Artificial Intelligence

Statistical Inference, Rules, Neural Networks, and GPU

Pedro Cosme da Costa Vieira
PENTAGON, Portugal
October 2024

Abstract: Nowadays, discussing Artificial Intelligence means discussing supervised multilayer neural networks solved with groups of thousands of GPUs with astronomical computational capacities. However, the foundation of AI is the simple rule, y(x) = 0 if x < k else 1, which statistically divides data into two subgroups. In this text, I will talk about rules, Shannon's information gain, the equivalence between rules and neural networks, and how viewing the neural network as a matrix operation unexpectedly made GPUs central to solving AI models.

Keywords: Artificial Intelligence, Shannon Information, Decision Rules, Neural Networks, Matrix Computation, GPU

1 = Introduction

The conceptual core of AI is inference by regression of a binary rule on an explanatory variable. Regression is the ability to infer the model that links the outcome, Y, to the causes X, based on observations (Yi, Xi) collected from the past. Statistics are central to AI because regression is never perfect whether due to missing variables, measurement errors, the functional form being an approximation of the true relationship, or randomness in the decision process.

$$Y = f(X) + \text{Error} \quad (1)$$

Wald (1950) proposes the concept of optimal decision rules as a statistical problem involving the choice of an action between two possible alternatives, Y, subject to exogenous variables, X. Then, the inference algorithm will separate the observations into two sets, A and B, as dissimilar as possible in terms of entropy, meaning the rule will maximize Shannon's Information Gain (1948).

In the following figure, we have Y=f(X1, X2)+Error, where Y is one (in blue) or zero (in brown).

The rule on X1 will separate the observations into two sets, A with X1 < K and B with X1 >= K, where P1 is the proportion of 1's in the set. The entropy will be given by:

$$Ent(P1) = -P1 * \log_2(P1) - (1-P1) * \log_2(1-P1) \quad (2)$$

The boundary value, K, will be determined in a way that maximizes Shannon's Information Gain (1948), given by the difference between the entropy of the original set, Ent(Or), and the entropy of the two sets created by the rule, Ent(A):

$$Gain(K) = Ent(O) - (Ent(A)*len(A) + Ent(B)*len(B)/(len(A)+len(B)) \quad (3)$$

$$\{K: Gain(K) \text{ is maximum } \} \quad (4)$$

Thus, the inference of the rule from the observations results from a maximization problem over statistical quantities.

The elementary algorithm has only one variable per rule, and the decision is 0 or 1. The power of the model arises from sequentially inferring rules in each subset until there is only a certain number of observations in each subset, for example, 10 observations.

When there are multiple alternatives in the decision (0, 1, 2 or 3), there will be a rule for each possible alternative, always treated against 0. For instance, the decision is 0 or 2, where 0 represents any of the other possible alternatives (0, 1 or 3).

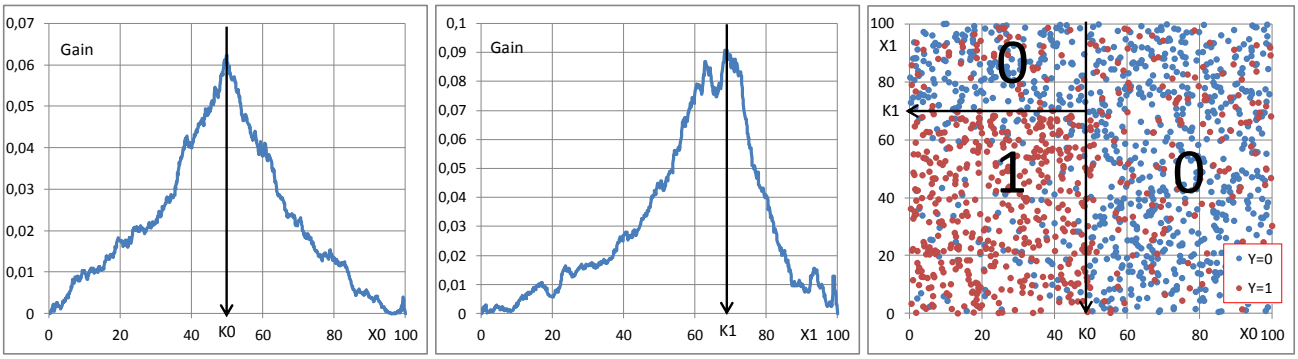

Fig. 1 – Shannon information gain (left, X0 and X1) and data (right)

The AI model inferred from the data is the rule:

$$Y = 1 \text{ if } (X0 < 49.84 \text{ and } X1 < 69.66) \text{ else } 0 \quad (5)$$

Important note: The percentage accuracy when using this model on the data from Fig. 1 is 70.1%, but this number does not play a central role in the model's evaluation because it depends on the original decision-making process. For example, if the original process is entirely random, the model could never achieve a high accuracy rate (if the original decision-maker were to decide again, the decisions would be different).

```
def f_data(n_obs,k0,k1,p_error):
    #model of the nature that creates data
    obs=[]
    for i in range(n_obs):
        X0=round(random.random()*100,6)
        X1=round(random.random()*100,6)
        cond0= X0 < k0
        cond1= X1 < k1
        Y= 1 if (cond0 and cond1) else 0
        Y = (1-Y) if random.random() < p_error else Y
        obs.append([X0,X1,Y])
    return sorted(obs)
```

```
def f_entropy(obs):
    sum_i=0
    for ob in obs:
        sum_i += ob[2]
    prop_i= sum_i/len(obs)
    entropy=-prop_i * math.log2(prop_i+1e-10)
    entropy +=-(1-prop_i) * math.log2(1-prop_i+1e-10)
    return entropy

def f_shannon_gain(obs,variable,k,entropy_i):
    sumA=sumB=nA=nB=0
    for ob in obs: #proportions
        if ob[variable] < k:
            sumA += ob[2]
            nA+=1
        else:
            sumB += ob[2]
            nB+=1
    if nA*nB > 0: #entropy gain
        propA,propB = sumA/nA,sumB/nB
        entropyA = -propA * math.log2(propA+1e-10)
        entropyA += -(1-propA) * math.log2(1-propA+1e-10)
        entropyB = -propB * math.log2(propB+1e-10)
        entropyB += -(1-propB) * math.log2(1-propB+1e-10)
        gain = entropy_i-(entropyA*nA+entropyB*nB)/(nA+nB)
    else:
        gain = 0
    return gain
```

I used these functions in the creation of the numerical example represented in the previous figure. I used trisection algorithm for maximizing information gain.

```
def f_learn_rule(obs,variable, a, b, tol=0.01, max_iter=10000):
    entropy_i=f_entropy(obs)
    for _ in range(max_iter):
        x1 = a + (b - a) / 3
        x2 = b - (b - a) / 3
        f_x1 = f_shannon_gain(obs,variable,x1,entropy_i)
        f_x2 = f_shannon_gain(obs,variable,x2,entropy_i)
        if f_x1 < f_x2: a = x1
        else          : b = x2
        if abs(b - a) < tol: break
    return (a + b) / 2


#Create data
k0, k1, n_obs, p_error = 50, 70, 1500, 0.3
random.seed(123)
obs=f_data(n_obs,k0,k1,p_error)

#learn the roule on X0
k0= f_learn_rule(obs,0, 0, 100, 0.01, 10000)
#learn the rule on X1, on the subgroup X0<k0
A=[]
for ob in obs:
    if ob[0] < k0: A.append(ob)
k1 = f_learn_rule(obs,1, 0, 100, 0.01, 10000)
print("k0=",k0,"k1=",k1)
```

2 - The Rule in the Form of a Neural Network

We can take the previously inferred model with the two rules and interpret it as a neural network with 4 McCulloch & Walter (1943) neurons organized into two layers.

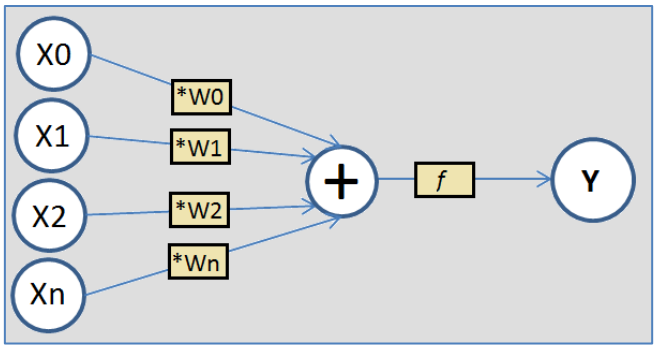

Fig. 2 – Representation of a neuron (the X's are the inputs, the W's are the weights, and *f* is the activation function).

The first layer has three neurons. The first neuron simply outputs the constant 1, and each of the other neurons takes as input the variable X0 or X1, which is multiplied by -1 and added to the threshold of the rule, K0 or K1, respectively. Then, the Heaviside (1892) step activation function is applied to the result of each neuron:

$$H(x) = (0 \text{ if } x < 0 \text{ else } 1) \tag{6}$$

The inputs to the second layer are the outputs from the first layer. In this layer, the outputs from the first layer are summed, and the constant 2, which represents the number of inputs, is subtracted. Then, the activation function H(x) is also applied.

If the neural network represents a set of rules but in a manner that appears to be more complex, it does not seem to make sense for them to become increasingly central in AI models. We will see that the advantage of neural networks lies in the systematization achieved by interpreting the rules as neurons, consequently facilitating the design of processors capable of solving a generic AI model, that is, GPU as AI accelerators.

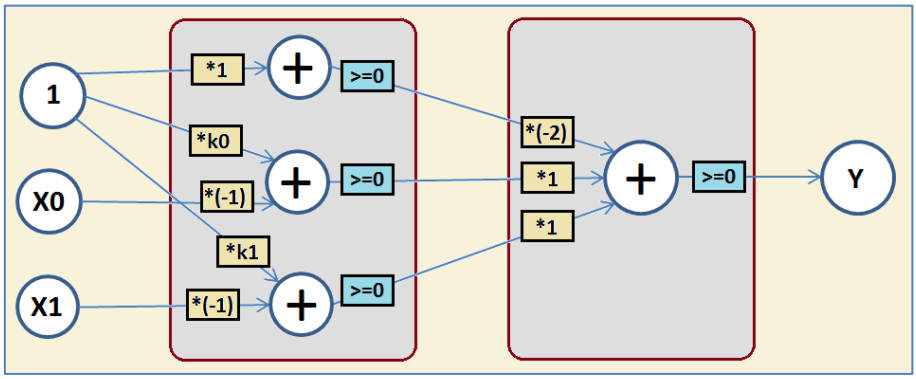

Fig. 3 – The two rules inferred from the data in Fig. 1 in the form of a neural network with 4 neurons.

The constants that we will use to multiply the inputs, 49.84, 69.66, -1, 1, and -2 in the specific case of my rule, are the flexible components that make it possible to solve a multitude of problems using the same abstraction model. For example, if the rule contained OR instead of AND, it would be sufficient to replace the constant -2 in the second layer with -1.

By increasing the number of neurons and layers, the neural network can perform more sophisticated tasks and solve more difficult problems. For instance, ChatGPT-4o is expected to have around 6 million neurons organized into 92 layers, with a total of about 400 billion parameters.

3 = Levels of Abstraction

If we were to develop a processor specialized in solving a specific problem, it would be very computationally efficient because, in metal, operations are executed very quickly—on the order of 5 billion operations per second—because they are simple and optimized. However, in economic terms, it would be a disaster because developing a new competitive processor (designing, testing, and making the lithography masks) incurs very high fixed costs, exceeding one billion Euros, which can only be manageable if diluted over many units sold. Successive levels of abstraction can solve very different problems with the same metal, thereby increasing potential sales. For example, it is estimated that 2 billion Euros were invested in the design of Apple's M2 processor, which is sold for around 400 Euros per unit. This price is only possible because the fixed development cost was intended to be diluted over 50 million units, translating to 40 Euros per unit.

The first level of abstraction is the ISA – Instruction Set Architecture. To improve efficiency, with each new generation of processors, the operations that the metal performs must change, but for the software, everything must remain the same (otherwise, programs would have to be rewritten every time a new processor is released). Therefore, there must be an abstraction layer between the metal and the instructions that programs use: the microcode, which constructs the instructions that the program uses (the ISA) using the instructions that the metal can execute. For example, the program uses the 64-bit Multiply operation, which the microcode resolves with 64 Sum operations. Economically, the ISA abstraction level allows for the dilution of the fixed development costs of programs (extending their use over time). Equally important is that microcode also allows parts of the metal developed and used in older processors to be reused in subsequent processors; for instance, cores used in the M2 may be reused in the M3 processor, further diluting the fixed costs of processor design.

The neural network is merely an abstraction that allows for the resolution of generic AI problems.

4 = The Neural Network Condenses into Matrix Form

The neural network in Fig. 3 is divided into layers, MLPs – Multilayer Perceptrons, and the outputs of one layer serve as the inputs to the next layer, FFN – Feed-forward Network. Although these rules are not mandatory, by following them, each layer can be treated as a multiplication between the vector X and the matrix W, and the neural network as a sequence of matrix operations. This systematization becomes central in the current state of AI because it allows the neural network to be solved by processors that were initially developed for image manipulation, namely GPUs.

If we look at the neuron in Fig. 2, each input is multiplied by a weight, and the products are summed:

$$Y1 = W0.X0 + W1.X1 + W2.X2 + \ldots. + Wn.Xn \quad (7)$$

So, the neuron encodes an equation in which the independent term results from the input X0 being 1.

If a neuron encodes an equation, we can encode a system of equations using a layer with m neurons. To systematize the encoding, all inputs will connect to all nodes, FCN - Fully Connected Network, where we encode the 'non-connection' of Xi in equation j with Wji=0 and the independent term with X0=1:

$$\begin{aligned} Y1 &= W10.X0 + W11.X1 + W12.X2 + \ldots + W1n.Xn \\ Y2 &= W20.X0 + W21.X1 + W22.X2 + \ldots + W2n.Xn \\ Ym &= Wm0.X0 + Wm1.X1 + Wm2.X2 + \ldots + Wmn.Xn \end{aligned} \quad (8)$$

This function condenses into matrix form:

$$[1 \quad Y1 \quad Y2 \quad Yn] = [1 \quad X1 \quad X2 \quad Xn] * \begin{bmatrix} W10 & W20 & Wm0 \\ W11 & W21 & Wm1 \\ W12 & W22 & Wm2 \\ W1n & W2n & Wmn \end{bmatrix} \quad (9)$$

A layer of the neural network is then the graphical representation of the matrix function where each dimension/component of the function is viewed as a neuron. For example, the first layer of the neural network in Fig. 3 can be formalized as this matrix multiplication:

$$[1 \quad Y0 \quad Y1] = [1 \quad X0 \quad X1] * \begin{bmatrix} 1 & 49.84 & 69.66 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad (10)$$

Thus, the layer of the neural network is a linear regression model where the input and output are vectors of dimension n and m, respectively.

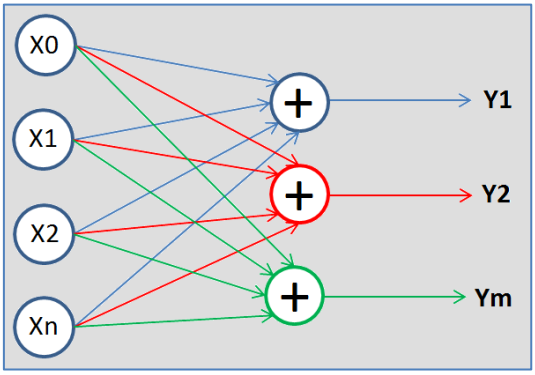

Fig. 4 – Layer of a full connected neural network with 4 inputs and 3 outputs.

In the following function, I will multiply the input vector (a matrix with one row) x_vector by the weight matrix W_matrix and return the result Y_vector in the form of a vector (a matrix with one row) so that it can be used in the next layer as input. The variable dec specifies the number of decimal places in the result to introduce the concept of "processors with few bits":

```
def f_multiply_vector_by_matrix(X_vector, W_matrix,dec=3):
    # Initialize the result vector with zeros
    Y_vector = [0] * len(W_matrix[0])
    # Iterate over the columns of the matrix W
    for j in range(len(W_matrix[0])):  # Number of columns in W
        # Calculate the value of multiplication for each component of Y
        for i in range(len(X_vector)):  # Number of elements in vector X
            Y_vector[j] += X_vector[i] * W_matrix[i][j]
        Y_vector[j] = round(Y_vector[j],dec)
    return Y_vector
```

5 = Activation Function

The neural network formed by only one layer, represented in Fig. 4, is linear and, therefore, not suitable for representing nonlinear phenomena. For example, the second-degree functional form would be:

$$Y1 = W10 + W11.X1 + W12.X1.X1 + W13.X1.X2 + W14.X2 + W15.X2.X2 \quad (11)$$

The activation function and the use of multiple layers will introduce nonlinearities into the function without losing the simplicity of the matrix representation. It is important to keep matrix multiplication as the core of the function to facilitate the design of generic processors to be used in the computation of specific neural networks. The most commonly used activation functions are the Heaviside step function, sigmoid, and hyperbolic tangent:

$$H(x) = 0 \text{ if } x<0 \text{ else } 1, \text{ with co-domain } \{0, 1\} \quad (12)$$

$$Sig(x) = 1/(1+\exp(-x)), \text{ with co-domain }]0, 1[\quad (13)$$

$$Tanh(x) = 2/(1+\exp(-2x))-1, \text{ with co-domain }]-1,1[\quad (14)$$

The Softmax function relates all the outputs, imposing that they sum to 1:

$$\begin{aligned} &Exp\_Yi = \exp(Yi) \\ &Total = Exp\_Y0 + Exp\_Y0 + \ldots + Exp\_Ym \\ &SoftMax(Yi) = Exp\_Yi/Total \end{aligned} \quad (15)$$

The activation function will also contain values in a limited co-domain; for example, the Sig(x) function keeps the output between 0 and 1, which allows the use of processors with few bits, such as 8 bits. In this case, the possible values for the output of each neuron will be:

8 bits	0	1	2	3	4	…	253	254	255
Values	0,0000	0,0039	0,0078	0,0117	0,0157	…	0,9922	0,9961	1,0000

The Sigmoid(x) activation function is suitable when the components of Y are independent; the Heaviside, H(x), function is suitable for logical operations and inequalities; the Softmax(x) function is suitable when there is a connection between the components of Y (for example, a decision within a set of possible decisions) and can be interpreted as the Bayesian probability of observing Xi.

```
import math
def f_H_vector(x_vector,dec=3):
    # Calculate the sigmoid function for each element in the input vector
    H_vector =[(0 if x < 0 else 1) for x in x_vector]
    return H_vector

def f_sigmoid_vector(x_vector,dec=3):
    # Calculate the sigmoid function for each element in the input vector
    S_vector =[1 / (1 + math.exp(-x)) for x in x_vector]
    for i in range(len(S_vector)):
        S_vector[i] = round(S_vector[i],dec)
    return S_vector
```

```
def f_softmax_vector(Z_vector,dec=3):
    # Calculate the exponential for each element in z
    exp_values_vector = [math.exp(zi) for zi in Z_vector]
    # Calculate the sum of all exponential values
    sum_exp_values = sum(exp_values_vector)
    # Calculate the softmax for each element
    softmax_vector = [round(value / sum_exp_values,dec) for value in exp_values_vector]
    return softmax_vector

def f_tanh_vector(x_vector,dec=3):
    # Calculate the sigmoid function for each element in the input vector
    tanh_vector =[2 / (1 + math.exp(-2*x))-1 for x in x_vector]
    for i in range(len(tanh_vector)):
        tanh_vector[i] = round(tanh_vector[i],dec)
    return tanh_vector
```

I can now create the function that calculates a layer of the neural network:

```
def f_neural_network_layer(X_vector, W_matrix,f_act="sigmoid",dec=3):
    # Step 1: Multiply X_vector by W_matrix to get Y_vector
    Y_vector = f_multiply_vector_by_matrix(X_vector, W_matrix, dec)
    # Step 2: Apply sactivation function to the result Y
    if   f_act == "sigmoid" : Y_act_vector = f_sigmoid_vector(Y_vector,dec)
    elif f_act == "tanh"    : Y_act_vector = f_tanh_vector(Y_vector,dec)
    elif f_act == "H"       : Y_act_vector = f_H_vector(Y_vector,dec)
    else                    : Y_act_vector = f_softmax_vector(Y_vector,dec)
    return Y_act_vector
```

The implementation of the neural network will be done through a succession of layers.

```
def f_neural_network(X_vector, Ws_matrix,f_activation="softmax",dec=3):
    #Ws=[W1,W2,W3]
    # Step 1: number of
    Y_vector=f_deepcopy(X_vector)
    for W_layer in Ws_matrix:
        Y_vector = f_neural_network_layer(Y_vector, W_layer,f_activation,dec)
    return Y_vector
```

6 = The neural network that encodes the model of the numerical example

```
#Create data
import random
k0, k1, n_obs, p_error = 50, 70, 1500, 0.3
random.seed(123)
obs=f_data(n_obs,k0,k1,p_error)
#two layers = two matrix
W_layers = [[[1,49.9,69.66],[0,-1,0],[0,0,-1]],  #Layer 1
            [[-2],[1],[1]]]                      #layer 2
equal_r=equal_nn=0
for ob in obs:#apply the rules to all the observations
    classe_r = 1 if ob[0]<49.9 and ob[1] < 69.66 else 0
    equal_r += 1 if (classe_r==ob[2]) else 0
    classe_nn =f_neural_network([1,ob[0],ob[1]], W_layers,"H",3)
    equal_nn += 1 if classe_nn[0]==ob[2] else 0
print ("Rule percentage of certain",equal_r/len(obs))
print ("NN percentage of certain",equal_nn/len(obs))
Rule percentage of certain 0.70133
NN percentage of certain 0.70133
```

7 = The matrix formalization allows the use of GPUs as AI accelerators.

A processor, the 64-bit CPU, must be capable of solving a multitude of problems to expand the potential market and thus dilute the high fixed costs of design. However, the emergence of video games opened a specific hardware market to solve graphical problems, the GPU. The GPU, in fact, performs matrix calculation operations directly in hardware with 8-bit numbers (the colour of a pixel has 3 tones, and each tone has 8 bits), lots of pixels at a time.

When we have a feed-forward neural network organized in layers and fully connected, we can encode each layer as a matrix operation followed by the application of the activation function, where X represents the n inputs of the layer (a matrix with 1 row and 1+n columns), W is the weight matrix (n+1 rows and m+1 columns), and Y are the m outputs (a matrix with 1 column and m+1 rows) that will serve as the inputs for the next layer:

$$Y = f(X*W) \tag{15}$$

For example, if I have 255 inputs and 255 outputs, to calculate one row, I need to perform 256 multiplications. If each of these operations takes one clock cycle and is executed sequentially, it will require 256 clock cycles.

The advantage of matrix formalization is that the multiplications are independent and can be executed simultaneously. In particular, if the numbers are 8 bits, each row can be calculated at once on a 4096-bit GPU.

Numerical example with numbers in the range [0, 255/256].

I want to multiply Y1 = 0.11010011 x 0.11110010 and Y2 = 0.00010101 x 0.11011101. Assuming that the numbers are in the range [0; 1[, the most significant bit is always 0.

I start by combining the numbers, allowing space for the sum of the terms:

0110100110000000000010101000000 x 0111100100000000011011101000000

Then, I can compute both multiplications on a 32-bit processor using just one operation:

```
  0110100110000000000010101000000
x 0111100100000000011011101000000
-----------------------------------
  0110100110000000000010101000000
  0011010011000000000010101000000
  0001101001100000000000000000000
  0000110100110000000000010101000
  0000000000000000000000010101000
  0000000000000000000000001010100
  0000000110100110000000000000000
  0000000000000000000000000010101
-----------------------------------
  1100011101110110000100100100001
```

Y1 = 0.11000111 and Y2 = 0.00010010

In hardware, it will be the implemented the sum of the first 8 bits of the multiplication of two 4096-bit numbers (a computing cluster), but in software, there appear to be 256 cores that multiply 256 pairs of 8-bit numbers in just one clock cycle.

To multiply all the lines at once, the processor will need to have 256 computing clusters of 4096 bits that appear to be 65536 cores.

With the numerical example, it becomes clear that there is an advantage in having numbers with few bits and in having numbers in a closed domain, whether in the interval [0; 1] or [-1; 1].

There are even neural networks with 4 bits.

Using a 4-bit neural network may seem insufficient for detecting subtle patterns in the data, but if we consider that the model has millions of neurons and parameters, we are dealing with millions of bits, and thus the model can detect sophisticated patterns. With 4 bits, applying the tanh activation function, the possible values will be:

-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
-1,00	-0,86	-0,71	-0,57	-0,43	-0,29	-0,14	0,00	0,14	0,29	0,43	0,57	0,71	0,86	1,00

References

Heaviside, Oliver (1892). *Electromagnetic Theory*, Volumes I e II. The Electrician Publishing Company.

McCulloch, Warren & Pitts, Walter (1943). “A Logical Calculus of Ideas Immanent in Nervous Activity”. *Bulletin of Mathematical Biophysics*, 5: 115-133.

Shannon, Claude Elwood (1948). “A Mathematical Theory of Communication”. *Bell System Technical Journal*, Part I at 27(3): 379–423 and Part II at 27(4): 623–666.

Wald, Abraham (1950). *Statistical Decision Functions*. John Wiley & Sons.

ChatGPT 4o was used to assist in writing this text.

Appendix 1- Numerical application

```
#data to be used as an application
n_obs=4000
p_error=0.25
random.seed(717)
#cria os dados
obs=[]
for i in range(n_obs):
    X0=random.random()*100
    X1=random.random()*100
    Y=0
    if (X0 < 20 and X1 < 40) : Y=1
```

```
    if (X0 > 70 and X1 < 30) : Y=1
    if (X0 > 50 and  X1 > 70) and not((X0 > 75 and X1 > 85)): Y=1
    Y = (1-Y) if random.random() < p_error else Y
    obs.append([X0,X1,Y])
obs=sorted(obs)
```

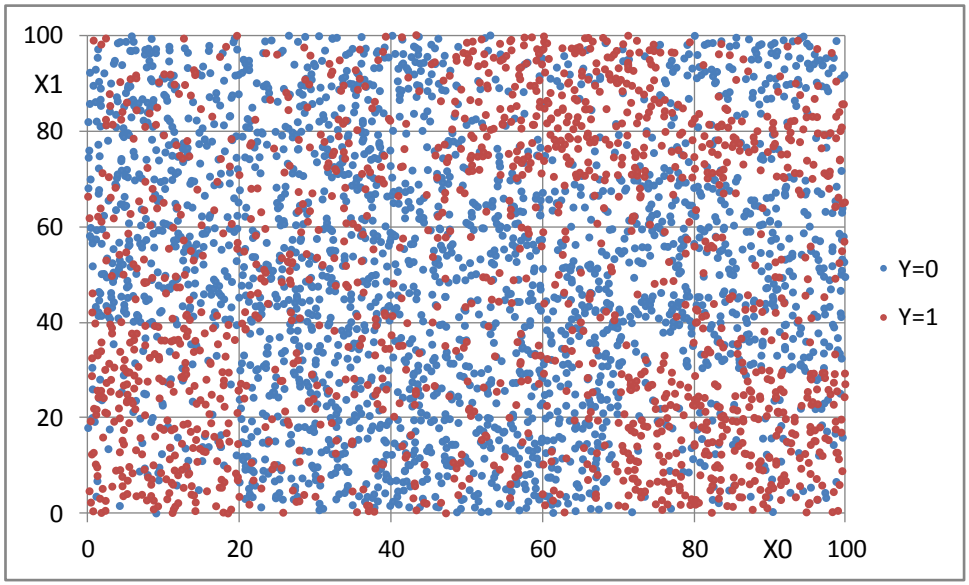

Fig. 5 – Data to be used as a numerical application.

```
def step(obs,variable,xy,n_min):
    global roules1
    k= f_learn_rule(obs, variable, xy[variable][0], xy[variable][1], 0.01, 100)
    A,B=[],[]
    if len(obs)>n_min:
        #two groups of observations
        for ob in obs:
            if (ob[variable] <= k): A.append(ob)
            else                  : B.append(ob)
        #<=K
        xyA=f_deepcopy(xy)
        xyA[variable][1]=round(k,3)
        if len(A)>0:
            step(A,1-variable,xyA,n_min)#change variable
        #>K
        xyB=f_deepcopy(xy)
        xyB[variable][0]=round(k,3)
        if len(B)>0:
            step(B,1-variable,xyB,n_min)
    else:
        y=0
        for ob in obs: y += int(ob[2])
        print(f"{xy} / {str(len(obs))} / {('x' if len(obs)==0 else
str(round(y/len(obs),2)))}")
        classe = 0 if len(obs)==0 else round(y/len(obs),2)
        if classe >= 0.50:
            rules1.append(xy)
    return True

rules1=[]
xy=[[0,100],[0,100]]
print(step(obs,0,xy,200))
```

The algorithm output is a classification tree in the form of 10 independent rules that result is y=1 and 19 independent rules that result is y=0. As there are just two classes, assuming zero as the base value, the neural network will implement just the first 16 rules.

Rule	Condition	X0 >	X0 <=	X1 >	X1 <=	n	Y
1	X0 < 20 and X1 < 40	0,00	19,99	0,00	6,02	46	0,89 = 1
2		0,00	5,92	6,02	39,34	76	0,82 = 1
3		5,92	19,99	6,02	39,34	172	0,69 = 1
4	X0 > 70 and X1 < 30	69,93	100,00	23,76	29,94	76	0,89 = 1
5		69,93	88,80	0,00	23,76	182	0,79 = 1
6		88,80	100,00	0,00	23,76	107	0,68 = 1
7	X0 > 50 and X1 > 70	62,49	74,94	69,98	100,00	166	0,80 = 1
8	and not(X0 > 75 and X1 > 85)	48,38	62,49	69,72	100,00	177	0,71 = 1
9		74,94	97,76	69,82	84,69	134	0,71 = 1
10		97,76	100,00	69,82	100,00	21	0,67 = 1
Y=1						1157	
11	Default value	26,08	34,43	51,63	78,86	102	0,35 = 0
12		19,99	50,55	0,00	6,28	79	0,32 = 0
13		37,81	62,49	21,25	39,34	171	0,30 = 0
14		0,00	13,59	39,34	66,22	182	0,29 = 0
15		0,00	13,59	66,22	92,67	154	0,24 = 0
16		34,43	48,38	51,63	78,86	156	0,24 = 0
17		26,08	48,38	78,86	92,67	124	0,24 = 0
18		74,94	86,38	29,94	69,82	192	0,24 = 0
19		62,49	69,93	0,00	29,94	93	0,23 = 0
20		19,99	37,81	21,25	39,34	133	0,22 = 0
21		13,59	48,38	39,34	51,63	181	0,21 = 0
22		13,59	26,08	51,63	92,67	161	0,21 = 0
23		62,49	74,94	29,94	69,98	198	0,21 = 0
24		48,38	62,49	39,34	69,72	151	0,20 = 0
25		86,38	100,00	29,94	69,82	197	0,20 = 0
26		74,94	97,76	84,69	100,00	123	0,20 = 0
27		50,55	62,49	0,00	21,25	123	0,18 = 0
28		19,99	50,55	6,28	21,25	200	0,17 = 0
29		0,00	48,38	92,67	100,00	123	0,17 = 0
Y=0						2843	

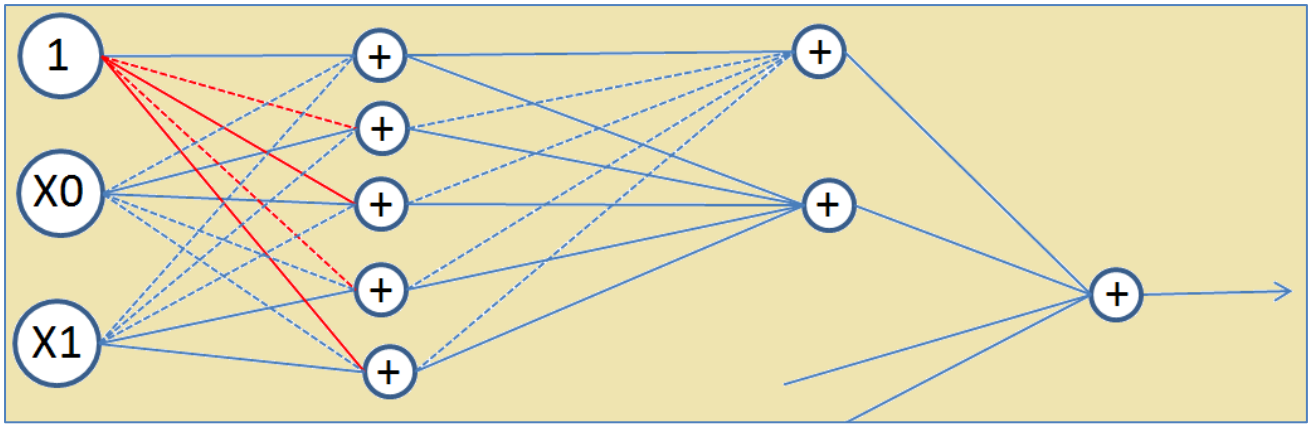

Fig. 6 – Each rule is a neural network with 8 neurons.

The neural network will have 3 layers and, for each rule, first layer will have 4 neurons for each rule, a matrix 3x65, and the second layer will have one neuron for each rule, a matrix 65x17 (it is the logical operation AND). The last layer has just one neuron, a matrix 17 x 1 (it is the logical operation OR). For example, the rule 3 will be:

$$\text{Rule3} = \text{f}\left(\text{f}\left([1 \quad \text{X0} \quad \text{X1}] * \begin{bmatrix} 1 & -5.92 & 19.99 & -6.02 & 39.34 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \rightarrow\right) * \begin{bmatrix} -4 \\ 1 \\ 1 \\ 1 \\ 1 \\ \downarrow \end{bmatrix}\right) \tag{16}$$

f(x)= 1 if x >= 0 else 0

If we use as framework a neural network with layers of 256 neurons, most of the weights will be zero.

Now, using the model “learned” from data, I can compute the well classified cases.

```
n_right=0
for ob in obs:
    result = False
    for rule in rules1:
        result = result or (ob[0] > rule[0][0] and ob[0] <= rule[0][1] and
            ob[1] > rule[1][0] and ob[1] <= rule[1][1])
    result = 1 if result else 0
    if ob[2] == result: n_right +=1
print(n_right, n_right / n_obs)
```

3065 0.76625