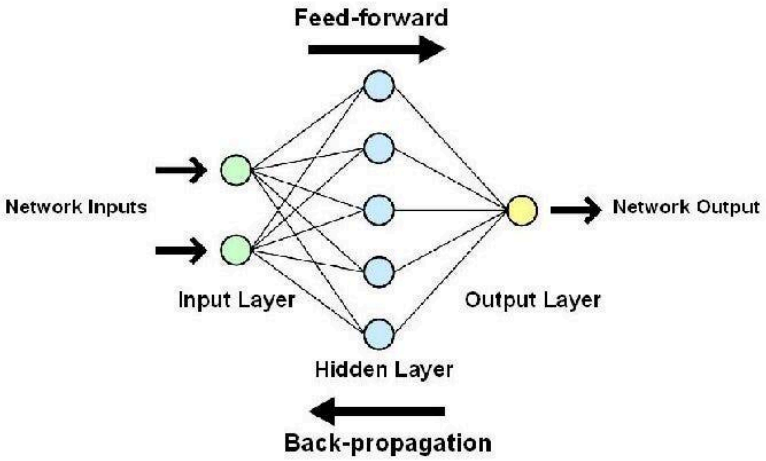


Purpose and functioning of the trial:

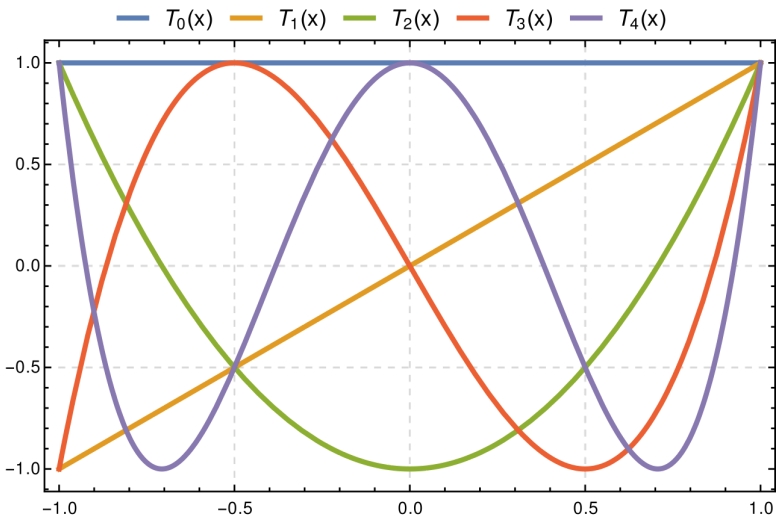
This is an attempt at comparing the efficiency and intrinsic differences that impose the choosing of the activation function for an ANN (a neural network with only just one hidden layer, which by the approximation is theoretically analogous to some smaller multiple layer NN aka DNN through the **limit theorem**).
In this example the two scenarios we chose are: a variation of the flappy bird challenge by the coding train, using a variation of the codes provided by him. And the second scenario is a regression problem, where the ANN should predict random values (between x= -1 and +1) of a function of our choosing, preferably continuous, otherwise it would counter the very theoretical assumptions and explanation of why this problem may be tackled using ANN.
Said function is passed to the ANN as an input in the following way:
"console.log(nn.predict(point.map(x=>f(x)=Math.pow(x,3)+3/2*Math.pow(x,2)-Math.sin(x)))); <-----it predicts the value, here f(x)=x^3+2/2....
console.log((ra.map(x=>f(x))));<-----here we check the real random value of f.
"

That is to say, in this specific case we know what the function is and expect de ANN to get its correct value at ra(random x), by feeding it the values at "points". We could also assume that we only know some values of f and expect to reconstruct it by changing the array "points" to the array whereby f has already been evaluated (this is a typical interpolation problem), this yields the desired random evaluation and is altogether a separate case, though not much different than the current one.

Explanation of the second scenario:

We chose the second scenario with the Chebyshev approximation in mind. The chebyshev approximation relies on approximating a continuous function through a sum of N chebyshev polynomials. Say the function we desire to approximate is f, we can do so by summing the chebyshev polynomials -whose recurrent and easy construction we are aware of- multiplied with their respective coefficients, which depend on the chebyshev nodes and the evaluations of f in the latter. Were f to be evaluated at random, we can approximate said random value with chebyshev and this would imply a sum of constants (biases) and a complicated weighted sum of the f(xk) being [xk] an array of N chebyshev nodes, these nodes are the most important to the approximation or any

approximation given that they ensure the least possible error within the [-1,1] interval. That is why we chose the chebyshev approximation over lagrange or newton; its simplicity for our particular case and its minimum error.



The chebyshev first order polynomials (N=5)

This being said, it is clear that what we intend our ANN to "learn" through trial and error in its training is to find the specifics weights of f(xk) for our random x in the hidden to output weights, and to identify which of the equally spaced nodes are the most important before the activation.

This is no more than a test for a simple neural network, to determine its capabilities and a reminder of how powerful an activation function really is to achieve one's goals.

Meaning of the code, restrictions, and neuroevolution:

FIrst of all, the library nn.js is responsible for modeling the ANN. Given that there is only one single hidden layer, this translates the weights to matrices and the addition of two bias vectors. The allocation of the error in the backpropagating process is done equitably with respect to the weights (hidden→output) by which said error was achieved, in addition to the gradient descent:

```
let targets = Matrix.fromArray(target_array);

// Calculate the error
// ERROR = TARGETS - OUTPUTS
let output_errors = Matrix.subtract(targets, outputs);
// Calculate gradient
let gradients = Matrix.map(outputs, this.activation_function.dfunc);
gradients.multiply(output_errors);
gradients.multiply(this.learning_rate);

// Calculate deltas
let hidden_T = Matrix.transpose(hidden);
```

```
  let weight_ho_deltas = Matrix.multiply(gradients, hidden_T);
```
The backpropagating process for the input→hidden layers is similar.
For the second scenario we train the neural network as follows:
```
let nn =new NeuralNetwork(2*Math.pow(N,2)+1,2*N,1);
        console.log(nn.predict(point.map(x=>1)));

        for (i=0;i<200000;i++){
                let r=Math.floor(Math.random()*funt.length);
                tdata=data[r];
                u=tdata.input;
                //console.log(u);
                d=tdata.output;

        nn.train(u,d);}
```
Where we randomly feed the ANN 20000 samples of the evaluations of functions of our choosing as inputs, and f(ra) as the output.

For the first scenario we modified the following lines of code:
Pipes.js:
```
class Pipe {
 constructor() {

  // How big is the empty space
  let spacing = 80;
        let x = random(1);
        if(x<0.35){
                //pipe up
                // Where is th center of the empty space
  let centery_1 = random(spacing, height/2 - spacing);

  // Top and bottom of pipe
  this.top_1 = centery_1 - spacing / 2;
  this.bottom_1 = centery_1 + spacing / 2;

        let centery_2 = height + spacing;
        this.top_2 = 0;
        this.bottom_2 = height/2;
         }
        else if(x>0.65){
                //pipe down
        let centery_2 = random(spacing, height/2 - spacing);

  // Top and bottom of pipe
```

```
  this.top_2 = centery_2 - spacing/2;
  this.bottom_2 = 30

        let centery_1 = -spacing;
        this.top_1 = 0;
        this.bottom_1 = 0;
         }
         else{
                //both pipes
                  // Where is th center of the empty space
                let centery_1 = random(spacing, height/2 - spacing);

  // Top and bottom of pipe
  this.top_1 = centery_1 - spacing / 2;
  this.bottom_1 = centery_1 + spacing / 2;
  let centery_2 = random(spacing, height/2 - spacing);

  // Top and bottom of pipe
  this.top_2 = centery_2 - spacing/2;
  this.bottom_2 = 25;
        }
  // Starts at the edge
  this.x = width;
  // Width of pipe
  this.w = 80;
  // How fast
  this.speed = 6;
}

// Did this pipe hit a bird?
hits(bird) {
        if (bird.x > this.x && bird.x < this.x + this.w) {
                if((bird.y < bird.r *2.5 + this.top_2 + height/2) && (bird.y + bird.r *2.5 >
this.top_2 + height/2)){
                        return false;/*tal y como están las pipes, cuando */
                        }
                else{
  /* else */if((bird.y - bird.r) < this.top_1 || (bird.y + bird.r) > (height - this.bottom_2)){

     return true;
    }
  else if(((bird.y + bird.r) > this.bottom_1 && (bird.y - bird.r) < (height/2 + this.top_2))){
  return true;
```

```
}
        else{
                return false;
        }
                        }
        }
                }


 show() {
  stroke(255);
  fill(200);
  rect(this.x, this.bottom_1, this.w, height/2 - this.bottom_1);
        rect(this.x, 0, this.w, this.top_1);
  rect(this.x, height/2, this.w, this.top_2);
       rect(this.x, height - this.bottom_2, this.w, this.bottom_2);


 }
```

Addition of an extra kind of pipe which necessitates the addition of two new values of top and bottom. Furthermore, we added a condition which does not match the desired collision to ***avoid rapid population decayment.***

Bird.js:

```
if (closest != null) {
    // Now create the inputs to the neural network
    let inputs = [];
    // x position of closest pipe
    inputs[0] = map(closest.x, this.x, width, 0, 1);
    // top of closest pipe opening
    inputs[1] = map(closest.top_1, 0, height, 0, 1);
    // bottom of closest pipe opening
    inputs[2] = map(closest.bottom_1, 0, height/2, 0, 1);


    // bottom of closest pipe opening
    inputs[4] = map(closest.bottom_2, 0, height, 0, 1);
    // bird's y position
    // bird's y position
    inputs[5] = map(this.y, 0, height, 0, 1);
    // bird's y velocity
    inputs[3] = map(this.velocity, -5, 5, 0, 1);
```

```
    // Get the outputs from the network
    let action = this.brain.predict(inputs);
    // Decide to jump or not!
    if (action[1] > action[0]) {
      this.up();
```

We added the additional information of the modified pipes and thus increased the number of inputs in 'new Neural Network(6,...)'

Ga.js:

```
function generate(oldBirds) {
  let newBirds = [];
  for (let i = 0; i < oldBirds.length; i++) {
    // Select a bird based on fitness

          let j = Math.floor(Math.random() * poolSelection(oldBirds).length);
    newBirds[i] = poolSelection(oldBirds)[j];


  }
  return newBirds;
}
```

A generalization to generate the next generation given any poolselection function which returns an array of the "best birds".

```
let newbirdsseed = [];
  let index = 0;

  // Pick a random number between 0 and 1
  let r = random(1);

  // Keep subtracting probabilities until you get less than zero
  // Higher probabilities will be more likely to be fixed since they will
  // subtract a larger number towards zero
  while (r > 0) {
    r -= birds[index].fitness;
    // And move on to the next
    index += 1;
  }

  // Go back one
  index -= 1;
```

```
  // Make sure it's a copy!
  // (this includes mutation)
  /*return birds[index].copy();*/
          newbirdsseed[0] = birds[index].copy();
          // Start at 0
  let index_1 = 0;

  // Pick a random number between 0 and 1
  let r_1 = random(1);

  // Keep subtracting probabilities until you get less than zero
  // Higher probabilities will be more likely to be fixed since they will
  // subtract a larger number towards zero
  while (r_1 > 0) {
    r_1 -= birds[index_1].fitness;
    // And move on to the next
    index_1 += 1;
  }

  // Go back one
  index_1 -= 1;

  // Make sure it's a copy!
  // (this includes mutation)
  newbirdsseed[1] =  birds[index_1].copy();
return newbirdsseed;
}
```

This poolSelection function return the “two best birds” usually and ensures that the “best birds” are always selected, selecting just one instead of two may allow for a normal bird to continuously throw off the genetics if it keeps getting selected.

The restrictions mainly come from nn.js, since its randomization initializes the values of the weights between zero and one, and there is only one hidden layer which doesn't allow for further abstraction.

However only the latter affects both scenarios, since the gradient descent allows for new values to be achieved in the approximation. And the selection of birds in the first scenario doesn't occur through any other process aside from “natural selection” and mutation.
The birds are selected and given a chance equal to the normalized fourth power of their score, which corresponds to the amount of pipes lived through.

Results and observations:

The selection of birds and their “reproduction, mutations, and crossovers” are unaffected by the type of activation function we choose. Albeit, their reasoning as to jump or not is directly connected to their “brain”; which means that said function **does** play a role in their survival. Overall not much difference was observed between tanh and sigmoid, however tanh tended to be swifter at increasing the score. Furthermore, a specific run of ½ proved to be exceptional and immediately surpassed every other run, nonetheless said run differed greatly from every single other ½ run which performed overall worse than sigmoid. This makes sense since we are looking for boolean-like values to determine whether to jump, and it becomes difficult to find through sheer mutations decent weights which happen to work with ½ without knowing its linear properties (we lack gradient descent upon the birds). The birds also performed better with a redundancy value, a third input which wasn’t taken into account for the jumping decision, this allowed for redundancy to be dumped there instead of fed into the important outputs. (output.length=3)

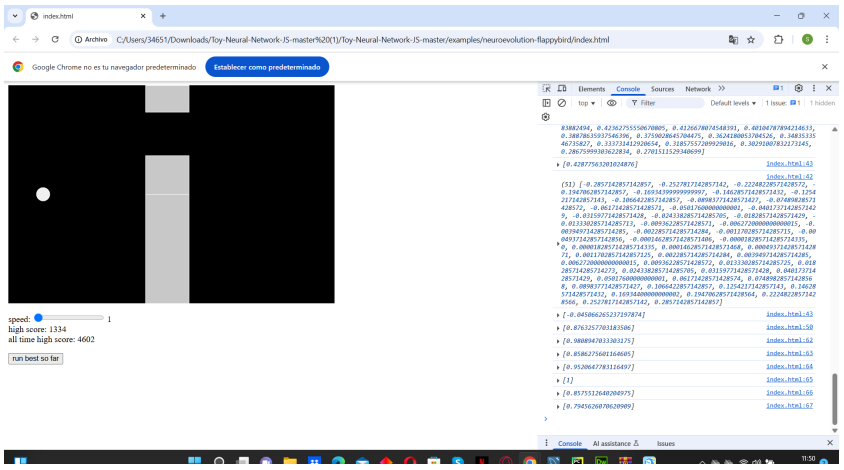


Figure 1 Sigmoid activation Function

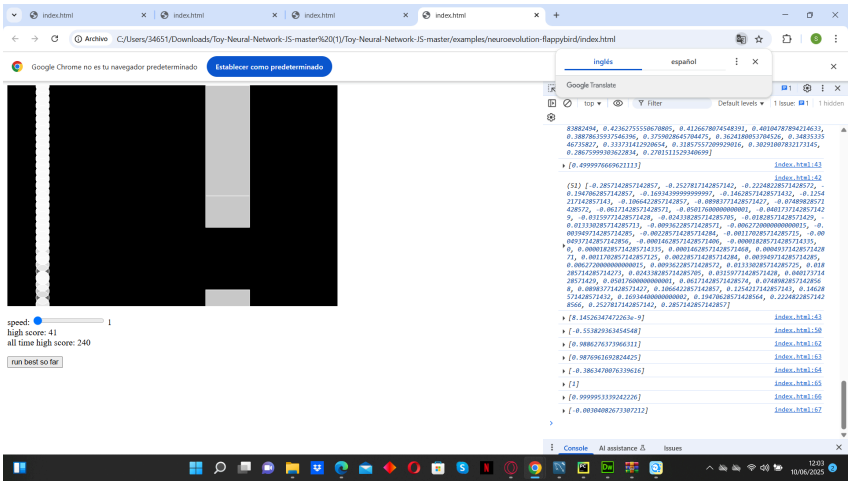
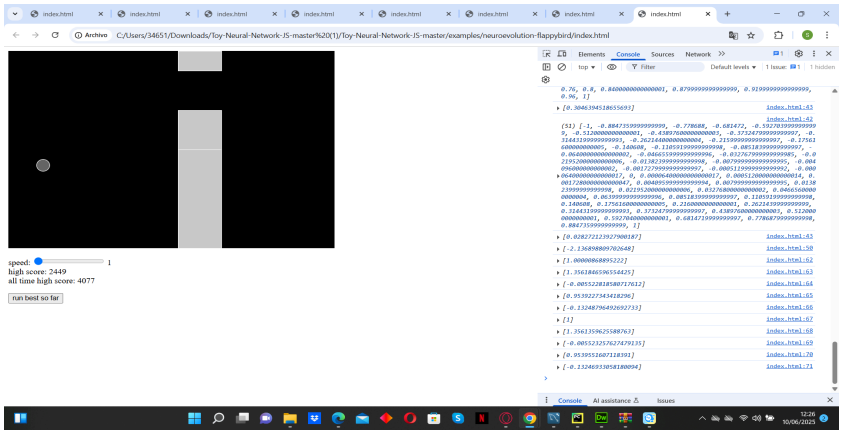


Figure 2 tanh

Finally, we observed disappointing results with the tanh and sigmoid activation functions within the approximations. This was logical since said functions erase the values of the functions after the activation, and the weights must be universal regardless of the inputs (evaluations). Thus making it really difficult for the weights to exploit the linearity of said functions to ensure the preservation of the information. The



latter function also posed a significant problem which was its inability to produce negative values, even with the gradient descent the weights refused to switch signs, thus becoming a significant problem.All this was to be expected.

Figure 3 ½ (not really the Relu function)

Hence we elected ½ as the activation function to maintain the information through the network, be able to produce negative and positive values, and to exploit its linearity. We had to choose ½ < 1 to make the gradient descent converge.

This activation function produced results consistent with the real output with the errors magnitude oscillating between e-4 and e-7. *The results can be viewed in figures* 1,2 *and* 3, *for their respective activation functions.*

As a final remark we can highlight that we must have access to or infer some additional information regarding the desired function or our training functions. The upper and lower bound must be known over the interval of ra and the testing interval, otherwise we risk having the NN's weight not converging and diverging to infinity due to large differences between initial guesses and real values, (also fixable with a lower learning rate).

For example having the same ½ activation leads to divergence at ra=2, and the predictions thus become <0:NaN>, using and coping from chrome console: [

null

]

We must lower the fraction to 1/16 to even get a prediction, while only using the interval [-1,1] to make our predictions. Nevertheless, said prediction has an error of the magnitude of e-0, so it's not as impressive.

Funnily enough, setting the x to x=>½ and y=1 allows for the weights to adapt themselves perfectly even though the gradient is not defined correctly.

Now we check the relation to the Chebyshev approximation:

Checking the weights of input→hidden→ It seems they follow a uniform distribution U(-2,2), but some weights exceed said interval. By checking the direct relationship between the inputs and outputs:

```
console.log(Matrix.multiply(nn.weights_ho,nn.weights_ih));
```

And storing said vectors in R to perform a simple analysis of average and variance, we get:

Average	Std
[1] -0.002869434	0.727783792
[1] 0.2816932	0.7741737
[1] -0.03982416	0.49519302
[1] 0.2390063	0.5217363
[1] 0.2128195	0.6283776
[1] 0.06818374	0.65974226
[1] 0.02774532	0.57478388
[1] 0.1641509	0.6387893
[1] 0.005909774	0.643660583
[1] -0.1029094	0.6362422
[1] 0.4666990	0.6722978
[1] 0.09461284	0.68682416
[1] -0.03873046	0.62228794
[1] 0.03997943	0.61146120
[1] -0.04110752	0.58441564
[1] -0.01900481	0.69553278
[1] 0.1702143	0.8500599
[1] 0.01492376	0.78261454
[1] -0.1207175	0.8032691
[1] -0.01048789	0.54183281
[1] 0.2884165	0.7536239
[1] -0.1420923	0.7543894
[1] 0.2279600	0.6475597
[1] 0.02112526	0.35768659
[1] 0.2331158	0.4390542
[1] 0.3381977	0.6150754
[1] -0.04129706	0.57233405
[1] 0.2148707	0.8405013
[1] 0.1771264	0.7233413
[1] -0.005197124	0.768793103
[1] 0.003910027	0.769717392
[1] 0.0231362	0.6444629
[1] -0.3264926	0.7209017
[1] 0.4211514	0.6505660
[1] 0.06652471	0.84256464
[1] -0.01008577	0.57830995
[1] 0.1846169	0.6712509
[1] -0.05545613	0.93229459
[1] 0.09010356	0.61155026
[1] 0.2699925	0.5463951
[1] 0.008053972	0.616565317
[1] 0.3341436	0.7309441
[1] 0.09707048	0.77205369
[1] -0.1942413	0.6297855
[1] 0.08341727	0.62378121
[1] -0.1208818	0.4392290
[1] 0.07773805	0.57117367
[1] 0.1578193	0.5108659
[1] -0.1234207	0.7374338
[1] 0.1844417	0.6993826
[1] 0.1059588	0.4385494

The following line of code alerts us when the standard deviation doesn't exceed the average for some of the 51 "direct weights" between inputs and outputs:

```
for (i in 1:51) {
  print(c(mean(y[,i]),(var(y[,i]))^(1/2)))
  if(abs(mean(y[,i])>(var(y[,i]))^(1/2))){
    print("t")
  }
}
```

Using:

```
d <- density(y[,43]) # returns the density data
plot(d)
```

for a number of arbitrary rows indicates a distribution of the weights akin to a normal distribution (with μ=0 and σ=0.65), *where a number of funt.length equations must be satisfied, reducing the degrees of freedom by the same amount. This suggests a saturation of training data, with functions independent enough, will force some approximating method by getting rid of the distribution, save for normal homoscedastic errors and noise. The plots can be seen in figure 4.*

Overall, it becomes clear that the model's training looks for the best approximation of said random variable.Were we using more random variables instead of just one, or a more extensive data training set, with more independent functions (ie the jacobi or chebyshev polynomials), this would force more equations, less degrees of freedom, and thus a better approximation with a clear structure which should resemble the best possible approximation over all the interval, with it being chebyshev, which shall be a test I'll perform soon.

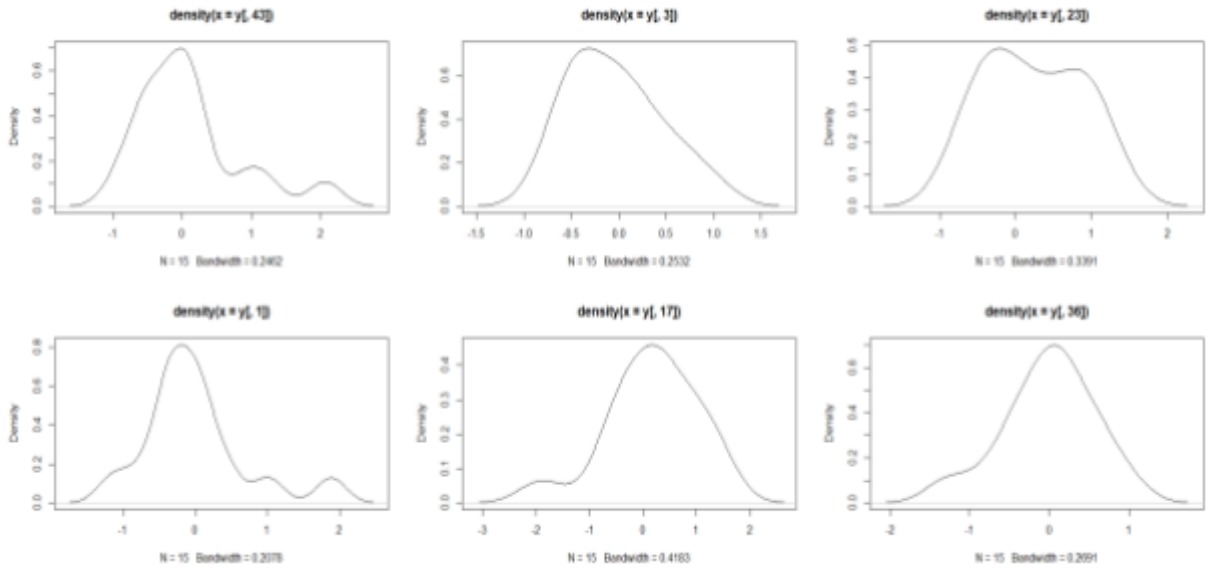


Figure 4 Compilation of smoothed plots of the distribution of weights inputs→outputs

Benefits of working with an ANN model:

While we could use The chebyshev polynomials to instantly obtain an accurate enough approximation, the benefits of working with this kind of ANN prove to be its reliance on a set of functions whose evaluations we can calculate. This means the model draws information from these functions, unlike a traditional code giving us the result. Our choosing of the functions is fundamental, since choosing functions which share similarities and properties can help the output qualitatively. Furthermore, were the dataset of initial points to be corrupted or get some points wrong, whichever is the case, the formulaic approach could fail radically, meanwhile, a more complex DNN which captures the intended design of the function we desire to approximate will shut off during activation the corrupted points if they alter dramatically the results, meaning a significant change in some points will thwart the formulaic approach but be ignored by the DNN, since the DNN will learn through the given functions, which share some properties with our goal function, to allocate low weights to said points since they alter our results significantly. It may learn to detect errors and anomalies in the inputs since significant changes do not make sense and thus are “disconnected” from the output; as long as it is initialized with this nuance in mind and the learning rate and activation function make it converge early enough for it to realize the necessity of avoiding corrupted values.
Moreover it is a more general and flexible approach, since we could change the output to be the integral over an interval (and we could try to prove the weights to be related to the gaussian quadrature),or another output whose complete mathematical expression we don’t know.Proving that relying on stochasticíty and inference is a great tool for more general problems, when the theoretical scope cannot tackle them completely.