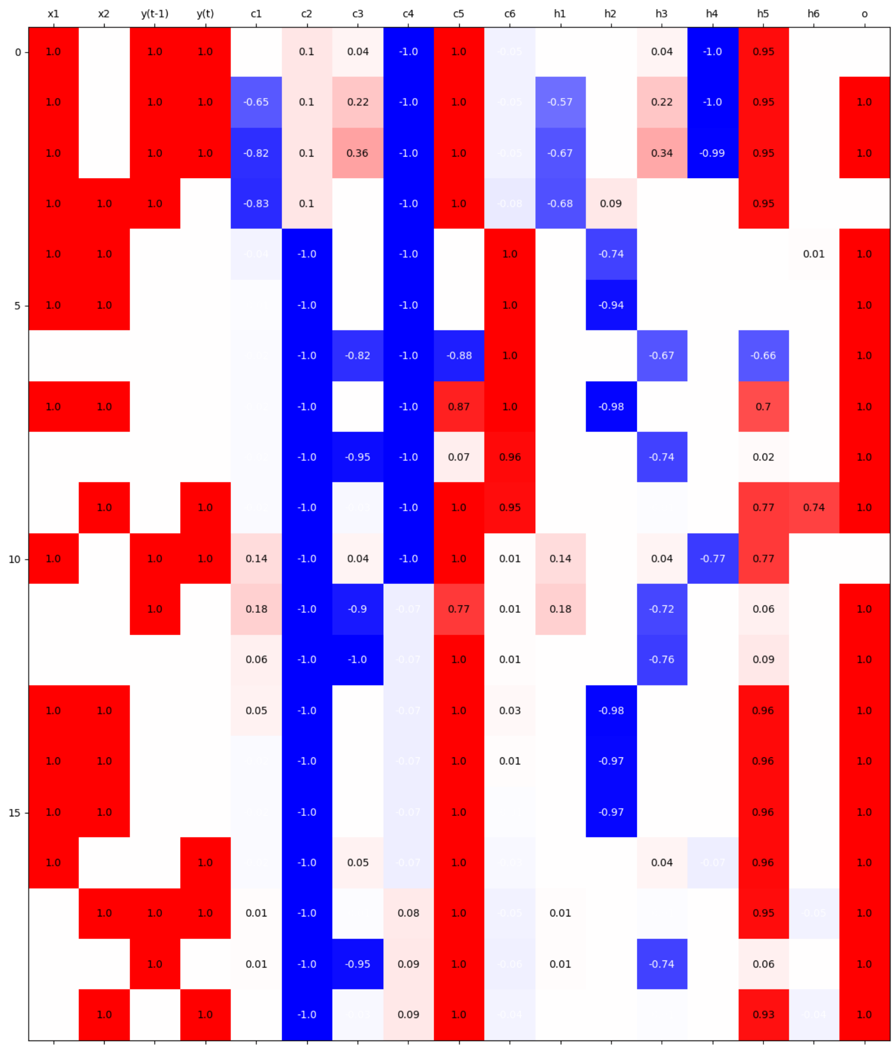


Figure 19: Boolean function 6

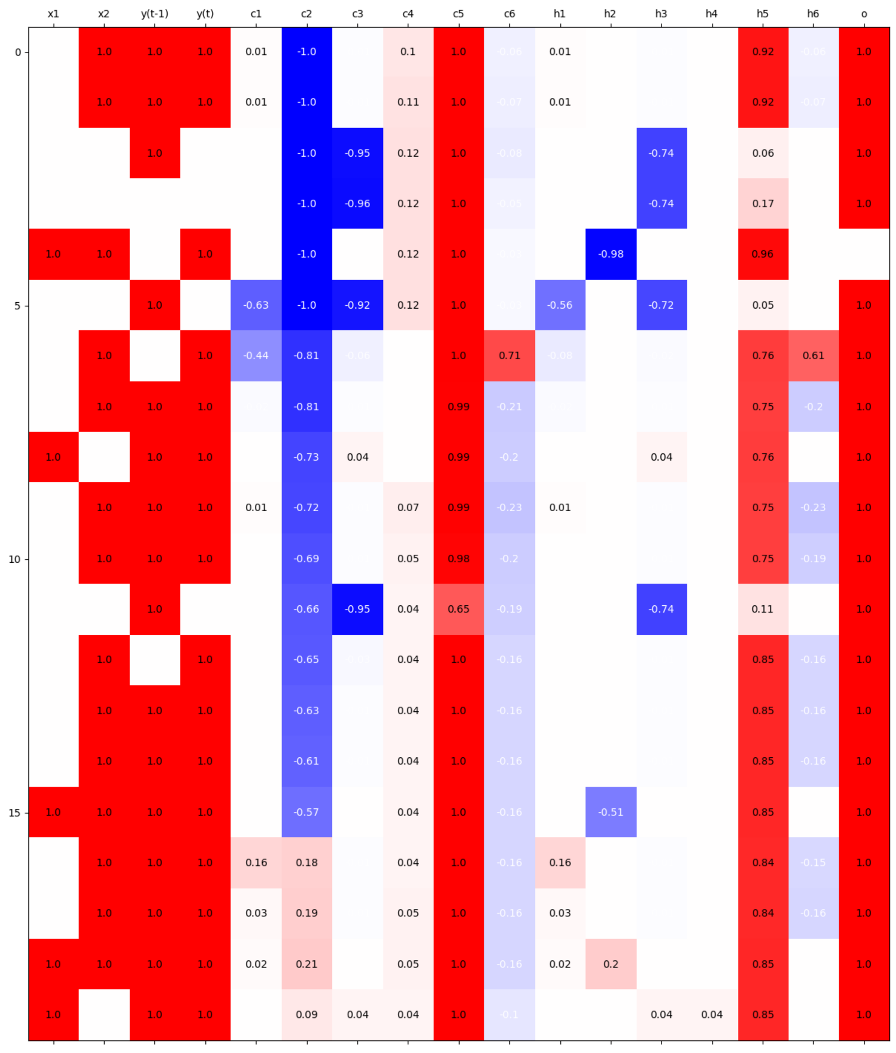


Figure 20: Boolean function 7

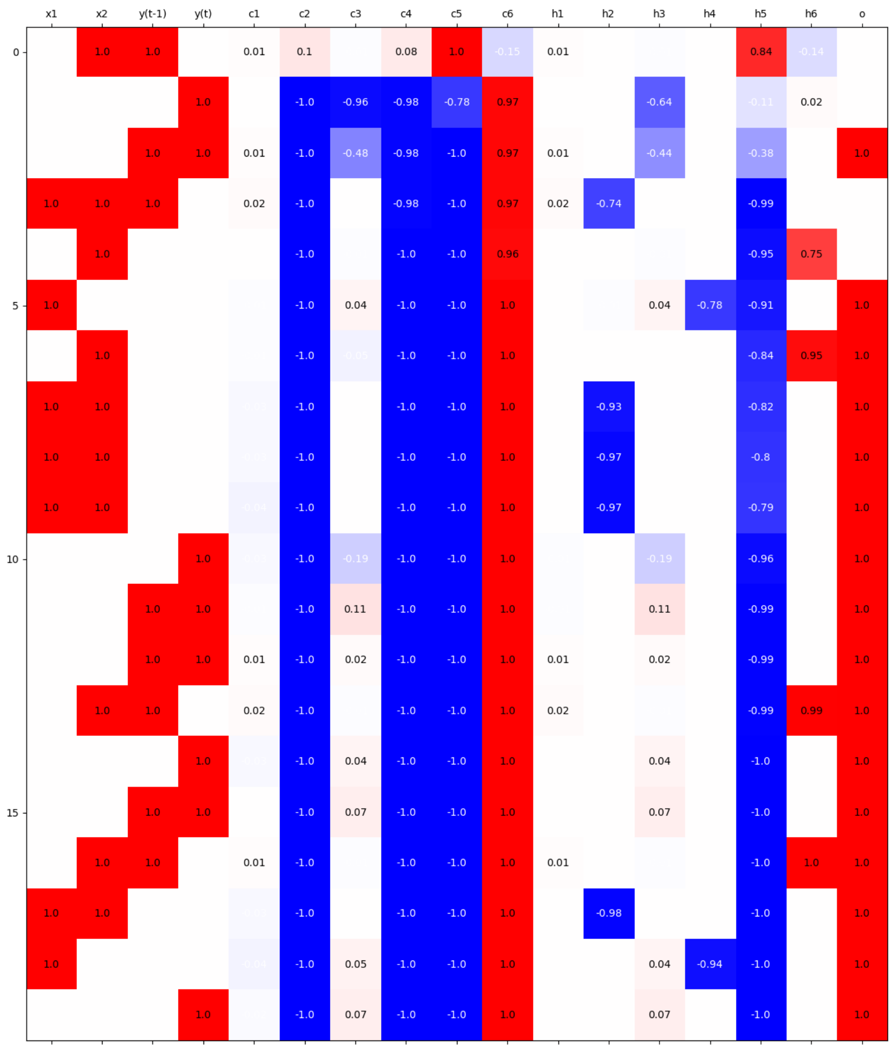


Figure 21: Boolean function 8

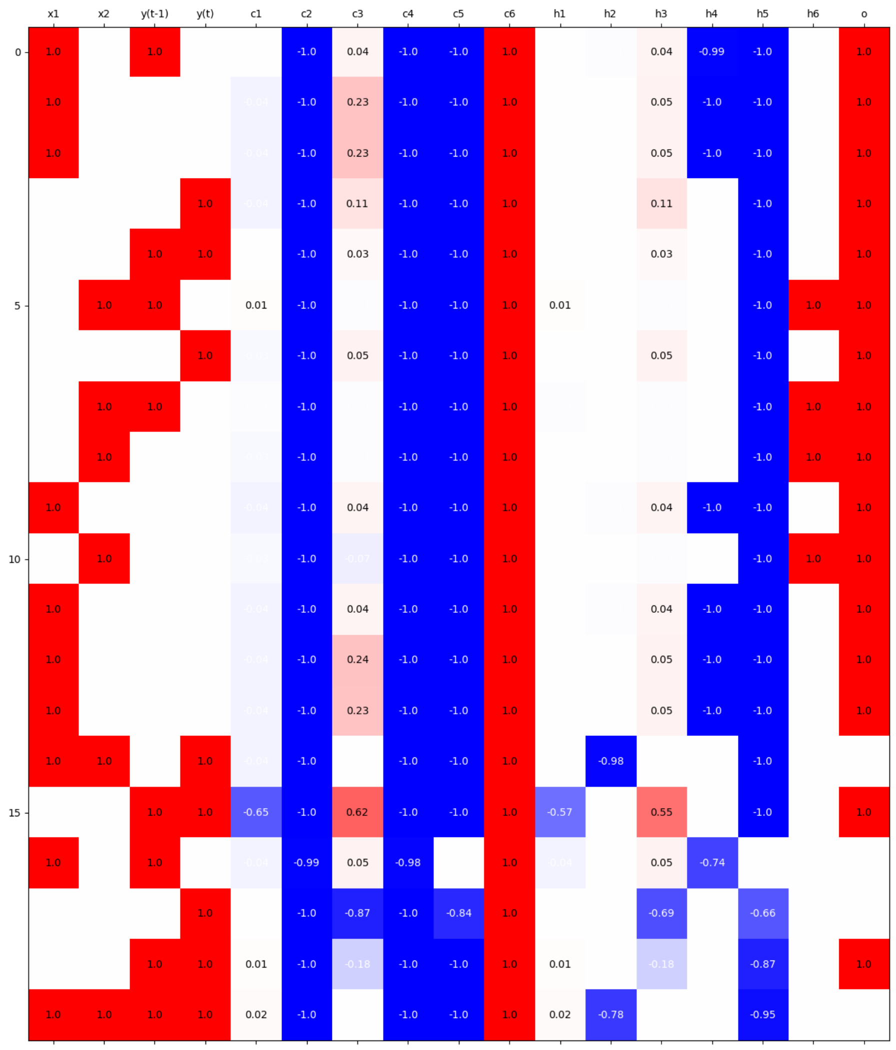


Figure 22: Boolean function 9

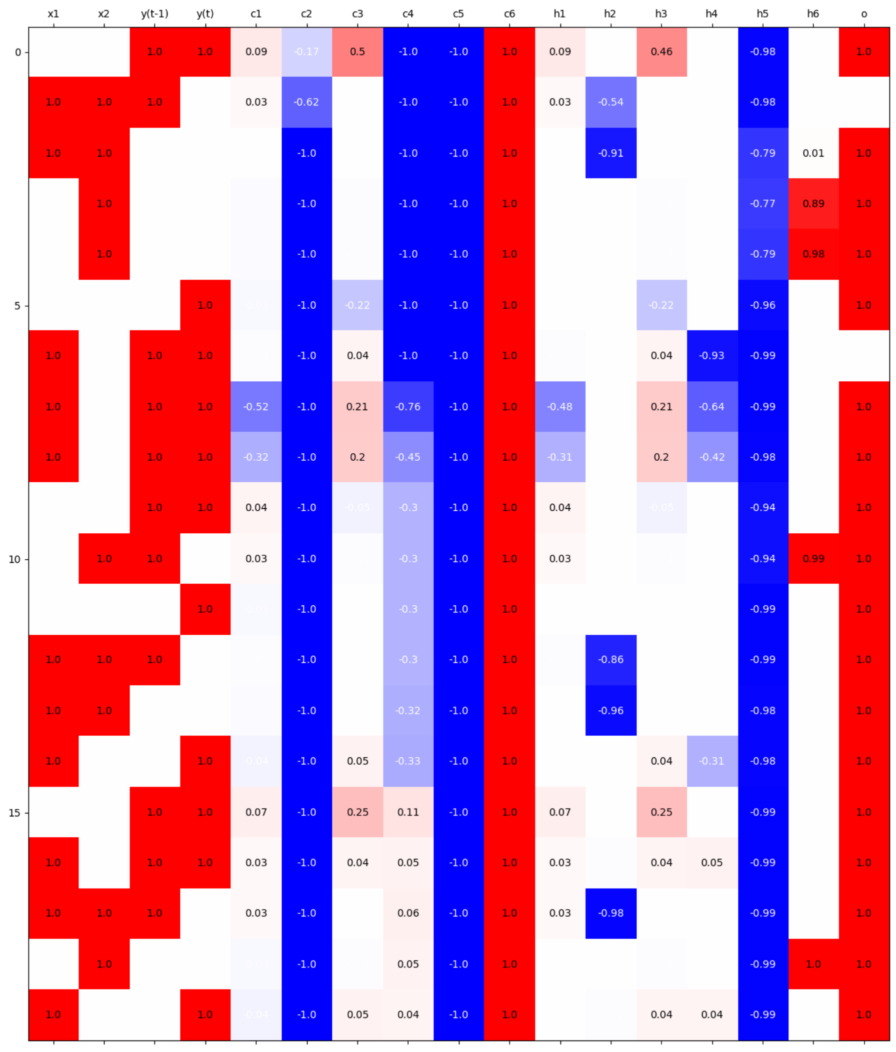


Figure 23: Boolean function 10

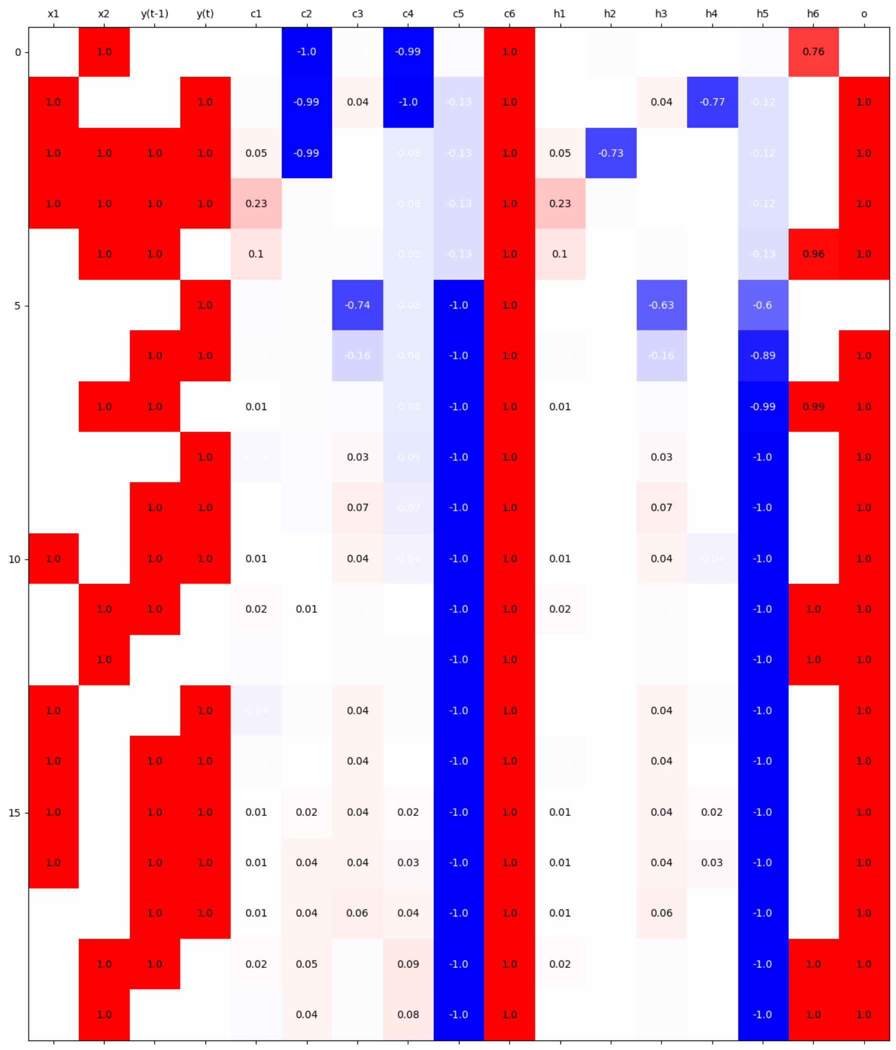


Figure 24: Boolean function 11

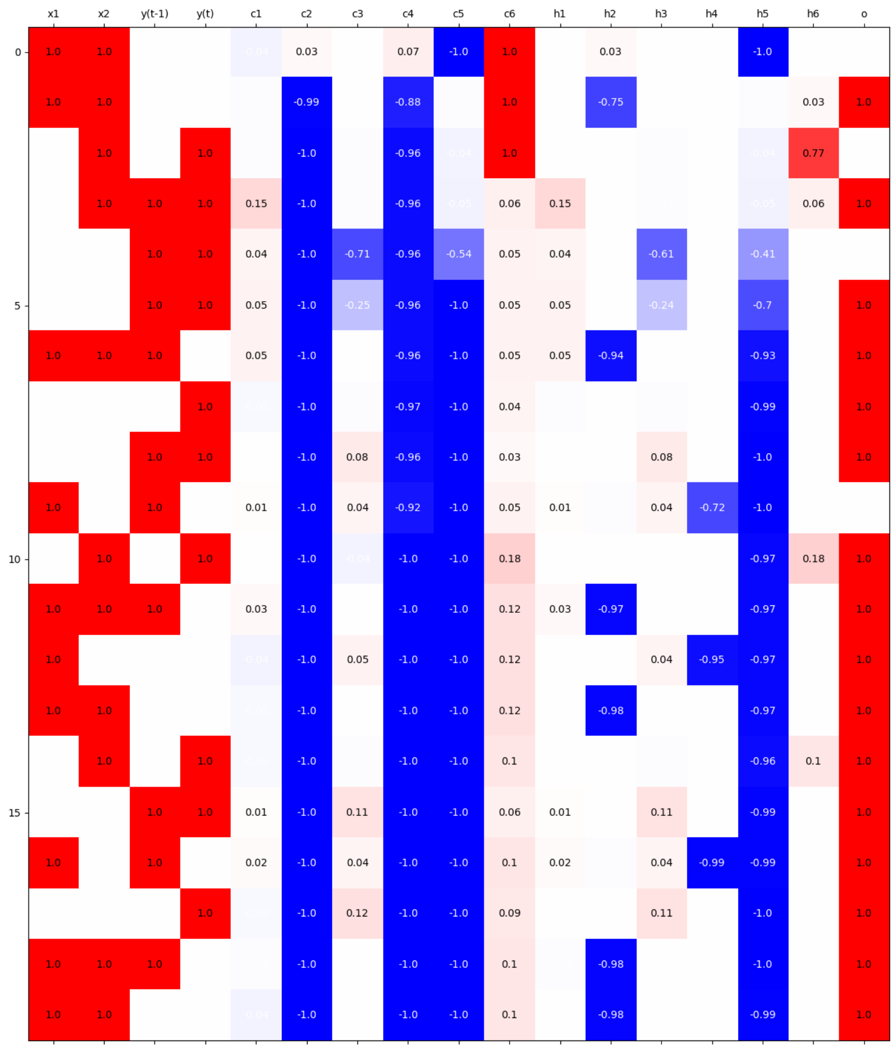


Figure 25: Boolean function 12

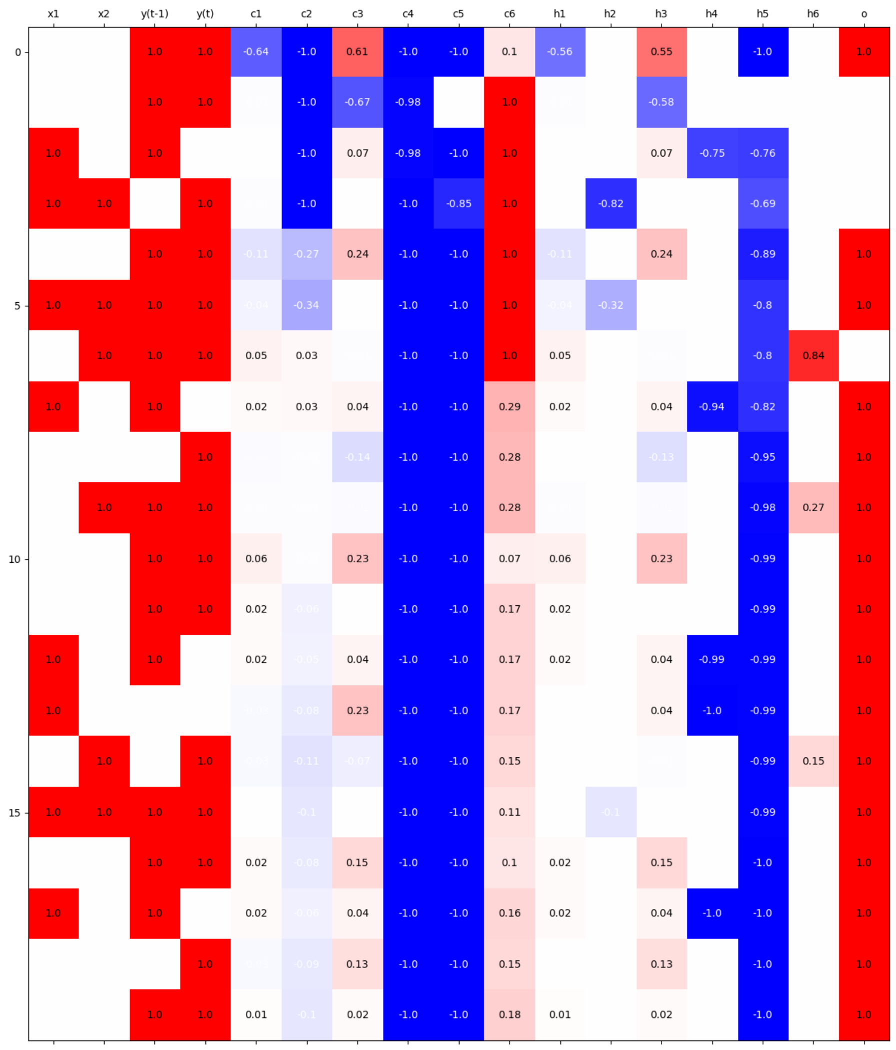


Figure 26: Boolean function 13

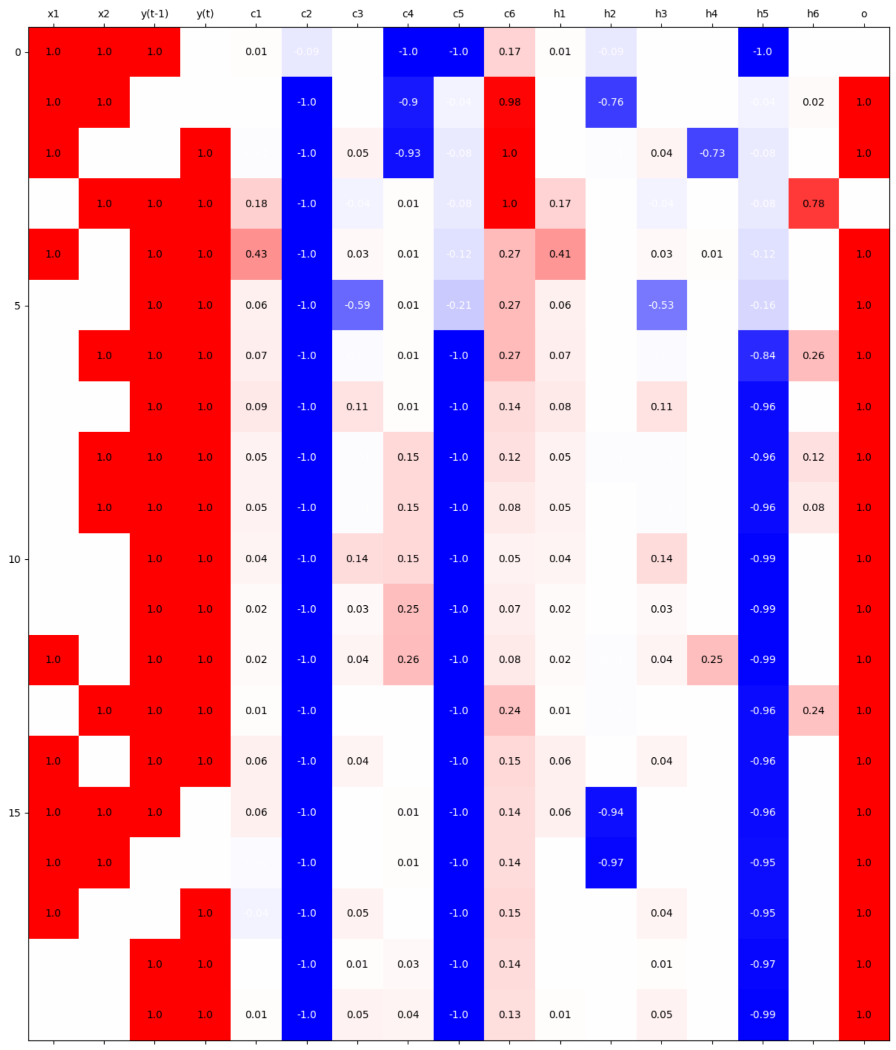


Figure 27: Boolean function 14

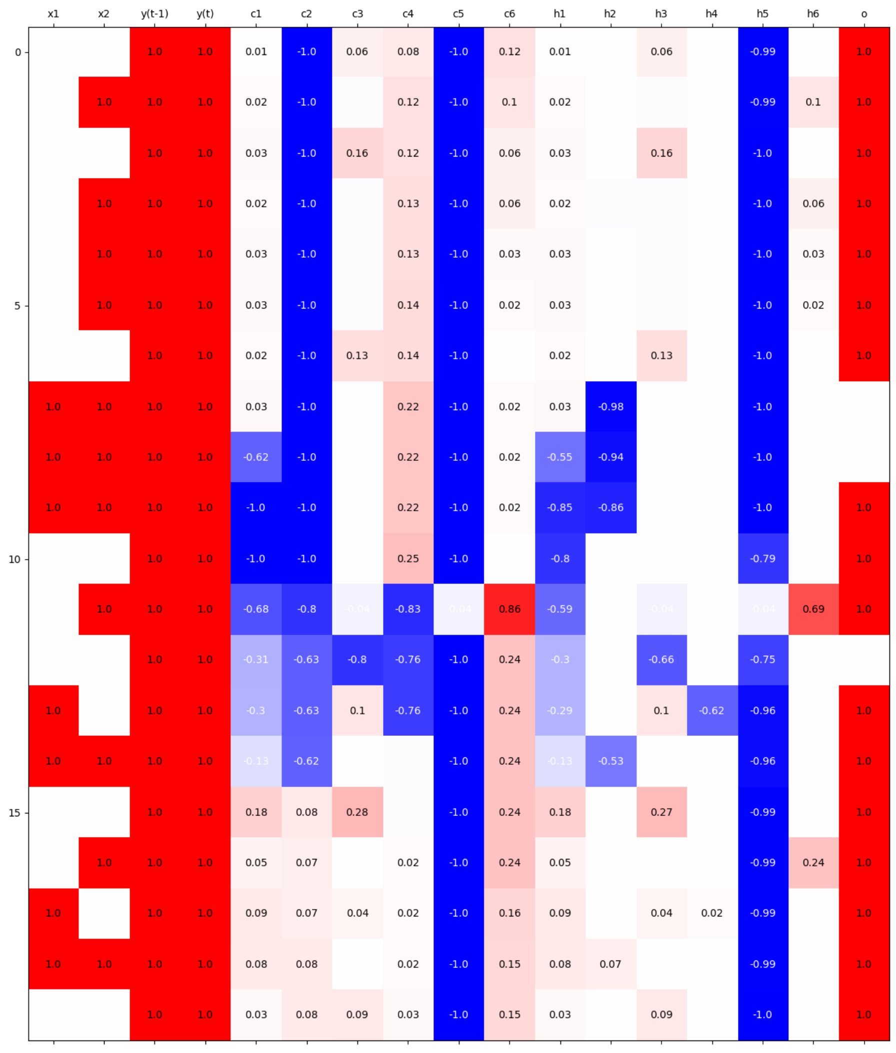
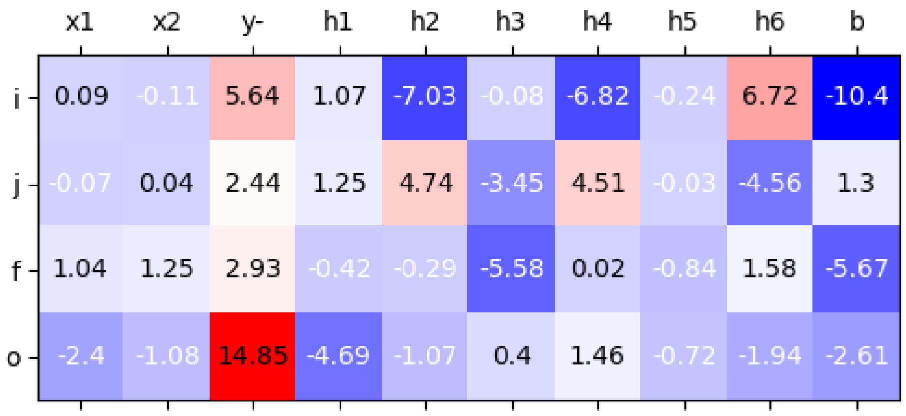
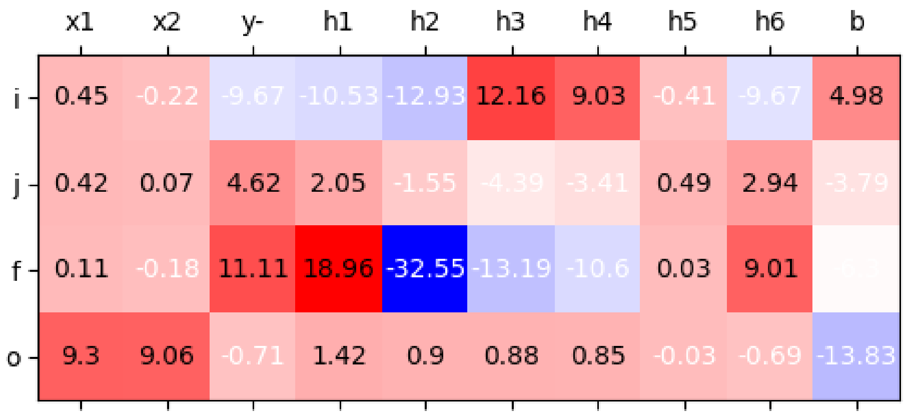


Figure 28: Boolean function 15



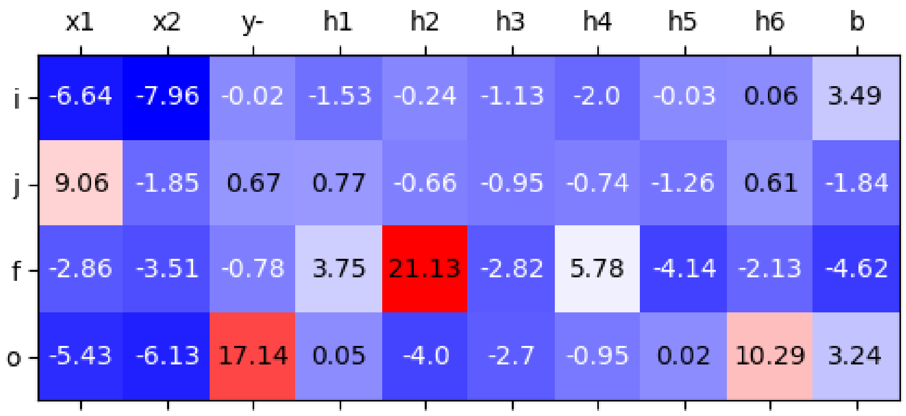
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	0.09	-0.11	5.64	1.07	-7.03	-0.08	-6.82	-0.24	6.72	-10.4
j	-0.07	0.04	2.44	1.25	4.74	-3.45	4.51	-0.03	-4.56	1.3
f	1.04	1.25	2.93	-0.42	-0.29	-5.58	0.02	-0.84	1.58	-5.67
o	-2.4	-1.08	14.85	-4.69	-1.07	0.4	1.46	-0.72	-1.94	-2.61

weight matrix 0



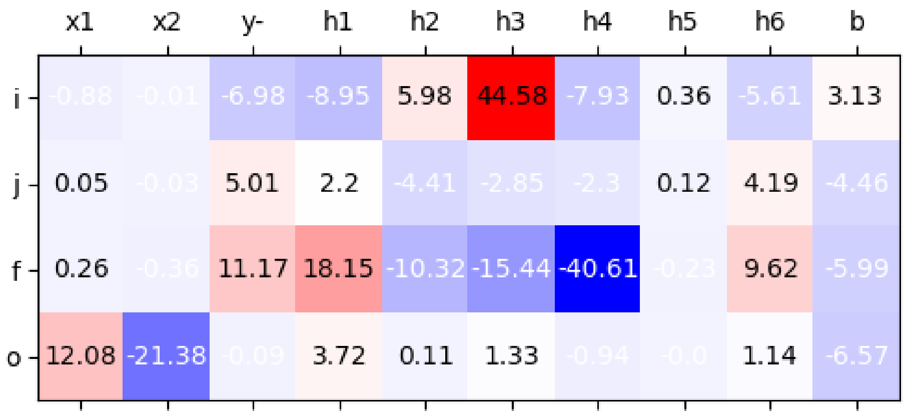
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	0.45	-0.22	-9.67	-10.53	-12.93	12.16	9.03	-0.41	-9.67	4.98
j	0.42	0.07	4.62	2.05	-1.55	-4.39	-3.41	0.49	2.94	-3.79
f	0.11	-0.18	11.11	18.96	-32.55	-13.19	-10.6	0.03	9.01	-6.3
o	9.3	9.06	-0.71	1.42	0.9	0.88	0.85	-0.03	-0.69	-13.83

weight matrix 1



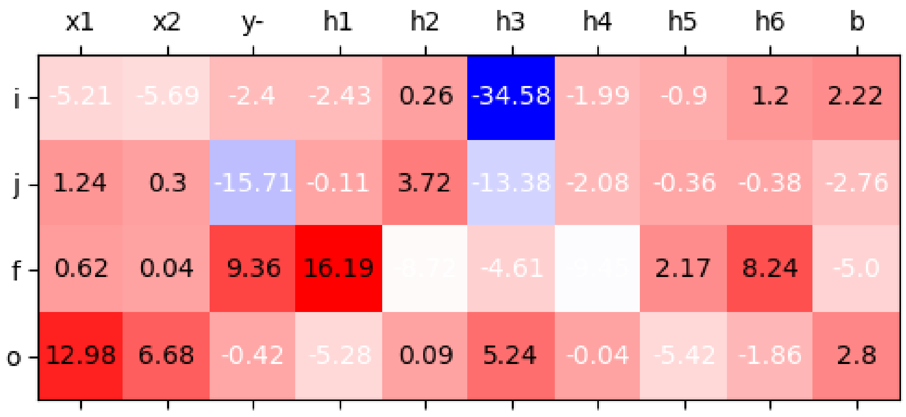
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-6.64	-7.96	-0.02	-1.53	-0.24	-1.13	-2.0	-0.03	0.06	3.49
j	9.06	-1.85	0.67	0.77	-0.66	-0.95	-0.74	-1.26	0.61	-1.84
f	-2.86	-3.51	-0.78	3.75	21.13	-2.82	5.78	-4.14	-2.13	-4.62
o	-5.43	-6.13	17.14	0.05	-4.0	-2.7	-0.95	0.02	10.29	3.24

weight matrix 2



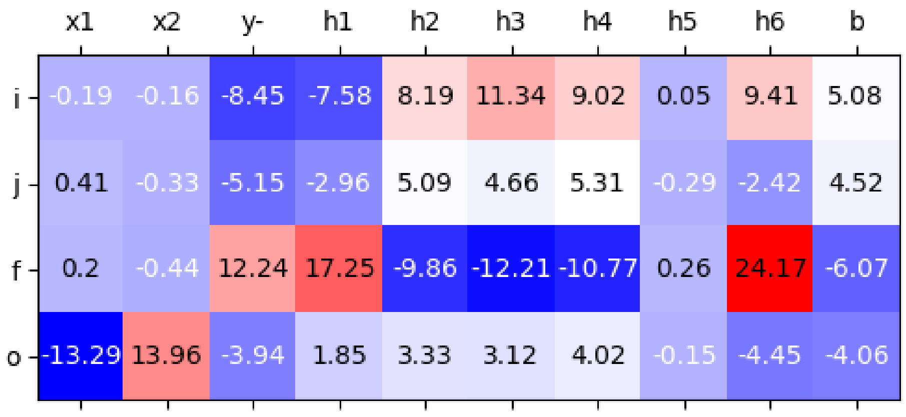
	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-0.88	-0.01	-6.98	-8.95	5.98	44.58	-7.93	0.36	-5.61	3.13
j	0.05	-0.03	5.01	2.2	-4.41	-2.85	-2.3	0.12	4.19	-4.46
f	0.26	-0.36	11.17	18.15	-10.32	-15.44	-40.61	-0.23	9.62	-5.99
o	12.08	-21.38	-0.09	3.72	0.11	1.33	-0.94	-0.0	1.14	-6.57

weight matrix 3



	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-5.21	-5.69	-2.4	-2.43	0.26	-34.58	-1.99	-0.9	1.2	2.22
j	1.24	0.3	-15.71	-0.11	3.72	-13.38	-2.08	-0.36	-0.38	-2.76
f	0.62	0.04	9.36	16.19	-8.72	-4.61	-9.45	2.17	8.24	-5.0
o	12.98	6.68	-0.42	-5.28	0.09	5.24	-0.04	-5.42	-1.86	2.8

weight matrix 4



	x1	x2	y-	h1	h2	h3	h4	h5	h6	b
i	-0.19	-0.16	-8.45	-7.58	8.19	11.34	9.02	0.05	9.41	5.08
j	0.41	-0.33	-5.15	-2.96	5.09	4.66	5.31	-0.29	-2.42	4.52
f	0.2	-0.44	12.24	17.25	-9.86	-12.21	-10.77	0.26	24.17	-6.07
o	-13.29	13.96	-3.94	1.85	3.33	3.12	4.02	-0.15	-4.45	-4.06

weight matrix 5