Neural Networks - An Introduction

To quickly define neurons, neural networks, and the back propagation algorithm.

Author	Abdul Habra
Email	ahabra@yahoo.com
URL	www.tek271.com
Date	2005.10.05
Version	0.13

1. Introduction

A Neural Network (NN) is a computer software (and possibly hardware) that simulates a simple model of neural cells in animals and humans. The purpose of this simulation is to acquire the *intelligent* features of these cells. In this document, when terms like *neuron, neural network, learning, or experience* are mentioned, it should be understood that we are using them only in the context of a NN as computer system.

NNs have the ability to learn by example, e.g. a NN can be trained to recognize the image of car by showing it many examples of a car.

We will discuss neurons, NNs in general, and Back Propagation networks. Back Propagation networks are a popular type of network that can be trained to recognize different patterns including images, signals, and text.

This article does not try to prove the usefulness of NNs, when they should be used, or why do they work. It is a high level summary with emphasize on how Back Propagation networks work.

2. History

- Serious research started in the 1950's and 1960's by researchers like Rosenblatt (Perceptron), Widrow and Hoff (ADALINE).
- In 1969 Minsky and Papert wrote a book exposing Perceptron limitations. This effectively ended the interest in neural network research.
- In the late 1980's interest in NN increased with algorithms like Back Propagation, Cognitrons and Kohonen. (Many of them where developed quietly during the 1970s)
- Progress continued during the 1990's.
- Currently NNs are used in many commercial applications like character recognition, image recognition, credit evaluation, fraud detection, insurance, and stock forecasting.

3. Sigmoid Function

The function: $s(x) = 1 / (1 + e^{-a * x})$

is called a Sigmoid function. The coefficient *a* is a real number constant. Usually in NN applications *a* is chosen between 0.5 and 2. As a starting point, you could use *a*=1 and modify it later when you are fine-tuning the network. Note that $s(0)=0.5$, $s(\infty)=1$, $s(-\infty)=0$. (The symbol ∞ means *infinity*).

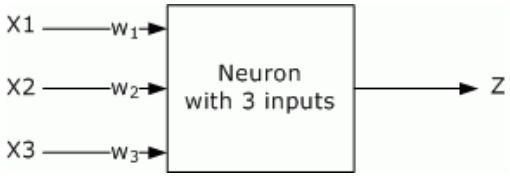
Think of the sigmoid function, in layman terms, as a function that will convert values less than 0.5 to 0, and values greater than 0.5 to 1. The Sigmoid function is used on the output of neurons as will be explained next.

4. Neuron

In a NN context, a neuron is a model of a neural cell in animals and humans. This model is simplistic, but as it turned out, is very practical. Think of the neuron as a program (*or a class if you like* ☺) that has one or more inputs and produces one output. The inputs simulate the stimuli/signals that a neuron gets, while the output simulates the response/signal which the neuron generates. The output is calculated by multiplying each input by a different number (called weight), adding them all together, then scaling the total to a number between 0 and 1.

The following diagram shows a simple neuron with:

1. Three inputs `[x1, x2, x3]`. The input values are usually scaled to values between 0 and 1.
2. Three input weights `[w1, w2, w3]`. The weights are real numbers that usually are initialized to some random numbers. Do not let the term*weight* mislead you, it has nothing to do with the physical sense of weight, in a programmer context, think of the weight as a variable of type float/real that you can initialize to a random number between 0 and 1.
3. One output z. A neuron has one (and only one) output. Its value is between 0 and 1, it can be scaled to the full range of actual values.



Let `d= (x1 * w1) + (x2 * w2) + (x3 * w3)`
In a more general fashion, for n number of inputs: (∑ means the *sum of*)

$$d = \sum x_i * w_i \qquad \text{... for } i=1 \text{ to } n$$

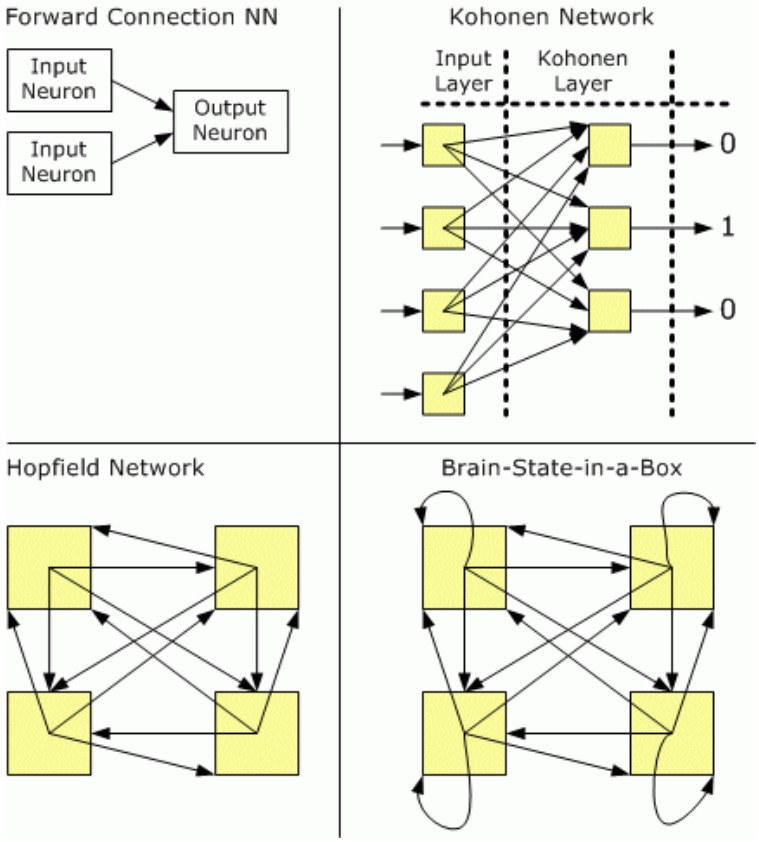
Let θ be a real number which we will call *Threshold*. Experiments have shown that best values for θ are between 0.25 and 1. Again, in a programmer context, θ is just a variable of type float/real that is initialized to any number between 0.25 and 1.

$$z = s(d + \theta) \qquad \text{... Apply the sigmoid to get a number between 0 and 1}$$

This says that the output z is the result of applying the sigmoid function on *(d + θ)*. In NN applications, the challenge is to find the right values for the weights and the threshold.

5. Neural Networks

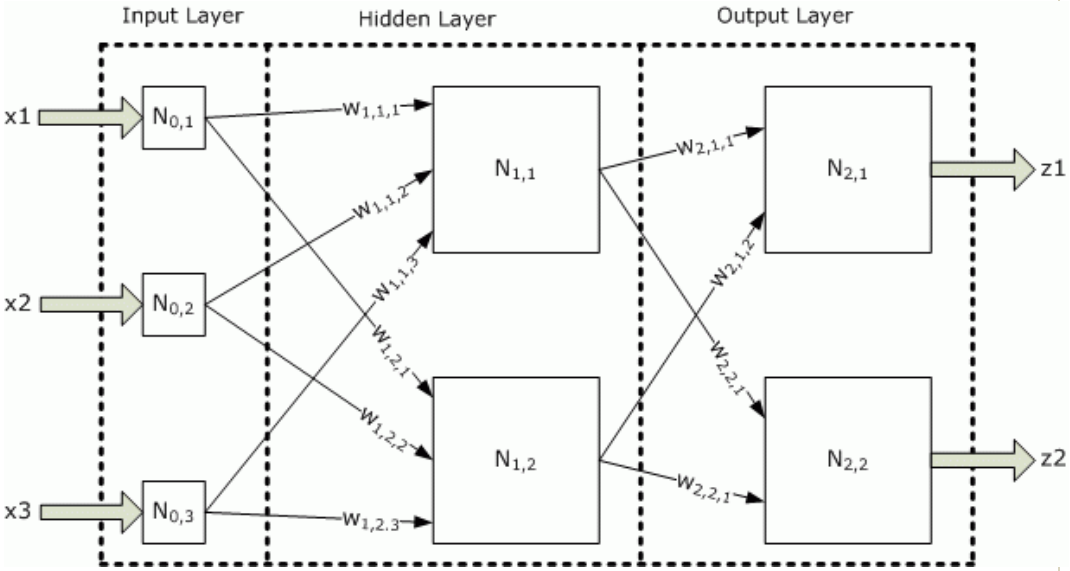
A neural network is a group of neurons connected together. Connecting neurons to form a NN can be done in various ways, next are some examples:



One of the popular NN is called the *Back Propagation* network which will be discussed next.

6. Back Propagation Networks

The following diagram shows a Back Propagation NN:



This NN consists of three layers:

1. Input layer with three neurons.
2. Hidden layer with two neurons.
3. Output layer with two neurons.

Note that:

1. The output of a neuron in a layer goes to all neurons in the following layer.
2. Each neuron has its own input weights.
3. The weights for the input layer are assumed to be 1 for each input. In other words, input values are not changed.
4. The output of the NN is reached by applying input values to the input layer, passing the output of each neuron to the following layer as input.
5. The Back Propagation NN must have at least an input layer and an output layer. It could have zero or more hidden layers.

The number of neurons in the input layer depends on the number of possible inputs we have, while the number of neurons in the output layer depends on the number of desired outputs. The number of hidden layers and how many neurons in each hidden layer cannot be well defined in advance, and could change per network configuration and type of data. In general the addition of a hidden layer could allow the network to learn more complex patterns, but at the same time decreases its performance. You could start a network configuration using a single hidden layer, and add more hidden layers if you notice that the network is not learning as well as you like.

For example, suppose we have a bank credit application with ten questions, which based on their answers, will determine the credit amount and the interest rate. To use a Back Propagation NN, the network will have ten neurons in the input layer and two neurons in the output layer.

6.1 Supervised Training

The Back Propagation NN works in two modes, a supervised training mode and a production mode. The training can be summarized as follows:

Start by initializing the input weights for all neurons to some random numbers between 0 and 1, then:

1. Apply input to the network.
2. Calculate the output.
3. Compare the resulting output with the desired output for the given input. This is called the *error*.
4. Modify the weights and threshold θ for all neurons using the *error*.
5. Repeat the process until *error* reaches an acceptable value (e.g. *error* < 1%), which means that the NN was trained successfully, or if we reach a maximum count of iterations, which means that the NN training was not successful.

The challenge is to find a good algorithm for updating the weights and thresholds in each

iteration (step 4) to minimize the *error*.

Changing weights and threshold for neurons in the output layer is different from hidden layers. Note that for the input layer, weights remain constant at 1 for each input neuron weight.

Before we explain the training, let's define the following:

1. λ (*Lambda*) the Learning Rate: a real number constant, usually 0.2 for output layer neurons and 0.15 for hidden layer neurons.
2. Δ (*Delta*) the change: For example Δx is the change in x. Note that Δx is a single value and not Δ multiplied by x.

6.2 Output Layer Training

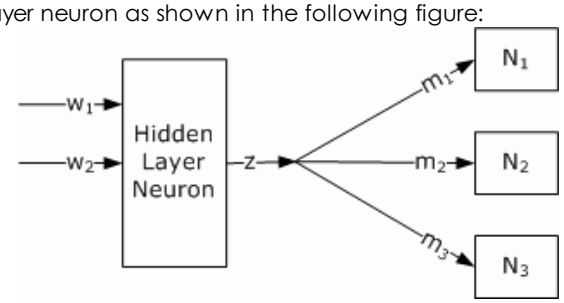
- Let z be the output of an output layer neuron as shown in section 4.
- Let y be the desired output for the same neuron, it should be scaled to a value between 0 and 1. This is the ideal output which we like to get when applying a given set of input.
- Then e (the *error*) will be:

$$e = z * (1 - z) * (y - z)$$
$$\Delta\theta = \lambda * e \quad \text{... The change in } \theta$$
$$\Delta w_i = \Delta\theta * x_i \quad \text{... The change in weight at input i of the neuron}$$

In other words, for each output neuron, calculate its error e, and then modify its threshold and weights using the formulas above.

6.3 Hidden Layer Training

Consider a hidden layer neuron as shown in the following figure:



- Let z be the output of the hidden layer neuron as shown in section 4.
- Let m_i be the weight at neuron N_i in the layer following the current layer. This is the weight for the input coming from the current hidden layer neuron.
- Let e_i be the error (e) at neuron N_i.
- Let r be the number of neurons in the layer following the current layer. (In the above diagram r=3).

$$g = \sum m_i * e_i \quad \text{... (for i=1 to r)}$$
$$e = z * (1 - z) * g \quad \text{... Error at the hidden layer neuron}$$
$$\Delta\theta = \lambda * e \quad \text{... The change in } \theta$$
$$\Delta w_i = \Delta\theta * x_i \quad \text{... The change in weight i}$$

Notice that in calculating g, we used the weight m_i and error e_i from the following layer, which means that the error and weights in this following layer should have already been calculated. This implies that during a training iteration of a Back Propagation NN, we start modifying the weights at the output layer, and then we proceed backwards on the hidden layers one by one until we reach the input layer. It is this method of proceeding backwards which gives this network its name *Backward Propagation*.

7. Conclusion

NNs are being used in many businesses and applications. Their ability to learn by example is very attractive in environments where the *business rules* are either not well defined or are hard to enumerate and define.

You may wander about the formulae and constants values, why did we do this or choose that. That's good but beyond the scope of this article. Check some of the resources for more

details.

8. Resources

1. *C++ Neural Networks and Fuzzy Logic* by V. B. Rao and H. V. Rao. MIS 1993.
2. *Neural Network Computing* by R. Bharath and J. Drosen. McGraw-Hill 1994.
3. *Neurocomputing* by R. Hecht-Nielsen. Addison-Wesley 1990.
4. *Neural Computing: An Introduction* by R. Beale and T. Jackson. Institute of Physics Publishing 1990.
5. *Perceptrons - Expanded Edition: An Introduction to Computational Geometry* by M. Minsky and S. Papert. MIT Press 1987.
6. www.jooneworld.com: a free Neural Network engine written in Java.
7. www.tek271.com/free/nuExpert.html: a free Neural Network engine.

Acknowledgement: *Doug Estep provided valuable feedback reviewing this article.*

Comments

Abdul Habra's Blog

Neural Networks - An Introduction

Moved this paper to the new Google Apps site on 2011.03.29

http://www.tek271.com/documents/others/into-to-neural-networks

Posted 2 years ago 2 notes • 17 Comments

 ahabra posted this

Disqus seems to be taking longer than usual. Reload?

Comments

Commenting disabled due to a network error. Please reload the page.

You do not have permission to add comments.