

Neural Networks

# BEGINNERS' TUTORIAL



## INTRODUCTION

---

I will talk a little, in this introduction, about the generalities of Neural Networks (a.k.a. NNs or ANNs) and their applicability, regardless of their implementation in informatics.

So, to begin with, there are some questions that must be answered before we can proceed, questions like: What are these Artificial Neural Networks? What is Artificial Intelligence? Can a computer really think on its own?

### WHAT IS ARTIFICIAL INTELLIGENCE?

Well, you have to admit that, by default, this term makes you think of some futuristic ideas such as human-like robots or other “intelligent” devices. Indeed this idea is widely promoted by SF writings and SF movies, and... after all... there’s no one who can tell that this won’t happen in a near or not-so-near future. OK, but for now we will stick to present times and forget about the future, as for now artificial intelligence is simply another way (another approach) of solving problems, like the algorithmic approach. Confused? Let’s give an example: Let’s say that you want to solve a problem that calculates the sum of the first  $N$  odd numbers. Now, this problem can be solved by using the algorithmic way (fetching the first  $N$  odd numbers and then summing them) or it can be solved using a neural network (how? keep reading...). Of course this is just a simple and unpractical example, just to better understand the idea.

Good, but our next thought would be why do we actually need A.I. and how did it “take life”? That’s a nice question... why oh why do we need it when we already have enough on our minds... Actually as many other things it is not really necessary unless we want to use it. The fact is that there are certain problems that can’t be solved using the algorithmic approach, I’m talking here about those problems that require exponential or factorial running times. Surprised or not, artificial intelligence can solve (it’s true, with a certain approximation) many of these problems. An example here would be the well-known “travelling sales-man” problem.

For the second part of the question, how did A.I. appear, that’s easy and you probably foresaw the answer already; it “took life” thanks to some great men like Frank Rosenblatt who had the inspiration to “take a look” inside the human brain, to study the behaviour of the human thinking and to try and simulate it. Shortly speaking, that’s it, NNs are a simulated replica of the human thinking process.

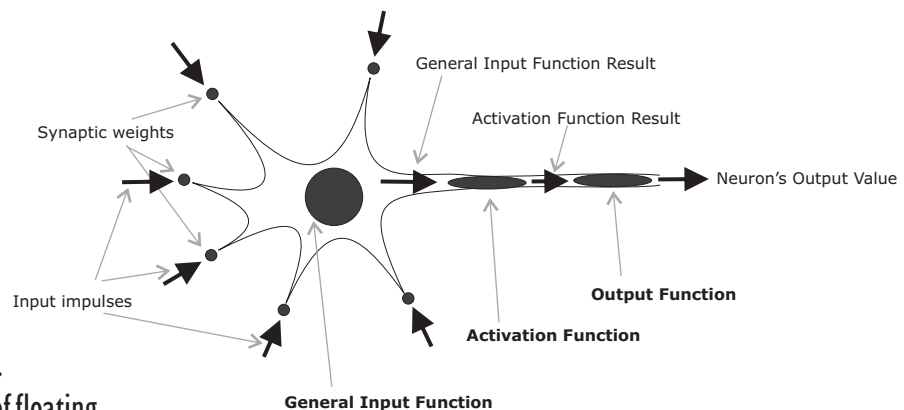
So this answers the first question, “What are Neural Networks?”, and also I would like to add here the connection between A.I. and NNs which is that Neural Nets is a branch of Artificial Intelligence (there are some others like genetic algorithms which we won’t discuss about).

OK, so let’s talk a little about the way these NNs work. Well, as I was saying they were inspired by the human thinking, so the first question here is how the human brain thinks? Hmm... through learning, of course, don’t you agree? Yeah, this is the keyword, learning. A Neural Net must learn before it can be used properly, we also call this process the training of the neural network. Good, but let’s be more specific and give a clear example: How do humans learn things? Let’s say how do we learn handwriting? Well, if you remember, we are given a set of characters, called the alphabet, and we see there how each letter should look like, and then we keep on practicing, writing each letter for some hundreds of times until we are satisfied with the result. That is exactly how NNs also learn, they are given a “training set” (e.g.: file, database etc.) which must be scanned and applied  $N$  times until we are satisfied with the learning result. This is the whole idea of Neural Networks. In the next pages we will discuss in the detail about the implementation of NNs and the learning process.

## IMPLEMENTATION - Neuron

To begin implementing neural networks on our applications we must first think where to begin from? So, again, let's think a little what is the structural and functional unit of the human brain? Indeed, it is the neuron, the biological neuron to be more specific because on the following lines we will create a simulated replica of it which we will name the artificial neuron; this is the starting point of the implementation. Good, so let's make a brief analysis of the biological neuron: we know that it receives a series of electrical impulses, through dendrites, having different intensities, then each of these impulses are modified either up or down through the neuron's synaptic weights, and finally all these values are combined in a single value which is passed further through the neuron's axon. Ok, so we kind of know what we have to do, but there are still a few pieces missing out of this puzzle, like how the neuron combines those N values to become one and what other modifications are applied to the electrical impulse as it passes through the axon. Thanks to some fine people these studies were made and here are the results: the series of entries becomes a single value through **summing**, in most cases that is, because there are some neurons that use other methods like product, minimum value or maximum value. We will name this method the **General Input Function**. So far so good. The next modification to apply is called the **Activation Function** of the neuron and its type is one of these: Hyperbolic Tangent, Sigmoidal or Linear. We will discuss each of these function a bit further. Of course, this activation function it's applied to the output value of the General Input Function. For most applications these transformations would be enough but sometimes the output of the Activation Function needs to suffer one more modification, which we'll name the **Output Function**. Its role is to transform the value of the activation function into binary information (0 and 1), according to a predefined border. You might get a bit confused after reading these lines so I'll try to make a graphical representation here for better comprehension.

Great, now the puzzle is complete, we can begin implementing a biological neuron in any programming language. How? Well, let's see. First we have to create the series of input impulses and also their corresponding synaptic weights. The best way it's by using two arrays of floating point numbers having the same size, N.

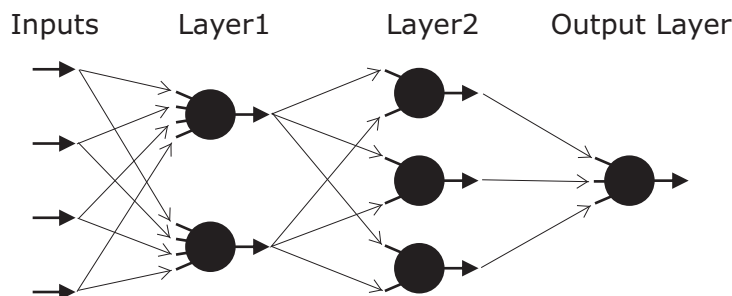


Decalring these data structures should be easy regardless the language used, something like `float in[n], sw[n];`. Now for the General Input Function (G.I.F). Let's assume we set the G.I.F. to summing the values. In pseudocode the function should look something like this: `gif_res=0; for (i=[1, n]) gif_res=gif_res+in[i]*sw[i]; return gif_res;` The activation function is even easier (we assume we set it to hyperbolic tangent): `af_res=Tanhyp(gif_res);` The last modifcator is the output function which is practically used only if we want binary output case in which it looks like this: `of_res=af_res<border?0:1;` Border is usually 0 for hyperbolic tangent activation function (because hyp. tan. returns values in the interval [-1,1] so 0 is the middle point here). This would be the implementation of the neuron, but there are still a few comments to be added. The activation function type can differ of those mentioned here, the condition is that the mathematical function used must be continous and derivable on the working interval. Ok and a second comment is still related to the activation function, that is that there are two, let's say optional, parameters that can tune up the activation function. These params are the G factor and the TETA factor. The first controls the tension of the function and the second one, TETA, controls the displacement (reported to the origin point). As I said they are optional, though they are implemented in my downloadable API. If you want to read more about these parameters you should read the API documentation.

## IMPLEMENTATION - Neural Network

Before jumping to the actual implementation let's try and define a neural network. This is not that hard since, generally speaking, a NN is simply a series of neurons connected between them. Of course there are certain rules that govern these connections, rules which we will discuss in the following lines. Also a neural network is what, again, very generally speaking, forms the human brain, but I must advise you not get any idea like trying to simulate on the PC an entire human brain because that would be practically impossible with today technologies. Explanation? Simple. Studies show that the human brain contains about a billion neurons (yeap, a billion is that number with twelve zeros in the end) each of them having about ten thousand connections in the network they form. So let's translate this in computer terms. One billion neurons, 10000 connections, this means that each of the 1 billion neurons must have 10000 input values and another 10000 synaptic weights values, and also three function definitions. Assuming we use 32bit floats, each neuron will require  $32 * 10000 * 2$  bits to be stored, that is 640000 bits  $\sim = 0.6\text{MB}$ . Now multiplying this with that 1 billion would result in some 572205 TB (yes, terrabytes) that should be stored in the RAM Memory, not to mention a CPU required to process such amount of data in reasonable time. Ok, so there you have it, now about that NN implementation.

First thing to talk about is how the neurons should be organized to form a proper neural network. The best way seems to be the layer organization. We distinguish here three types of layers: the input layer, the hidden layers and the output layer. The input layer is formed by simple values (floats), the other layers are arrays of neurons. The output layer is the last layer on the network, layer which gives the output of the neural network. Good, now how we connect these neurons in order to form a neural network? Well, the connection rule is rather simple: **each** neuron on layer J is connected to **every** neuron on layer J-1; more specific, each output of each neuron on layer J-1 goes to each input of each neuron on layer J. This also means that each neuron on layer J must have as much inputs and synapses as the number of neurons on the J-1 layer. Again, I sense that at this point you could be a little confused so here goes a graphic representation of a neural network:



Hoping you managed to understand the structure of a neural network, here is how it should be implemented. First, the definition of the input layer, an array of floats: `float in_1[N]`; then the structure of the neural network (number of layers and neurons on each layer): `neuron nn_model[no_of_layers][ ]`; `for(layer=1,no_of_layer)` `nn_model[layer] = neuron[no_of_neurons_on_layer]`; Also there should be implemented a series of functions that use the neural network, functions that calculate and return the output of the network, functions that set or reset the synaptic weights, functions that change neurons' parameters for each layer etc. I won't discuss about their implementation here, but if you are interested read the API Reference as they are all implemented there.

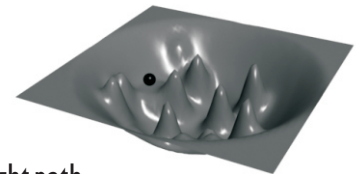
## TRAINING A NEURAL NETWORK

The most common method for training a NN is the backpropagation rule. As I was saying in the Introduction section of this document, training a neural network requires a training set (could be from files, databases etc.). This training set is composed of a series of inputs and their corresponding desired outputs, that is the output the network should give if it is well-trained. The training process goes like this: the inputs from the training file are set to the NN's input layer then the output of the network is calculated and compared with the desired output corresponding to these inputs. The absolute value of the difference between the actual output and the desired one is called the error of the network at that point, error which is backpropagated (this is where the name comes from) to each previous layer in order to recalculate the synaptic weight value of each neuron in such a manner that the error will be smaller. When the network processed an entire set of training values we say that an **epoch** has been performed. For a network to be well-trained the training data must be processed several times, so the learning process must perform several epochs to get well-trained. Even so, there are chances that the learning process will not run as smoothly as expected and that it will get stuck in some point of local minimum on the error's graph, instead of the global minimum point which represents the best level on learning a neural network can achieve. Here is a graphical representation of the error function for a better comprehension.



The image in the left shows how an optimal error graph should look like. The ball represents the level of learning achieved by the network. You can see that it doesn't matter from which point the learning process starts because it will always reach the global minimum of the error graph, it has no "obstacles". But, of course, this situation can't be achieved, a real error graph will look something like this (picture on the right).

You can see in this image that the learning process (the black ball) got stucked in that local minimum point and can't go further. In this case the learning process should be restarted so that the next starting point will be on a path which could reach the global minimum. This is an example of learning failure, another possible situation is when learning is on the right path but it skips the global minimum, going up again.



The learning process has a series of settable parameters: the maximum number of epochs to be performed after which the learning process will end, the timeout of the learning process that is the time interval the learning should run and of course the acceptable error, which represents the level of error we want our network to achieve. The implementation of the backpropagation algorithm is contained by the API and also documentation on it can be found in the API's Documentation.

In the end I would like to talk a bit about data normalization. Well, data normalization is the process through which we reduce an interval of values (e.g.: reducing the interval  $[0, 100]$  to  $[-1, 1]$ ). It is absolutely necessary for neural networks because as you could have realized the activation functions will return values only in either  $[-1, 1]$  or  $[0, 1]$ , so of course, a neural network will only return values in one of these intervals, but using data normalization its very easy to encode and than decode values in order to obtain a real and usable numeric result. Again, to clear things out I will give an example: Let's assume you want to train a neural network to learn the square root function for usage with unsigned integers. We know that the data types can take values in the interval  $[0, 65535]$ , but as I was saying, a NN can work with values between  $[-1, 1]$ , thus we will normalize input values to fit the  $[-1, 1]$  interval, train the network with these normalized values and then, when we use the trained network we will simply de-normalize the output of the NN. The API has implemented data normalization methods which make it extremely easy for you to encode and decode data.

Well, what can I say in the end... just hoping that this document cleared things out for you, at least a bit.

Dan Hintea.