Artificial Neural Networks
Lecture Notes

Stephen Lucci, PhD

Part 5

About this file:

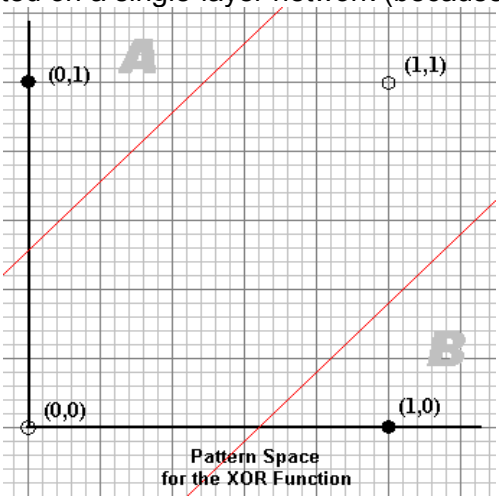
- If you have trouble reading the contents of this file, or in case of transcription errors, email gi0062@bcmail.brooklyn.cuny.edu
- Acknowledgments:
  Background image is from http://www.anatomy.usyd.edu.au/online/neuroanatomy/tutorial1/tutorial1.html (edited) at the University of Sydney Neuroanatomy web page. Mathematics symbols images are from metamath.org's GIF images for Math Symbols web page. Other image credits are given where noted, the remainder are native to this file.

Contents

Backpropagation

- ***Multilayered Networks***
  Multilayered Networks can compute a wider range of Boolean functions than single-layered networks.

e.g., XOR cannot be computed on a single-layer network (because it is not linearly separable):



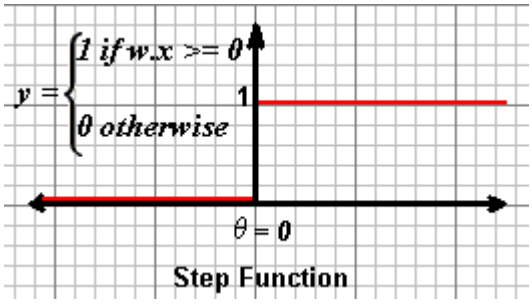
- Pricetag (the trade-off): Learning is *slower*!

- ***The Backpropagation Algorithm*** - employs **gradient descent** to find the minimum of the error function in weight space:

  $E(w_1, w_2, ..., w_n)$

- The Error Function must be:
  - *Continuous*, and
  - *Differentiable*

- Hence, the *Step function* is inadequate:



  It is neither continuous nor differentiable.

- A popular activation function is the **Sigmoid** - a real-valued function:

  $S_c: \mathbb{R} \longrightarrow (0,1)$,

  $$S_c(x) = \frac{1}{1+e^{-cx}}$$

  See figure 7.1 in Rojas.

- For higher values of *c* - the sigmoid resembles (or approximates) the step function.
- We let *c* = 1 (i.e., $S_1(x)$, written $s(x)$.)

- ***Recalling our calculus***

  The Reciprocal Rule

  If g is differentiable at x, and g(x) ≠0, then
  1/g is differentiable at x, and

  $$(1/g)'(x) = -g'(x) / (g(x))^2$$

- Hence, the derivative of the sigmoid with respect to x is

$$\frac{d}{dx} s(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= s(x)(1 - s(x)) .$$

*Alternatives to the Sigmoid **

***** (See figure 7.2 in Rojas; classic sigmoid is the upper-left graph in the figure.)

- **Symmetrical Sigmoid S(x)**:
  $S(x) = 2s(x) - 1 = (1 - e^{-x}) / (1 + e^{-x})$
  Hyperbolic tangent for x/2, where $\tanh x = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$
  (The upper right graph in figure 7.2 in Rojas)
- **Ramp Function**:
  (Lower-right graph in figure 7.2 in Rojas)
  One must avoid the two points where the derivative is undefined.

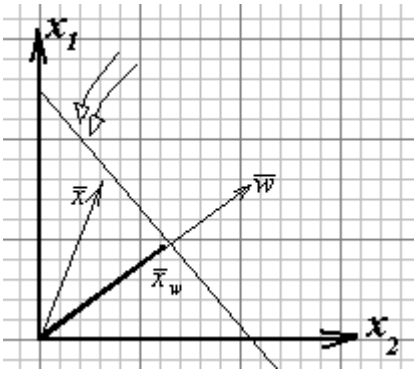
- ***Smoothing*** is produced by a sigmoid in a step of the error function.
  (see figure 7.3 in Rojas.)
- We follow the gradient direction (negative gradient direction!) to find the minimum of this function.
- Sigmoid always has a positive derivative; the slope of the error function provides a descent direction which can be followed:

$$\frac{e^{-x}}{(1+e^{-x})^2}$$

- "We can think of our search algorithm as a physical process in which a small sphere is allowed to roll on the surface of the error function until it reaches the bottom."
  Rojas, p. 151.
- The sigmoid uses the net amount of excitation as its argument.
- Given weights $w_1, ..., w_n$ and a bias $-\theta$, a sigmoidal unit computes for the input $x_1, ..., x_n$, the output

$$\frac{1}{1 + \exp\left(\sum_{i=1}^{n} w_i x_i - \theta\right)}$$

- A higher net amount of excitation brings the unit's output closer to 1.



(Also see fig. 7.4 in the textbook.)

- The step of the sigmoid is normal to the vector $(w_1, ..., w_n, -\theta)$.
- The weight vector points in the direction in extended input space, in which the output of the sigmoid changes faster.

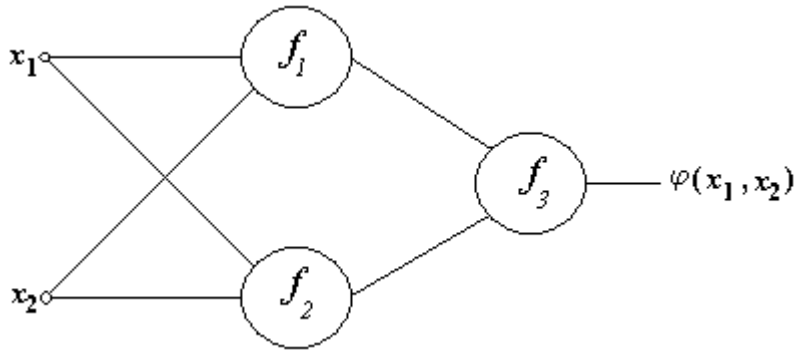
Drawback for the Sigmoid As Activation Function

- *Local minima* appear in the error function which would not be present if the step function had been used. *Refer to fig. 7.5 in Rojas.*
- There is a local minimum in figure 7.5 with a higher error level than in other regions.
- If gradient descent begins in this valley, the algorithm will not converge to the global minimum.

The Learning Problem

- Each neuron (computing unit) in a neural network evaluates a single primitive function of its input.
- The network represents a chain of function compositions which transform an input to an output vector (called a *pattern*.)
- The network is a particular implementation of a *composite function* from input to output space, which is called the *network function*.
- The **learning problem** consists of finding the optimal combination of weights so that the network function φ approximates a given function *f* as closely as possible.

  HOWEVER, we are not given the function *f* explicitly, but only implicitly through some examples.



- We are given a *feed-forward network* with n inputs and m output units. The number of *hidden units* is arbitrary and any feed-forward connection pattern is permitted.
- We are also given a train set { $(\bar{x}_1, \bar{t}_1)$, ... , $(\bar{x}_p, \bar{t}_p)$ }
  $\varphi: \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2^m$
  Input patterns $\bar{x}_i$ are *n*-dimensional vectors.
  Output patterns $\bar{t}_i$ are *m*-dimensional vectors.
- The *primitive functions* at each node in the network are continuous and differentiable.
- The weights of the edges are Real numbers randomly chosen.
- Input pattern $\bar{x}_i$ from the training set is presented to this network. It produces an output $\bar{o}_i$ which may be different from the target $\bar{t}_i$ (the desired output).
- Our goal is to make $\bar{o}_i$ and $\bar{t}_i$ identical for i = 1, ..., p by using a *learning algorithm*.

  i.e., our goal is to minimize the error function of the network, where

$$E = \frac{1}{2}\sum_{i=1}^{p} \| o_i - t_i \|^2$$

- After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to *interpolate*.

  i.e., the network must recognize whether a new input vector is similar to learned patterns and produce a similar output.

- The backpropagation algorithm is used to find a local minimum of the error function.
- The gradient of the error function is computed and is used to correct the initial weights.
  see figure 7.6 in Rojas.
- The network is extended, so that it computes the error function automatically.
- Every one of the j output units of the network is connected to a node which evaluates the function

  $$\frac{1}{2}(o_{ij} - t_{ij})^2$$

  where o_{ij} and t_{ij} denote the j^{th} component of the output vector $\bar{o}_i$ and the target $\bar{t}_i$.

- The outputs of these m additional nodes are collected at a node which adds them up and gives the sum E_i as its output.
- The same network extension is built for each pattern t_i or repeated p times !
- A computing unit collects all quadratic errors and outputs their sum $E_1 + ... + E_p$.
- We now have a network capable of calculating the total error for a given training set.
- The weights in the network are the only parameters that can be modified to make the quadratic error E as low as possible.
- We can minimize E by using an iterative process of gradient descent.
- We will need to calculate the gradient

  $$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_L}\right)$$

- Each weight is updated using the increment

  $\Delta w_i = - y (\partial E / \partial \Delta w_i)$ for i = 1, ..., L .

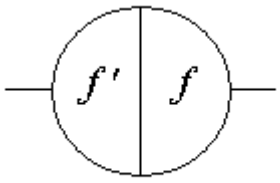
  where y is the learning rate - i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

- The learning problem reduces to calculating the gradient of a network function with respect to its weights.
  $\nabla E = 0$ at the minimum of the error function.

Derivatives of Network Functions

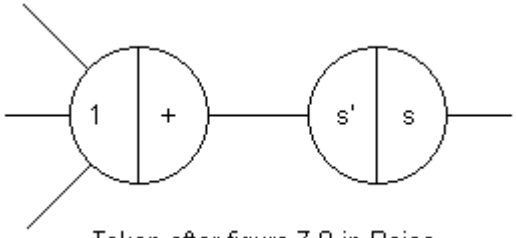
Calculating the Gradient

- B-diagram - (Backpropagation diagram):
  Each node in the neural network consists of a left and right side.



Taken after fig. 7.7 in Rojas

- The right side computes the primitive function *f* associated with the node.
- The left side computes the derivative *f'* for some input.
- **Note**: The *Intergration* function can be separated from the *Activation* function by splitting each node s.t. - The First node computes sum of inputs, - The second node computes the activation function s.



Taken after figure 7.8 in Rojas

- The derivative of s is s'.
- The partial derivative of the sum of n arguments with respect to any one of them is just 1.

Network Evaluation

Stage 1

Feedforward Step -
Information comes from the left and each unit evaluates its primitive function *f* in its right side, as well as the derivative *f'* in its left side.
Both results are stored in the unit, but only the result from the right side is transmitted to the units connected to the right.

Stage 2

Backpropagation Step -
The whole network is run backwards, and the stored results are now used.

First Case: Function Composition

- Refer to figure 7.9 in Rojas.
- In the feed-forward step, the network computes the composition of the functions *f* and *g*. Incoming information into a unit is used as the argument for the evaluation of the node's primitive function and its derivative.
- After the feedforward step, we have what is shown in figure 7.10.
- The correct result of the function composition has been produced at the output.

Each unit has stored some information on its left side.

- In the backpropagation step, the input from the right of the network is the constant 1.
- Incoming information to a node is multiplied by the value stored in its left side.
- The result of the multiplication is transmitted to the next unit to the left. The result at each node is called the *traversing value* at this node:

  Refer to figure 7.11.

- The final result of the backpropagation step is $f'(g(x))\ g'(x)$, i.e. , the derivative of the function composition $f(g(x))$ implemented by the network.
- The backpropagation step thus provides an implementation of the chain rule.
- One may think of the network as being run *backwards* with the input 1, whereby, at each node, the product with the value stored in the left side is computed.

Second Case: Function Addition

- A network for the computation of the addition of the functions f_1 and f_2.
- Refer to figure 7.12:

  Note that the extra node handles the addition of the functions.

- In the feed-forward step the network computes $f_1(x) + f_2(x)$.
- In the backpropagation step the constant 1 is fed from the right side into the network:

  Refer to figure 7.13.

- All incoming edges to a unit fan out the *traversing value* at this node and distribute it to the connected units to the left.
- When two right-to-left paths meet, the computed traversing values are added.

Third Case: Weighted Edges

- In the feed-forward step, the incoming information x is multiplied by the edge's weight w.

  Refer to figure 7.14: In the feedforward step, the input is x , and the result is wx.
  In the backpropagation step, the input is 1, and the result is d(wx)/dx.

- In the backpropagation step, the traversing value 1 is multiplied by the weight of the edge. The result is w, which is the derivative of wx with respect to x.

Steps of the Backpropagation Algorithm

- We assume we are dealing with a network with a single input and a single output unit.

This is easily generalized...

- From Rojas:

Algorithm 7.2.1 Backpropagation algorithm [Computing the derivative of a network function]

Consider a network with a single real input *x* and network function *F*. The derivative *F'(x)* is computed in two phases:

Feed-forward:	the input *x* is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.
Backpropagation:	the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to *x*.

The following proposition shows that the algorithm is correct.

Proposition 7.2.1 *Algorithm 7.2.1 computes the derivative of the network function F with respect to the input x correctly.*

[- End Rojas Quote -]

- The above algorithm generalizes to Activation functions of several variables, see pp. 160-161.

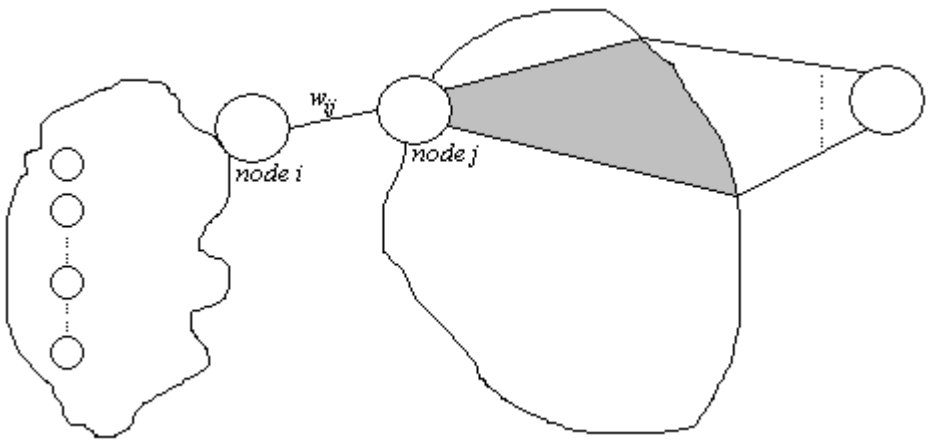
Learning with Backpropagation

- **Goal** - to minimize the error function E

$$E(w_1, w_2, \ldots, w_{n+1})$$

We must deal with all the weights in the network one at a time. We store the output of each unit in its right side.

- We perform the backpropagation step in the extended network that computes the error function.
- We fix our attention on one of the weights, say w_{ij} .



- This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending at the single output unit of the network (shaded region above.)
- The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i.

  $x \xrightarrow{w} wx$

- The backpropagation step computes the gradient of E with respect to this input:

  i.e., $\partial E / \partial o_i w_{ij}$

- In the backpropagation step, o_i is treated as a constant. Hence we have,

  $\partial E / \partial w_{ij} = o_i \, (\partial E / \partial o_i w_{ij})$

- All subnetworks defined by each weight of the network can be handled simultaneously.
- The following additional information must be stored at each node:
  - The output o_i of the node in the feed-forward step.
  - The cumulative result of the backward computation in the backpropagation step up to this node. This quantity is called the *backpropagated error*.
- Let σ_j = the backpropagated error at the j^{th} node. Then,

  $\partial E / \partial w_{ij} = o_i \sigma_j$

  Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

  $\Delta w_{ij} = -y \, o_i \sigma_j$

  where y is the learning rate, and $o_i \sigma_j$ is the gradient. This correction step transforms the backpropagation algorithm into a learning algorithm for neural networks.

Layered Networks

- An important special case of feed-forward networks is that of *layered networks* with one or more hidden layers.

  Refer to figure 7.17 in Rojas.

n	input sites
k	hidden units
m	output units
$w^{(1)}_{ij}$	the weight between input i and hidden unit j
$w^{(2)}_{ij}$	the weight between hidden unit i and output j
$-\theta$	the bias of each unit implemented as the weight of an additional edge
$w^{(1)}_{n+1,i}$	the weight between the constant 1 and the hidden unit j
$w^{(2)}_{k+1,i}$	the weight between the constant 1 and the output unit j

---