www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 6 June 2015, Page No. 12692-12697

Implementing Actual Working Like Intuition Mechanism In Artificial Neural Network

Mr.Vivek B. Patil
Research Student
Vivekpatil010@Gmail.Com

Abstract:
One fascinating thing about artificial neural networks is that, they are mainly inspired by the human brain. This doesn't mean that Artificial Neural Networks are *exact* simulations of the biological neural networks inside our brain - because the actual working of human brain is still a mystery. The concept of artificial neural networks emerged in its present form our very limited understanding about our own brain *("I know that I know nothing").*
Keywords: Neuron, input/output/Hidden Layer, unit, Network.

Introduction:

Before understanding how neurons and neural networks actually work, let us revisit the structure of a neural network. As I mentioned earlier, a neural network consists of several layers, and each layer has a number of neurons in it. Neuron is one layer is connected to multiple or all neurons in the next layer. Input is fed to the neurons in input layer, and output is obtained from the neurons in the last layer.

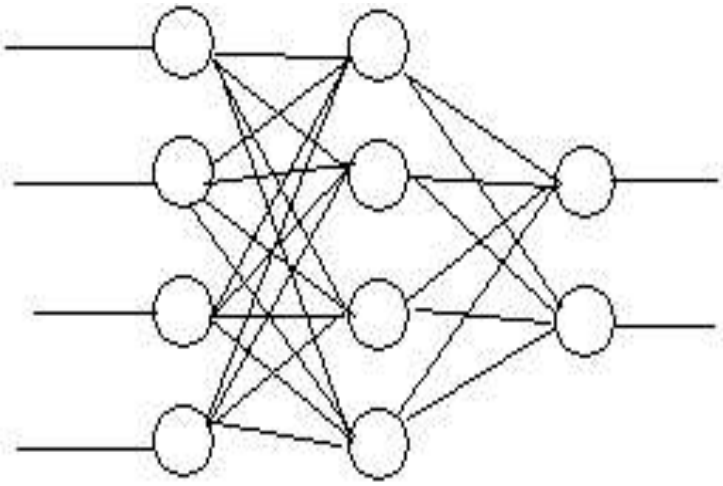
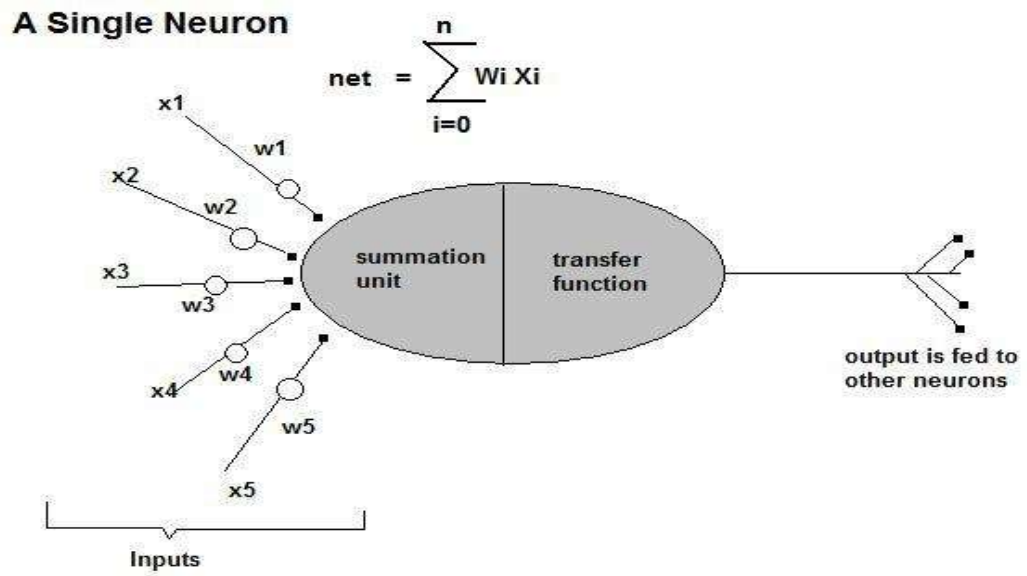


Fig: A Fully Connected 4-4-2 neural network with 4 neurons in input layer, 4 neurons in hidden layer and 2 neurons in output layer.

For training a neural network, first provide a set of inputs and outputs. For example, if need a neural network to detect fractures from an X-Ray of a born, first train the network with a number of samples. provide an X-Ray, along with the information that whether that particular X-Ray has a fracture or not. After training the network a number of times with a number of samples like this (probably thousands of samples), it is assumed that the neural network can 'detect' whether a given X-Ray indicates a fracture in the born (This is just an example)., in this article, discuss the theory behind network learning.

Artificial Neurons

Now, let us have a look at the model of an artificial neuron.



An artificial neuron consists of various inputs, much like the biological neuron. Instead of Soma and Axon, we have a summation unit and a transfer function unit. The output of one neuron can be given as input to multiple neurons.

Summation Unit

- When inputs are fed to the neuron, the summation unit will initially find the *net-value*. For finding the Net Value, the product of each input value and corresponding connection weight is calculated.
- i.e., input value x (i) of each input to the neuron is multiplied with the associated connection weight w(i). In simplest case, these products are summed and fed to the transfer function. See the pseudo code below, it is simpler to understand.
- Also, a neuron has a bias value, which affects the net value. A bias of a neuron is set to a random value, when the network is initialized. We will change the connection weights and bias of all neurons in the network (other than neurons in the input layer), during training phase.
- I.e., if **x** is the input, and **w** is the associated weight, then pseudo code for net value calculation is as follows

```
netValue=0

for i=0 to neuron.inputs.count-1
   netValue=netValue + x(i) * w(i)
next

netValue=netValue + Bias
```

Transfer Function

Transfer function is a simple function, that uses the net value to generate an output. This output is then propagated to the neurons in the next layer. We can use various types of transfer functions as shown below.

Hard Limit Transfer Function: For example, a simple hard limit function will output 1 if net value is greater than 0.5, and will output 0 if the net value is lesser than 0.5 - as shown.

Hide Copy Code

```
if (netValue<0.5)
     output = 0
else
     output = 1
```

Sigmoid Transfer Function: Another type of transfer function is a sigmoid transfer function. A sigmoid transfer function will take a net value as input and produce an output between 0 and 1 as shown.

```
output = 1 / (1 + Exp(-netValue))
```

The implementation of summation unit and transfer function unit may vary in different networks.

This, a neural network is constructed from such basic models, called neurons, arranged together in layers, and connected to each other as explained earlier. Now let us see how all these neurons work together, inside a neural network.

1 A Neural Network Actually 'Works'

Working with a neural network includes

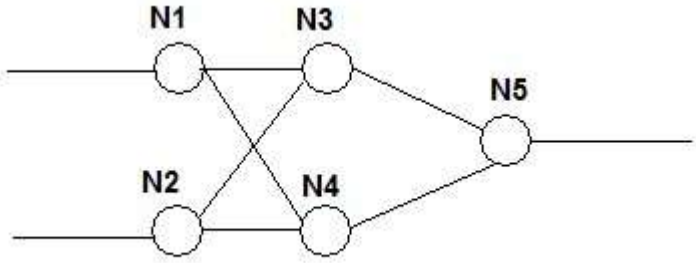
- **Training the network - by providing inputs and corresponding outputs.**
  - In this phase, we train a neural network with samples to perform a particular task.
- **Running the network - by providing the input to obtain the output.**
  - In this phase, we will provide an input to the network, and obtain the output. The output may not be accurate always. Generally speaking, the accuracy of the output during running phase depends a lot on the samples we provided during the training phase, and the number of times we trained the network.

1.1. Training Phase

This section explains how the training takes place, in a back ward propagation neural network. In a backward propagation neural network, there are several layers, and each neuron in each layer is connected to all neurons in the next layer. For each connection, a random weight is assigned when the network is initialized. Also, a random bias value is assigned to each neuron during initialization.

Training is the process of adjusting the connection weights and bias of all neurons in the network (other than neurons in the input layer), to enable the network to produce expected output for all input sets.

Now, let us see how the training actually happens. Consider a small 2-2-1 network. Now, we are going to train this network with AND truth table. As know, AND truth table is



AND TRUTH TABLE		
A	**B**	**Output**
0	0	0
0	1	0
1	0	0
1	1	1

Fig: A 2-2-1 Neural Network and Truth Table Of AND

In the above network, N1 and N2 are neurons in input layer, N3 and N4 are neurons in hidden layer, and N5 is the neuron in output layer. The inputs are fed to N1 and N2. Each neuron in each layer is connected to all neurons in next layer. We call the above network a 2-2-1 network, based on the number of neurons in each layer.

The above diagram will be used to illustrate the process of training.

First, let us see how we train our 2-2-1 network, the first condition in the truth table, i.e, when A=0, B=0 then output=0.

Step 1 - Feeding the Inputs

Initially, we will feed the inputs to the neural network. This is done by simply setting the output of neurons in Layer 1, as the input values we need to feed. I.e., as per the above example, our inputs are 0, 0 and output is 0. We will set the output of Neuron N1 as 0, and the output of N2 is set to 0. Have a look at this pseudo code, and it will make things clear. An input is the input array. The number of elements in Input array should match the number of neurons in input layer.

```
i = 0
For Each neuron In InputLayer
    someNeuron.OutputValue = Inputs(i)
    i = i + 1
Next
```

Step 2 - Finding the output of the network

We have already seen how we calculate the output of a single neuron. As per our above example, the output of neurons N1 and N2 will act as the inputs of N3 and N4.

Finding the output of neural network involves, calculating the outputs of all hidden layers and output layer. As we discussed earlier, a neural network can have a number of hidden layers.

```
'Find output of all neurons in all hidden layers
 For each layer in HiddenLayers
    For Each neuron In layer.Neurons
        neuron.UpdateOutput()
    Next
 Next

 'Find output of all neurons in output layer
 For Each neuron In OutputLayer.Neurons
        neuron.UpdateOutput()
 Next
```

UpdateOutput() function of a single neuron works exactly as we discussed earlier. First, net value is calculated by the summation unit, and then it is provided to a transfer function to obtain the output of the neuron. Pseudo code is again shown below.

Summation Unit works like this:

```
Dim netValue As Single = bias

For Each InputNeuron connected to ThisNeuron
    netValue = netValue + (Weight Associated With InputNeuron * _
                           Output of InputNeuron)
Next
```

I.e, as per our above example, let us calculate the net value of neuron N3. We know that N1 and N2 are connected to N3.

- Net Value Of N3 = N3.Bias + (N1.Output * Weight Of Connection From N1 to N3) + (N2.Output * Weight Of Connection From N2 to N3)

Similarly, to calculate the net value of N4,

- Net Value Of N4 = N4.Bias + (N1.Output * Weight Of Connection From N1 to N4) + (N2.Output * Weight Of Connection From N2 to N4)

Activation Unit Or Transfer Unit:

Now, let us see how we are generating the output, using Transfer unit. Here, we are using the sigmoid transfer function. This is exactly as we discussed earlier.

```
Output of Neuron = 1 / (1 + Exp( - NetValue )
```

Now, the output of N3 and N4 will be passed to each neuron in the next layer as inputs. This process of propagating the output of one layer as the input to the next layer is called forward propagation part in the training phase.

Thus, after step 2, we just found the output of each neuron in each layer - starting from the first hidden layer to the output layer. The output of the network is simply the output of all neurons in the output layer.

Step 3 - Calculating the Error or Delta

In this step, we will calculate the error of the network. Error or Delta can be stated as the difference between the expected output and the obtained output. For example, when we find the output value of the network for the first time, most probably the output will be wrong. We need to get 0 as the output for inputs A=0 and B=0. But the output may be, some other value like 0.55, based on the random values assigned to the bias and connection weights of each neuron.

Now let us see, how we can calculate the error. Let us see how to calculate the error or delta of each neuron in all the layers.

- First we will calculate the error or delta of each neuron in the output layer.
- The delta value thus calculated will be used to calculate the error or delta of neurons in the previous layer (i.e, the last hidden layer)
- The delta value of all neurons in the last hidden layer is used to calculate the error or delta of all neurons in the previous layer (i.e, second last hidden layer)
- This process is continued, till we reach the first hidden layer (delta of input layer is not calculated).

Please note one interesting point. In Step 2, we are propagating values forward - starting from the first hidden layer to the output layer, for finding the output. In Step 3, we are starting from the output layer, and propagating the error values backward - and hence, this neural network is called as a Backward Propagation neural network.
Time to see how things actually work. The general equation for finding the delta of a neuron is

```
Neuron.Delta = Neuron.Output * (1 - Neuron.Output) * ErrorFactor
```

Now, let us see how the error factor is calculated for each neuron. The Error Factor of neurons in output layer can be calculated directly (since we know the expected output of each neuron in output layer).
For a neuron in output layer,
Hide Copy Code

```
ErrorFactor Of An Output Layer Neuron = _
           ExpectedOutput - Neuron's Actual Output
```

i.e., with respect to our above example, if the output of N5 is 0.5 and the expected output is 0, then error factor = 0 - 0.5 = - 0.5

For a neuron in hidden layer, error factor calculation is somewhat different. To calculate the error factor of a neuron in hidden layer,

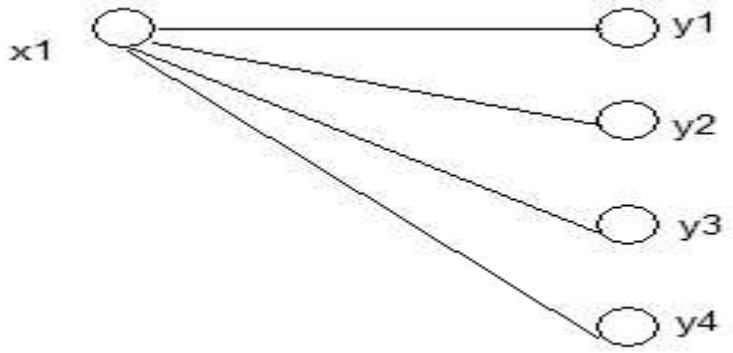
- First the delta of each neuron to which this neuron is connected is multiplied with the weight of this connection
- These products are summed up together to obtain the error factor of a hidden layer neuron

Simply speaking, a neuron in a hidden layer is using the delta of all connected neurons in next layer, along with the corresponding connection weights, to find the error factor. This is because, we don't have any direct parameters for calculating the error of neurons in the hidden layer (as we did in the output layer neurons).

```
'Calculating the error factor of a neuron in a hidden layer

 For Each Neuron N to which ThisNeuron Is Connected
   'Sum up all the delta * weight
   errorFactor = errorFactor + (N.DeltaValue * _
                 Weight Of Connection From ThisNeuron To N)
 Next
```

To illustrate this, consider a neuron x1 (ThisNeuron), which is a hidden layer neuron. X1 is connected to neurons y1, y2, y3 and y4 - and these are neurons in next layer.



i.e., to make things simple,

- Error Factor of X1 = (*Y1.Delta * Weight Of Connection From X1 To Y1*) + (*Y2.Delta * Weight Of Connection From X1 To Y2*) + (*Y3.Delta * Weight Of Connection From X1 To Y3*) + (*Y4.Delta * Weight Of Connection From X4 To Y4*)

Now, as we discussed earlier, the Delta of a X1 can be calculated as,

- X1.Delta = X1.Output * (1 - X1.Output) * ErrorFactor Of X1

Thus, after finishing step 3, we have the Delta of all neurons.

Step 4 - Adjusting the Weights and Bias

After calculating the delta of all neurons in all layers, we should correct the weights and bias with respect to the error or delta, to produce a more accurate output next time. Connection Weights and Bias, together are called free parameters. Remember that a neuron should update more than one number of weights - because, as we already discussed, there is a weight associated with each connection to a neuron.
See the pseudo code for updating the free parameters of all neurons in all layers

```
'Update free parameters of all neurons in hidden layer
 For each layer in HiddenLayers
    For Each neuron In layer.Neurons
        neuron.UpdateFreeParams()
    Next
 Next

 'Update free parameters of all neurons in output layer
 For Each neuron In OutputLayer.Neurons
        neuron.UpdateFreeParams()
 Next
```

UpdateFreeParams() function simply does two things.

- Find the new bias of a neuron, based on the delta we calculated above
- Update the connection weights based on the delta we calculated above

Finding the new bias value of a neuron is pretty simple. See the pseudo code. If Learning Rate is a constant (for e.g, Learning Rate=0.5)

```
New Bias Value = Old Bias Value + _
                LEARNING_RATE * 1 * Delta
```

Now let us see how to update the connection weights. The new weight associated with an input neuron can be calculated as shown below.

Hide Copy Code

```
New Weight  = Old Weight +  LEARNING_RATE * 1 * Output Of InputNeuron * Delta
```

As a neuron can have more than one input, the above step should be performed for all input neurons connected to this neuron.

I.e,

```
For Each InputNeuron N connected to ThisNeuron
    New Weight of N = Old Weight of N + _
                      LEARNING_RATE * 1 * N.Output * ThisNeuron.Delta
Next
```

Now, after step 4, we have a better network. This process is repeated for all other entries in the AND truth table - for probably more than thousand number of times, to train the network 'well'.

2.2. Running the Network

Running the network involves,

- Providing the inputs to the network exactly as described earlier in Step 1 above
- Calculating the outputs as explained in Step 2 above

However, it is important to note that the network should be trained with sufficient samples (and sufficient number of times), to obtain desired results. Anyway, it is almost impossible to say that the output of a neural network will be 100% accurate for any input.