Artificial Neural Networks

A simplified mathematical notation and its implementation

Elly Nkya

4/14/2009

Today, there exists numerous literature that describe the artificial neural network. However, the notation used is usually difficult for beginners. In this document, a recurrence notation is proposed that fully describes an artificial neural networks roll-up and back-propagation algorithms. Finally, an implementation of an artificial neural network is presented.

Table of Contents

Chapter 1. Artificial Neural Networks

An artificial neural network consists of neurons (nodes) that are interconnected by directed edges that carry a **weight**. The neurons are grouped in layers. Typically each successive layer of neurons is connected by edges from the previous layer. The first layer is commonly known as the **input layer**, the last layer is known as the **output layer**, whilst the intermediate layers are known as the **hidden layers**. Except for the output layer, a special neuron known as the **bias** is contained in each layer. These bias neurons output a constant value usually 1 or -1.

Neural networks form a mathematical mapping of the form

$$f(\bar{x}) \rightarrow \bar{y}$$

where,

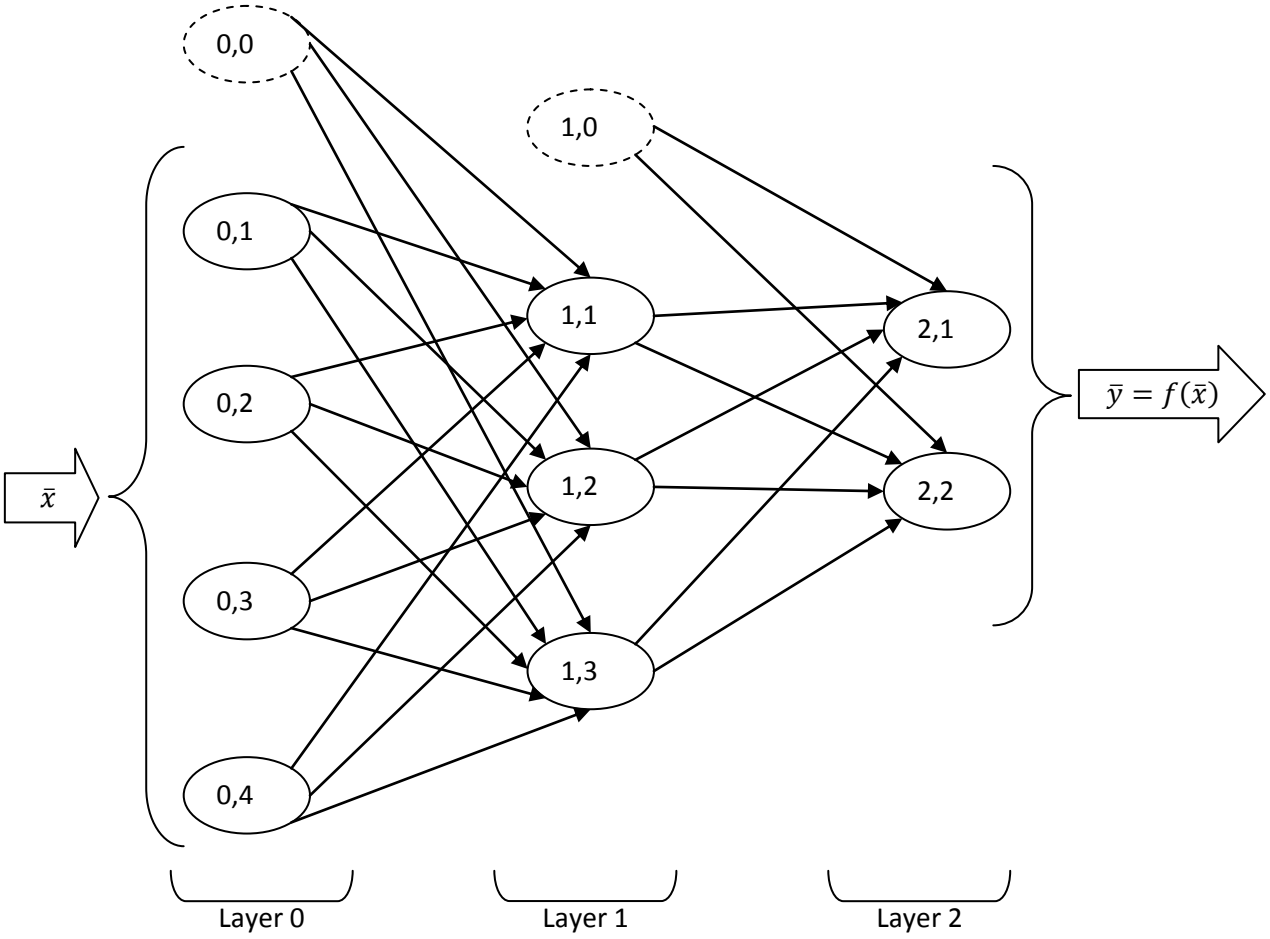
$\bar{x} \equiv (x_1, x_2, \dots, x_N)^T$ are the input signals vector accepted through the input layer which contains N nodes, each node accepts a value x_i and,

$\bar{y} \equiv (y_1, y_2, \dots, y_m)^T$ are output signals that are output through the output layer containing M nodes, each node represent a value y_j.

The mapping f, is performed by rippling the inputs through the network using the directed edges and strengthening them by their weights, until they arrive at the output nodes. Each neuron performs an **activation** of it inputs before it outputs a value. This mapping to the output layer is commonly known as a **roll-up**.

Before a neural network can perform a correct mapping of output, it is essential that the correct weights are computed on each of its directed edges. This achieved by training using a method commonly known as **back-propagation**.

The example figure below shows the essential features of an artificial neural network.



The diagram demonstrates the essential features of a neural network.
The oval shapes represent the neurons. These are grouped in layer 0, 1, and 2. Hence the network above is called a 2-layer neural network.
Layer 0 accepts the input $\bar{x}$. This input is rippled through the network by way of the directed edges represented as arrows. These edges carry weight. Values are eventually output on layer 2 as $\bar{y}$.
The dotted ovals represent the bias neurons.
Note also that the neuron are number for reference purposes as:
Layer 0 (0,0), (0,1),(0,2),(0,3),(0,4)
Layer 1 (1,0), (1,1), (1,2), (1,3)
Layer 2 (2,1), (2,2)
Hence the weights of edges can also be referenced. For example weight from neuron (0,1) to neuron (1,3) is expressed as weight([0,1],[1,3]).

Chapter 2. Roll-up of an Artificial Neural Network

Roll-up, the actual functional mapping of the input signal to produce an output signal, is the central feature of a neural network. In this section we will show how this can be expressed using a piece-wise and recurrence relation.

Suppose we concentrate on the value of what each neural node is supposed to output. We will denote the output of any node as, $\text{activate}(\bar{x}, l, i)$, where:

- $\bar{x}$, is the input vector that is accepted at the input layer.
- l, is the layer.
- i, is the node position in a layer.

We therefore observe there are three special cases that can describe the output value of any node in an artificial neural network.

1. Output of a Bias Node
   At the Bias Node the output is always a constant value. Hence can be expressed as,

$$activate(\bar{x}, l, 0) = -1$$

2. Output of an Input Node
   At the Input Node, the output is the value of the input vector itself. Hence this can expressed as,

$$activate(\bar{x}, 0, i) = x[i]$$

3. Output of all other Nodes
   All other nodes rely on the interconnection of the immediate preceding layer nodes that are connected to them by way of the directed edges. The output of these nodes, can be expressed as an activation by a function g, whose input is the sum of the product between the weight of edge directed into it and the signal of the activation of the neural node these weighted edges originate from. In a more mathematical expression, this can be expressed in recurrence form as follows:

$$activate(\bar{x}, l, i) = g\left(\sum_{j=0}^{n} activate(\bar{x}, l-1, j) w([l-1, j], [l, i])\right)$$

All three cases can now be combined using a piece-wise function and expressed as follows:

$$activate(\bar{x}, l, i) = \begin{cases} g\left(\sum_{j=0}^{n} activate(\bar{x}, l-1, j) w([l-1, j], [l, i])\right) & i, l > 0 \\ -1 & i = 0 \\ x[i] & l = 0 \end{cases}$$

Chapter 3. Back-propagation of an Artificial Neural Network

As mentioned in the first chapter, an Artificial Neural Network can be made to perform a mathematical mapping of the form,

$$f(\bar{x}) \rightarrow \bar{y}$$

Of which $\bar{x}$, is the input vector and $\bar{y}$, is the desired output.

However for it to produces correct outputs, given any set of inputs, Artificial Neural Networks must be trained. Training involves adjusting the weights of the directed edges until the network performs mapping accurately even for a majority input it has not yet encountered.

One commonly used tactic applied to adjust the weights, is to perform what is known as a back-propagation of the network. That is, given a known pair of desired input-output values $\{\bar{x}, \bar{y}\}$, weights are adjusted, starting with those linking the output layer, and moving towards the direction of those linking the input layer.

Each weight is then updated using the formula:

$$w([l-1,j],[l,i]) = w([l-1,j],[l,i]) + \alpha \times activate(\bar{x}, l-1, j)\Delta(\bar{x}, l, i)$$

Whereby for the weights connected to the outer lay

$$\Delta(\bar{x}, l, i) = g'\big(activate(\bar{x}, l, i)\big)\big(y[i] - activate(\bar{x}, l, i)\big)$$

And for all other weights,

$$\Delta(\bar{x}, l, i) = g'\big(activate(\bar{x}, l, i)\big)\sum_{j=0}^{n}\Delta(\bar{x}, l+1, j)w([l,i],[l+1,j])$$

The two preceding formulas can then be combined using the following piece-wise function as:

$$\Delta(\bar{x}, l, i) = \begin{cases} g'\big(activate(\bar{x}, l, i)\big)\big(y[i] - activate(\bar{x}, l, i)\big) & l \text{ is an outer layer} \\ g'\big(activate(\bar{x}, l, i)\big)\sum_{j=0}^{n}\Delta(\bar{x}, l+1, j)w([l,i],[l+1,j]) & l \text{ is a hidden layer} \end{cases}$$

NOTE: g' is the gradient function of the function g

Chapter 4. Implementation of an Artificial Neural Network

In this section, we will present an implementation of an Artificial Neural Network. Along with it, implementation of the networks, Roll-up and Back-propagation accompany it. The implementation made use of the descriptions made in the earlier chapters.

With this implementation you can perform the following

- Instantiation of the Neural Network class and initialization
  Example:
  ```
  NeuralNetwork neuralNetwork = new NeuralNetwork("11,9,4");
  ```
  Using the example above, a 2-layer Neural Network is created with
  - 11 input nodes in the input layer,
  - 9 node in the hidden layer and,
  - 4 output nodes.
- Roll-up of the network
  Example:
  ```
  Example inputVector = new Example(11);
  inputVector.Attributes[1] = ...;
  inputVector.Attributes[2] = ...;
  …
  inputVector.Attributes[11] = ...;

  List<Example> rollupResult = neuralNetwork.RollupNetwork(eg);
  ```
- Update the network using Back-propagation
  Example:
  ```
  Double outputVector, sqOfWeightChange;
  …
  Example outputVector = new Example(4);

  outputVector.Attributes[1] = ...;
  outputVector.Attributes[2] = ...;
  …
  outputVector.Attributes[4] = ...;

  neuralNetworkOffLine.BackPropagateNetwork(
    inputVector, outputVector, ref error, ref sqOfWeightChange);
  ```

Class NeuralNetwork

```
class NeuralNetwork
{
  public List<List<NeuralNetworkNode>> network =
    new List<List<NeuralNetworkNode>>();
  private static Double temperature = 2;

  public static void SetTemperature(double newTemperature)
  {
    temperature = newTemperature;
  }

  public NeuralNetwork(String layerNodes)
  {
    String[] layerNodesArray = layerNodes.Split(',');
    for (Int32 layer = 1; layer < layerNodesArray.Length; layer++)
    {
      List<NeuralNetworkNode> layerNetwork =
        new List<NeuralNetworkNode>();
      for (int numOfWeights = 0;
           numOfWeights <= Convert.ToInt32(layerNodesArray[layer]);
           numOfWeights++)
        layerNetwork.Add(
          new NeuralNetworkNode((numOfWeights == 0 ? 0 :
                                 Convert.ToInt32(
                                   layerNodesArray[layer - 1])),
                                1.0,
                                layer == layerNodesArray.Length - 1));

      network.Add(layerNetwork);
    }
  }

  public List<Example> RollupNetwork(Example eg)
  {
    Int32 outerlayer = network.Count - 1;
    Int32 nodesInOuterlayer = network[outerlayer].Count;

    List<Example> nodeActivationResults = new List<Example>();
    nodeActivationResults.Add(new Example(eg.attributes.Length - 1));
    for (Int32 i = 0; i < network.Count; i++)
      nodeActivationResults.Add(new Example(network[i].Count - 1));

    for (Int32 i = 1; i < nodesInOuterlayer; i++)
    {
      ActivateNetworkNode(ref nodeActivationResults, eg,
                          outerlayer + 1, i);
    }
    return nodeActivationResults;
  }

  public void BackPropagateNetwork(
    Example eg, Example expectedResult,
```

```
    ref double totalError,
    ref double newSqOfWeightChange)
  {
    List<Example> rollupResult = RollupNetwork(eg);
    List<Example> networkErrorResult =
      ComputeNetworkError(ref totalError,
        rollupResult, expectedResult);

    UpdateNetwork(rollupResult, networkErrorResult,
      ref newSqOfWeightChange);
  }

  private Double ActivateNetworkNode(
    ref List<Example> nodeActivationResults,
    Example eg, Int32 layer, Int32 node)
  {
    if (nodeActivationResults[layer].isComputed[node])
      return nodeActivationResults[layer].attributes[node];

    nodeActivationResults[layer].attributes[node] =
      ComputeActivation(ref nodeActivationResults, eg, layer, node);
    nodeActivationResults[layer].isComputed[node] = true;
    return nodeActivationResults[layer].attributes[node];
  }

  private Double ComputeActivation(
    ref List<Example> nodeActivationResults,
    Example eg, Int32 layer, Int32 node)
  {
    if (node == 0)
      return -1;
    if (layer == 0)
      return eg.attributes[node];

    Double inputValue = 0;

    for (Int32 j = 0;
         j < network[layer - 1][node].weights.Length;
         j++)
    {
      inputValue +=
        ActivateNetworkNode(
          ref nodeActivationResults, eg, layer - 1, j) *
        network[layer - 1][node].getWeight(j);
    }
    return NeuralNetworkNode.ComputeTagentHyperbolicActivation(
      inputValue);
  }

  public List<Example> ComputeNetworkError(
    ref Double totalError,
    List<Example> actualResult,
    Example expectedResult)
  {
    Int32 firstHiddenLayer = 0;
    Int32 nodesInFirstHiddenlayer = network[firstHiddenLayer].Count;

    List<Example> errorLayerResultValues = new List<Example>();
    for (Int32 i = 0; i < network.Count; i++)
```

```
      errorLayerResultValues.Add(new Example(network[i].Count - 1));

    // the outer level does not have a bias node. So start with 1.
    for (Int32 i = 1; i < nodesInFirstHiddenlayer; i++)
    {
      ComputeNetworkNodeErrors(
        ref totalError,
        ref errorLayerResultValues, actualResult, expectedResult,
        firstHiddenLayer + 1, i);
    }
    return errorLayerResultValues;
  }

  private Double ComputeNetworkNodeErrors(
    ref Double totalError,
    ref List<Example> errLayerResultValues,
    List<Example> actualResult,
    Example expectedResult,
    Int32 layer, Int32 node)
  {
    if (errLayerResultValues[layer - 1].isComputed[node])
      return errLayerResultValues[layer - 1].attributes[node];
    return ComputeErrorSum(
      ref totalError, ref errLayerResultValues,
      actualResult, expectedResult, layer, node);
  }

  private Double ComputeErrorSum(
    ref Double totalError,
    ref List<Example> errLayerResultValues,
    List<Example> actualResult, Example expectedResult,
    Int32 layer, Int32 node)
  {
    Double err = 0.0;
    Boolean isOuterLayer = layer == network.Count;

    if (isOuterLayer)
    {
      err = expectedResult.attributes[node] -
        actualResult[layer].attributes[node];
      totalError += err * err;
    }
    else
    {
      for (Int32 j = 1; j < network[layer].Count; j++)
      {
        err +=
          ComputeNetworkNodeErrors(
            ref totalError, ref errLayerResultValues,
            actualResult, expectedResult, layer + 1, j) *
          network[layer][j].getWeight(node);
      }
    }

    errLayerResultValues[layer - 1].attributes[node] =
      Gradient(actualResult[layer].attributes[node]) * err;
    errLayerResultValues[layer - 1].isComputed[node] = true;
    return errLayerResultValues[layer - 1].attributes[node];
  }
```

Class Example

```
class Example
{
  public double[] attributes;
  public double likelyHood = 0;
  public Boolean[] isComputed;

  public Example(int noOfAttributes)
  {
    attributes = new double[noOfAttributes + 1];
    isComputed = new bool[noOfAttributes + 1];
    attributes[0] = -1.0;
  }

  public double[] GetAttributes()
  {
    return attributes;
  }

  public int numOfNodes()
  {
    return attributes.Length - 1;
  }
}
```

Class NeuralNetworkNode

```
class NeuralNetworkNode
{
  public double[] weights;
  Double bias = 0;
  private static Random randomWeight = new Random();

  public NeuralNetworkNode(
    int noOfweights, double bias, bool initializeWeights)
  {
    weights = new double[noOfweights + 1];
    setBias(bias);
    if (initializeWeights)
      for (int i = 1; i <= noOfweights; i++)
      {
        weights[i] = randomWeight.NextDouble() - 0.5;
      }
  }

  public Double getBias()
  {
    return bias;
  }

  public void setBias(Double newValue)
  {
    bias = newValue;
    weights[0] = newValue;
  }
  public Double getWeight(Int32 j)
  {
    return weights[j];
  }

  public void setWeight(int j, Double newValue)
  {
    weights[j] = newValue;
  }

  public static Double ComputeTagentHyperbolicActivation(Double input)
  {
    Double  pos2X = Math.Exp(2 * input);
    Double  result = (pos2X - 1) / (pos2X + 1);
    return (result);
  }
}
```

Bibliography

Bishop, C. M. (2007). *Neural Networks for Pattern Recognition.* Oxford.

Bishop, C. M. (2006). *Pattern recognition and Machine Learning.* Springer.

Kleinberg, J., & Tardos, E. (2006). *Algorithm Design.* Addison Wesley.

Russel, S., & Norvig, P. (2003). *Artificial Intelligence A Modern Approach.* Prentice Hall.

Nkya, E. O. (2008). *Implementing Realistic Human Motion in Games.*