Lecture 11: Neural Networks

CSE 427: Machine Learning

Md Zahidul Hasan

Lecturer, Computer Science
BRAC University

Spring 2023

An Inspiration From Neuroscience

The building block of artificial neural networks is the neuron. A neuron is a computational unit that takes several real valued inputs and produces one real valued output. In this way, an artificial neuron is very different from a biological neuron that produces a time series of spikes.

Roughly estimated, the human brain is a complex interconnected web of 10^{11} neurons. Each of these are connected to 10^4 other neurons on average. Each neuron produces a signal as an output that is fed to another neuron as an input. This event is called firing or neuron switching. For the human brain, it takes 10^{-3} seconds to perform a single neuron switch. For a computer it takes only 10^{-10} seconds. Yet human beings are capable of making astonishingly complex decisions in a very short amount of time. For example, it takes only 10^{-1} seconds to visually recognize our mothers. That is possible because in the human brain, these neuron-switchings take place in a highly parallel environment.

The Configuration of an Artificial Neural Network

There are two essentials layers to any neural network. They are the input layer and the output layer. The input layer consists of pre-processed values of the various features.

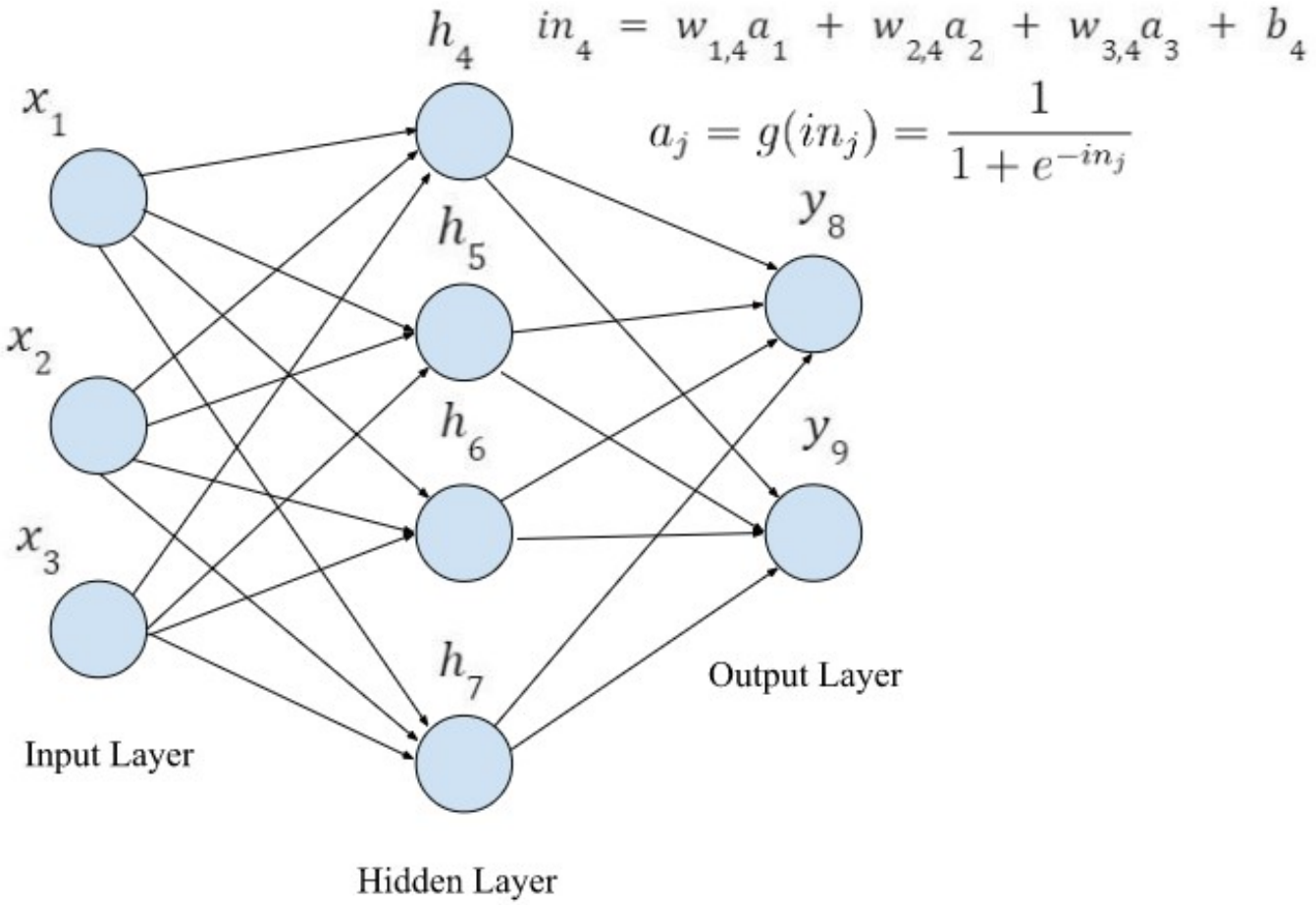


Figure: A Neural Network with one hidden layer.

Components of a Neural Network

There can be one or more hidden layers between the input and output layers. Each node in the neural network has an **activation function**.

In most cases, the activation function is the logistic/sigmoid function, or the ramp function, or the hyperbolic tan or the sign function or the softmax function or just the identity function.

Nodes in the input layer has the identity function as the activation function.

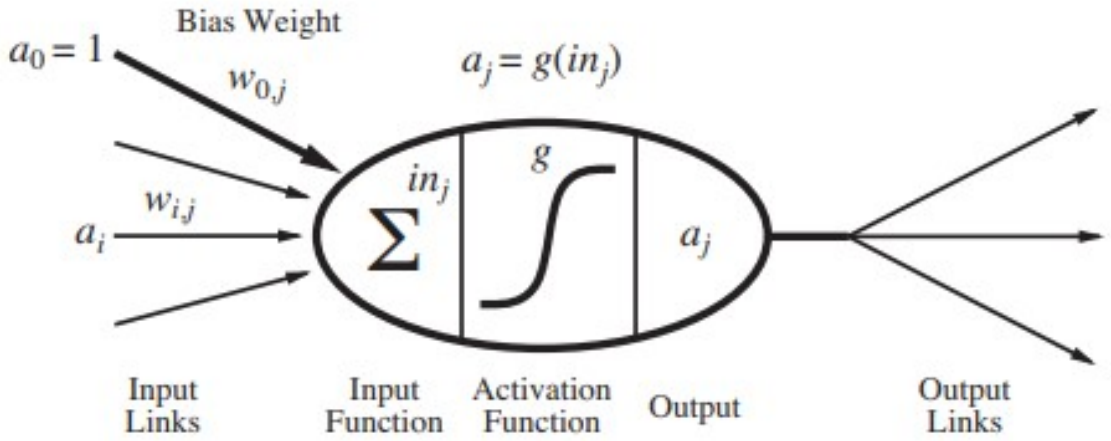
The activation function of node j takes an input in_j and produces an output called the activation $a_j = f(in_j)$ where f is the activation function. In the case of nodes in the input layer, $in_j = x_j$ and the activation function is the identity function $f(x) = x$. So, the activation produced by a neuron in the input layer is $a_j = f(in_j) = in_j = x_j$.

Components of a Neural Network Continued

A neuron in the hidden layer takes the activations of the neurons of the previous layer as inputs. It then takes a linear combination of those inputs and adds a bias to it. So, let's say, neuron j in a hidden layer has the following input:

$$in_j = w_{1,j}a_1 + w_{2,j}a_2 + \cdots + w_{k,j}a_k + b_j$$

Every neuron j has a parameter called the bias b_j. Each neuron j gives a neuron i from the previous node a weight called $w_{i,j}$.



Components of a Neural Network Continued

Sometimes the bias weight b_j is written as $w_{0,j}$ for neuron j. When neuron j receives an activation from neuron i, it gets multiplied by $w_{i,j}$. When a neuron has computed it's input in_j, then it produces an activation using an activation function f. Here are some examples of activation functions:

Logistic function: $g(x) = \frac{1}{1+e^{-x}}$.

Identity function: $f(x) = x$

Ramp function: $f(x) = max(0, x)$

Hyperbolic tan: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Softmax function:

$softmax(y) = \{\frac{e^{y1}}{e^{y1}+e^{y2}+\cdots+e^{yk}}, \frac{e^{y2}}{e^{y1}+e^{y2}+\cdots+e^{yk}}, \cdots, \frac{e^{yk}}{e^{y1}+e^{y2}+\cdots+e^{yk}}\}$

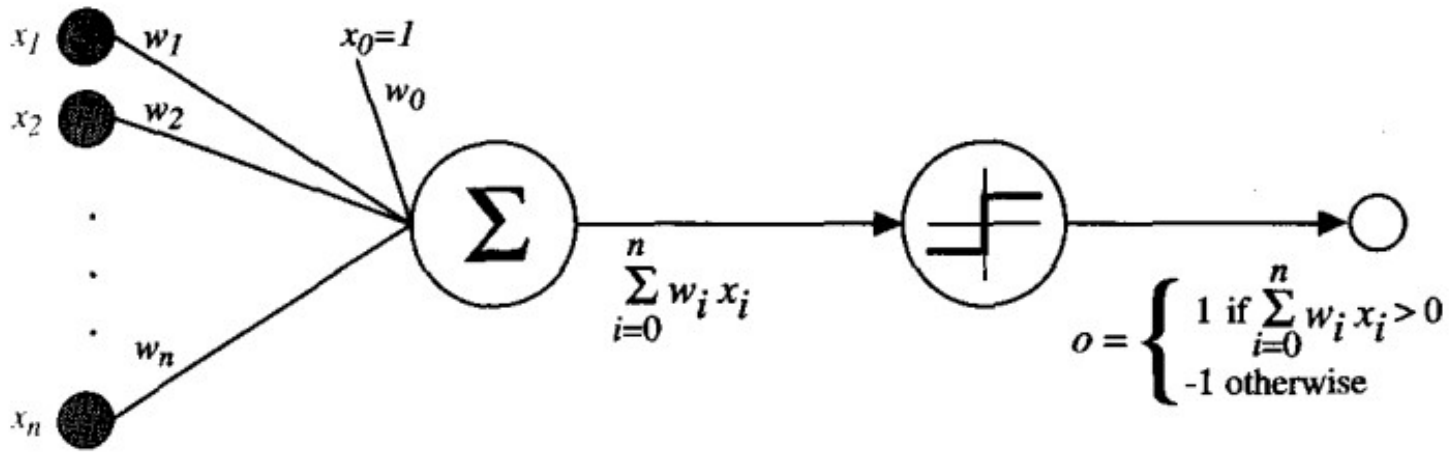
Threshold function: $sign(x) = \{-1 \text{ if } x \leq 0, +1 \text{ otherwise}\}$

If the activation is a threshold function, then the neural network is called a **Perceptron**. If the activation is a sigmoid function, then it's called a **Sigmoid Perceptron**.

Connecting Neurons in a Neural Network

The most common example of a NN is the **feed-forward network**. The input layer propagates it's inputs to the first hidden layer. The hidden layers in turn propagate their activations to the neurons in the next layer. No edge goes backwards in a feed-forward network. But in a **recurrent neural network (RNN)** the outputs of the output layer is fed back into the input layer as input. If the neurons of a layer is connected to all the neurons of the next layer of a neural network, it is called a **fully-connected neural network**. If a fully-connected multi-layer neural network has a threshold activation function in each of the neurons, then it is called a **multi-layer perceptron (MLP)**. The **optimal brain damage** algorithm helps to weed out useless connections from a fully-connected neural network. That way the neural network becomes lightweight and faster.

The Perceptron



The Perceptron is a neural network with no hidden layer. It's output layer has a single output neuron and the activation function is a threshold function. Boolean functions such as AND, NOT, OR can be implemented using perceptrons. Relatively complex functions such as the majority function (more than half of n people have to agree) or m of n functions can be implemented as well. For the majority function, each input $x_i \in \{0, 1\}$ and each $w_i = 1$ and $w_0 = \frac{-n}{2}$.

Learning Weights in a Multi-layer Neural Network

We do the usual. We feed the inputs forward and generate outputs in the output layer. In the case of regression, we take the squared error and then apply gradient descent to minimize the error. Let's say, we have k output nodes. Then, we can measure the error as follows:

$$E = \sum_k (t_k - o_k)^2$$

Here, t_k is the target value on the k'th output node. o_k is the output of the NN. o_k is nothing but the activation of the k'th output node. Therefore, $o_k = a_k$. Let's find out the weight update rule for weight $w_{j,k}$ which is the weight from node j to node k on the output layer. Node j is on the layer right before the output layer.

$$\frac{\delta E}{\delta w_{j,k}} = -\sum_k 2(t_k - a_k)\frac{\delta a_k}{\delta w_{j,k}} = -2(t_k - a_k)\frac{\delta a_k}{\delta w_{j,k}}$$

This happens because $w_{j,k}$ doesn't affect the outputs of other output nodes.

Weight Update Rules

Since, $a_k = g(in_k)$ and $in_k = a_1 w_{1,k} + a_2 w_{2,k} + ... + a_n w_{n,k}$ where $a_1, a_2, \cdots, a_n$ are the activations of neurons from the previous layer. Therefore,

$$\frac{\delta a_k}{\delta w_{j,k}} = \frac{\delta g(in_k)}{\delta w_{j,k}} = \frac{\delta in_k}{\delta w_{j,k}} \frac{\delta g(in_k)}{\delta in_k} = a_j g'(in_k)$$

So, the weight update rule should be:

$$w_{j,k} = w_{j,k} - \lambda \frac{\delta E}{\delta w_{j,k}} = w_{j,k} + 2\lambda(t_k - a_k) a_j g'(in_k).$$

We now define $\Delta_k = (t_k - a_k) g'(in_k)$. And taking 2λ to be λ, the weight update rule becomes:

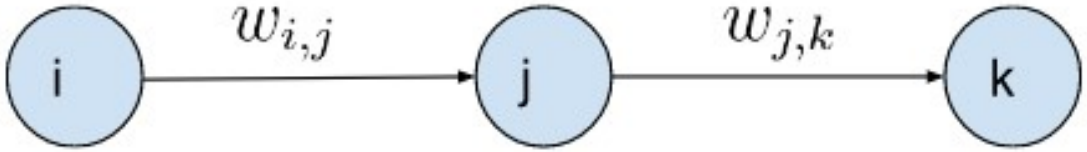
$$w_{j,k} = w_{j,k} + \lambda a_j \Delta_k.$$

We find $g(in_k)$ using the following formula:

$$g'(x) = g(x)(1 - g(x)).$$

Weight Updates Rules for Hidden Layers

Let's say, neuron k is on the output layer, neuron j is on the layer before that and neuron i on the layer before that. Let's derive weight update rules for $w_{i,j}$.



We know that the error is $E = \sum_k (t_k - a_k)^2$. Let's find out how E changes with respect to $w_{i,j}$:

$$\frac{\delta E}{\delta w_{i,j}} = -2\sum_k (t_k - a_k)\frac{\delta a_k}{\delta w_{i,j}} = -2\sum_k (t_k - a_k)\frac{\delta in_k}{\delta w_{i,j}} g'(in_k)$$

Previously, we defined $\Delta_k = (t_k - a_k)g'(in_k)$. Therefore,

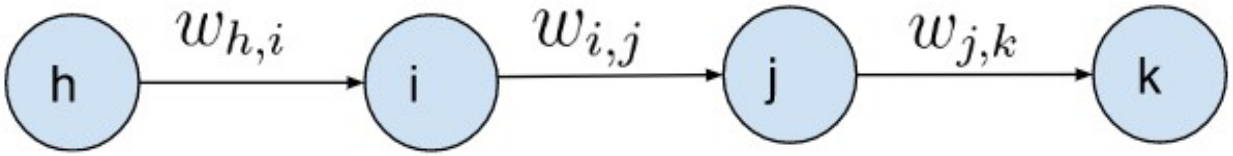
$$\frac{\delta E}{\delta w_{i,j}} = -2\sum_k \Delta_k w_{j,k}\frac{\delta a_j}{\delta w_{i,j}} = -2\sum_k \Delta_k w_{j,k}\frac{\delta in_j}{\delta w_{i,j}} g'(in_j) = -2\sum_k \Delta_k w_{j,k} a_i g'(in_j) = -2a_i g'(in_j)\sum_k \Delta_k w_{j,k}$$

By defining, $\Delta_j = g'(in_j)\sum_k w_{j,k}\Delta_k$, we get,

$$w_{i,j} = w_{i,j} + \lambda a_i \Delta_j$$

The General Back-Propagation Rule

So, is there a general rule for back-propagating the weight-updates? Let's find out.



$\frac{\delta E}{\delta w_{h,i}} = -2\sum_k (t_k - a_k)\frac{\delta a_k}{\delta w_{h,i}} = -2\sum_k (t_k - a_k) g'(in_k)\frac{\delta in_k}{\delta w_{h,i}}$

$= -2\sum_k (t_k - a_k) g'(in_k)\sum_j w_{j,k}\frac{\delta a_j}{\delta w_{h,i}} = -2\sum_k \Delta_k \sum_j w_{j,k}\frac{\delta a_j}{\delta w_{h,i}}$

$= -2\sum_k (t_k - a_k) g'(in_k)\sum_j w_{j,k} g'(in_j)\frac{\delta in_j}{\delta w_{h,i}} =$

$-2\sum_k \Delta_k \sum_j w_{j,k} g'(in_j)\frac{\delta in_j}{\delta w_{h,i}} = -2\sum_k \Delta_k \sum_j w_{j,k} g'(in_j) w_{i,j}\frac{\delta a_i}{\delta w_{h,i}}$

$= -2\sum_j w_{i,j}\frac{\delta a_i}{\delta w_{h,i}} g'(in_j)\sum_k \Delta_k w_{j,k}$

Since $\Delta_j = g'(in_j)\sum_k \Delta_k w_{j,k}$, we have,

$\frac{\delta E}{\delta w_{h,i}} = -2\sum_j w_{i,j}\frac{\delta a_i}{\delta w_{h,i}}\Delta_j = -2\sum_j w_{i,j} g'(in_i)\frac{\delta in_i}{\delta w_{h,i}}\Delta_j$

$\frac{\delta E}{\delta w_{h,i}} = -2a_h g'(in_i)\sum_j w_{i,j}\Delta_j = -2a_h\Delta_i$. Hence the general weight update rule is:

$w_{h,i} = w_{h,i} + \lambda a_h \Delta_i$

Adding Momentum to Back-Propagation

The gradient descent search trajectory reaches a global extrema like a momentum-less ball that gets stuck. To overcome this problem, we can add some momentum to the weight update formula as follows:

$$w_{i,j,n} - w_{i,j,n-1} = \alpha(w_{i,j,n-1} - w_{i,j,n-2}) + \lambda a_{i,n}\Delta_{j,n}$$

In other words,

$$w_{i,j,n} = w_{i,j,n-1} + \alpha(w_{i,j,n-1} - w_{i,j,n-2}) + \lambda a_{i,n}\Delta_{j,n}$$

Here, $w_{i,j,n}$ means the updated weight $w_{i,j}$ after iteration n. $a_{i,n}$ means the activation of node i during iteration n. $\Delta_{j,n}$ means the value of Δ_j during iteration n. In this way, even if $a_{i,n}\Delta_{j,n} = 0$, the gradient descent doesn't stop in local extremum points or on flat surfaces. Even in cases where the gradient doesn't change, the step size keeps increasing which speeds up convergence.