Multi-Layer Networks & Backpropagation
Multi-Layer Networks

- Generally much more versatile than single neurons
- No linear-separability requirement for problem space
- Training approach is less obvious and potentially more time consuming.
Multi-Level Networks

Several varieties, the most common of which is known as any of these:

- MLP (Multi-Level Perceptron)
- Feed-forward network
- Backpropagation Network (alluding to a common method of training these networks; other training methods could conceivably be used, so this is not a good name for the networks themselves.)
In the text the input is counted as a layer, so this is 3-layers.
The real layers other than the output are called “hidden” layers.
Demo nnd11nf
(nf = network function)

- Shows a simple 2-level network:
  - 1 input, 1 output
  - 2 neurons in first layer, with 1 weight and 1 bias each, logsig activation function
  - 1 neuron in output layer, with 2 weights and 1 bias
  - Output activation function selectable from: purelin (identity), tansig, logsig
- Plot is network output vs. input
Demo nnd11nf

The neural network design and network function are shown in the image. The network weights and biases can be altered by dragging the triangular shaped indicators. Drag the vertical line in the graph below to find the output for a particular input. Click on [Random] to set each parameter to a random value.

Chapter 11
Function Approximation Demo nnd11f

Input  Log-Sigmoid Layer  Linear Layer
(can change to sigmoid)

\[ f^1(n) = \frac{1}{1 + e^{-n}} \]
logsig

\[ f^2(n) = n \]
purelin

Nominal Parameter Values

\[ w^1_{1,1} = 10 \quad w^1_{2,1} = 10 \quad b^1_1 = -10 \quad b^1_2 = 10 \]

\[ w^2_{1,1} = 1 \quad w^2_{1,2} = 1 \quad b^2 = 0 \]
Nominal Response

Output

Input
Example Parameter Variations

\begin{align*}
0 \leq b_2^1 \leq 20 & \\
-1 \leq w_1^2, 1 \leq 1 & \\
-1 \leq w_1^2, 2 \leq 1 & \\
-1 \leq b_2^2 \leq 1 &
\end{align*}
Function Approximation Demo

Click the [Train] button to train the logsig-linear network on the function at left. Use the slide bars to choose the number of neurons in the hidden layer and the difficulty of the function.
Generalization Demo

Click the [Train] button to train the logsig-linear network on the data points at left. Use the slide bar to choose the number of neurons in the hidden layer.
How to Train a MLP?

- With a single linear neuron, we have an Adaline. We know how to adapt it. A similar approach can be used for a logsig neurone.

- With a multi-layer network, it is less obvious. For one thing, **what is the “error” for the neurons in non-final layers?** Without these, we don’t know how to adjust.

- This is called the “credit assignment” problem (maybe should be “blame assignment”).
Discovery of Backpropagation

- Werbos, in his Harvard PhD thesis in 1974 found a method, but it was not widely disseminated.

- Rumelhart and McClelland, in 1985, discovered the method, presumably independently, and popularized it under the current name.

- In mathematics, such methods are in the category of “optimization”.
Backpropagation

- The technique is gradient descent, as explained for Adalines.

- However, the computation of the gradient was less clear.
Backpropagation Training Cycle

- **Forward propagation**: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.

- **Compute the error** in the output.

- **Backpropagate** the error through the network to get gradient values, aka “*sensitivities*”, at each neuron.

- Use the sensitivities to **derive weight changes**.

- Apply the weight changes.
Backpropagation Training Cycle

- Backpropagate is mathematically a lot like forward propagate, with sensitivities instead of signal values.

- The sensitivities are the partial derivatives of the MSE with respect to the activation values.

- Basically both are iterated matrix multiplications and applications of the activation functions of the neurons or their derivatives.
Multi-Layer Network

Each box has a row-vector of weights and a bias.

Each layer has a matrix of weights and a column vector of biases.
Multi-Layer Network

- Given an input vector, can compute the outputs.
- Given a sample, can compute the errors in output.
- Knowing gradient, can adjust the weights.
- Big Question: How to compute the gradient?
Single-Layer Network

- Recall that the gradient consists of components $\partial J/\partial w$ where $J$ is the mean-squared error and $w$ is one of the weights (including biases) in the network.

- For the generalized Adaline, with activation function $f$, we previously derived the on-line approximation:

  $$\frac{\partial J}{\partial w_i} \approx -2 \varepsilon x_i f'(v)$$

  where $x_i$ is the input corresponding to weight $w_i$, $v$ is the weighted sum of inputs, and $\varepsilon$ is the error. This works as is for the multi-layer case at the output layer.
Previous Derivation

- $\text{MSE} = J = E[(\text{desired-actual})^2]$ expected value of squared error

- $\frac{\partial J}{\partial w_i} = \frac{\partial}{\partial w_i} \sum (\text{desired-actual})^2/n$
  
  $$= (1/n) \sum 2(\text{desired-actual}) \frac{\partial}{\partial w_i} (\text{desired-actual})$$
  
  $$= - \frac{2}{n} \sum (\text{desired-actual}) \frac{\partial}{\partial w_i} \text{actual}$$

- But actual = $f(\sum w_j x_j)$, so
  
  $$\frac{\partial}{\partial w_i} \text{actual} = \frac{\partial}{\partial w_i} f(\sum w_j x_j)$$
  
  $$= f'(\sum w_j x_j) \frac{\partial}{\partial w_i} (\sum w_j x_j)$$
  
  $$= x_i f'(\sum w_j x_j)$$

- Hence $\frac{\partial J}{\partial w_i} = - \frac{2}{n} \sum (\text{desired-actual}) x_i f'(\sum w_j x_j)$ where the **outer summation is over all patterns.**
Batch vs. On-Line

- **Batch**: $\frac{\partial J}{\partial w_i} = -\frac{2}{n} \sum \text{(desired-actual)} \cdot x_i \cdot f'(\sum w_j x_j)$
  
  where the outer summation is over all patterns.

- **On-line** is an approximation using just one pattern:
  $\frac{\partial J}{\partial w_i} \approx -2 \cdot \text{(desired-actual)} \cdot x_i \cdot f'(\sum w_j x_j)$
Inside one neuron

Here $n$ is used for “net” value, as it is sometimes called.

\[
\frac{\partial J}{\partial w_i} = \left( \frac{\partial J}{\partial n} \right) \left( \frac{\partial n}{\partial w_i} \right)
\]

chain rule

\[
= \left( \frac{\partial (d-f(n))^2}{\partial v} \right) \left( \frac{\partial n}{\partial w_i} \right)
\]

\[
= -2 \varepsilon f'(n) x_i
\]

\[
= s \times x_i
\]

where $s = \left( \frac{\partial J}{\partial n} \right)$ is called the sensitivity
Chain Rule Refresher

The derivative of a composition is the product of the derivatives.

Example

\[
\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}
\]

\[
f(n) = \cos(n) \quad n = e^{2w} \quad f(n(w)) = \cos(e^{2w})
\]

\[
\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (-\sin(n))(2e^{2w}) = (-\sin(e^{2w}))(2e^{2w})
\]

Application to the Gradient Calculation

\[
\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial J}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m}
\]
Multi-Variate Chain Rule

The rule is shown here for 2-variable functions, but extends to n-variable functions straightforwardly.

Let \( x = x(t) \) and \( y = y(t) \) be differentiable at \( t \) and suppose that \( z = f(x, y) \) is differentiable at the point \( (x(t), y(t)) \). Then \( z = f(x(t), y(t)) \) is differentiable at \( t \) and

\[
\frac{dz}{dt} = \frac{\partial z}{\partial x} \frac{dx}{dt} + \frac{\partial z}{\partial y} \frac{dy}{dt}.
\]

http://www.math.hmc.edu/calculus/tutorials/multichainrule/
Example of Multi-Variate Chain Rule

Let \( z = x^2 y - y^2 \) where \( x \) and \( y \) are parametrized as \( x = t^2 \) and \( y = 2t \).

Then

\[
\frac{dz}{dt} = \frac{\partial z}{\partial x} \frac{dx}{dt} + \frac{\partial z}{\partial y} \frac{dy}{dt}
\]

\[
= (2xy)(2t) + (x^2 - 2y)(2)
\]

\[
= (2t^2 \cdot 2t)(2t) + \left((t^2)^2 - 2(2t)\right)(2)
\]

\[
= 8t^4 + 2t^4 - 8t
\]

\[
= 10t^4 - 8t.
\]

http://www.math.hmc.edu/calculus/tutorials/multichainrule/
Chain Rule in Vector Form

\[ \frac{dz}{dt} = \frac{\partial z}{\partial x} \frac{dx}{dt} + \frac{\partial z}{\partial y} \frac{dy}{dt} \]

RHS can be rewritten as an inner product

\[ [\partial z/\partial x, \partial z/\partial y] \cdot [\partial x/\partial t, \partial y/\partial t]' \]

(The left term is recognized as the gradient of \( z(x, y) \) wrt \( (x, y) \).)
Jacobian: Chain Rule in Vector Form

If $z$ were a vector-valued function, or $x, y, ...$ were functions of multiple variables, then the vectors on the previous page become matrices of partial derivatives: Jacobians.

The derivative of a composition of two multi-variate functions is the (matrix) product of the Jacobians, analogous to the chain rule for ordinary derivatives of univariate functions.
Jacobian matrix of function $F$

\[ J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} . \]

also notated:

\[
\frac{\partial (F_1, \ldots, F_m)}{\partial (x_1, \ldots, x_n)}
\]

Let $F$ be the transformation defined by

$$x(u,v) = u^2 - 3uv, \quad y(u,v) = u^3 + 5v^2$$

Find $J_F$ evaluated at the point (-1,2).

**Solution**

First find the Jacobian by calculating the partial derivatives.

$$J_F = \begin{pmatrix} x_u & x_v \\ y_u & y_v \end{pmatrix} = \begin{pmatrix} 2u - 3v & -3u \\ 3u^2 & 10v \end{pmatrix}$$

Now plug in (-1,2) to get:

$$J_F = \begin{pmatrix} -8 & 3 \\ 3 & 20 \end{pmatrix}$$

Finally, multiply by the column vector:

$$\begin{pmatrix} -8 & 3 \\ 3 & 20 \end{pmatrix} \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} (-8)(-1) + (3)(2) \\ (3)(-1) + (20)(2) \end{pmatrix} = \begin{pmatrix} 14 \\ 37 \end{pmatrix}$$

http://www.ltcconline.net/greenl/courses/202/multipleIntegration/Jacobians2DTheory.htm
Jacobian vs. Gradient

- Gradient is the vector-valued derivative of a scalar-valued function.

- Jacobian is the matrix-valued derivative of a vector-valued function.

- In effect, Jacobian is a stack of gradients, one for each component of the vector-valued function.
What does any of this have to do with neural networks?

The key to backpropagation is the multivariate (or vector) form of the chain rule.

With all training samples fixed, the error is a function of only the weights.
What does any of this have to do with neural networks?

More precisely, the error at the output layer is a function of the weights in the output layer and the inputs to the output layer (= outputs of the hidden layer).

The outputs of the hidden layer are a function of the weights in the hidden layer and the inputs to the hidden layer.
What does any of this have to do with neural networks?

Still more precisely, the error at the output layer is a function of the activation values at that layer which are inputs to the activation functions at that layer.

Those inputs are a function of the activation values at the previous layer, and so on.
Last Layer

n is the **activation** or “net” value

\[ n = \text{net} \]

\[ \text{output} = f(n) \]
Last Layer

Error is $J(n_1, n_2, \ldots, n_m)$

Note: $J$ does not stand for “Jacobian” here.
Next-to-Last Layer

Error is $J(n_1, n_2, \ldots, n_m)$

But each $n_i = J_i(n'_1, n'_2, \ldots, n'_k)$
Error is $J(n_1, n_2, \ldots, n_m)$

But each $n_i = J_i(n'_1, n'_2, \ldots, n'_k)$

Chain Rule says: $\nabla J(n'_1, n'_2, \ldots, n'_k) = \nabla J(n_1, n_2, \ldots, n_m) \cdot [\nabla J_1(n'_1, n'_2, \ldots, n'_k), \nabla J_2(n'_1, n'_2, \ldots, n'_k), \ldots, \nabla J_m(n'_1, n'_2, \ldots, n'_k)]^T$

This one equation tells us how to get the needed derivatives for backpropagation.
Utility of Sensitivities $s$ in Deriving Gradient wrt Weights

$$\frac{\partial J}{\partial w_i} = (\frac{\partial J}{\partial n}) (\frac{\partial n}{\partial w_i})$$

$$= (\frac{\partial J}{\partial n}) \frac{\partial}{\partial w_i} (w_1 x_1 + w_2 x_2 + \ldots + w_k x_k)$$

$$= (\frac{\partial J}{\partial n}) x_i$$

$$= s x_i$$

$s = \frac{\partial J}{\partial n}$ is meaningful at any net value $n$ in the network, not just at the final layer.

There is one $s$ value per neuron.

The gradients wrt weights are derivable from the sensitivities using this equation.
e.g. in Next-Last Layer

\[
\frac{\partial J}{\partial w_i} = \left( \frac{\partial J}{\partial n} \right) \left( \frac{\partial n}{\partial w_i} \right)
= s \, x_i \quad (s \text{ is "sensitivity"})
\]
How to compute sensitivity $\partial J/\partial n_i'$ from sensitivities $\partial J/\partial n_1$, $\ldots$, $\partial J/\partial n_m$:

We know $J = J(n_1, n_2, \ldots, n_m)$. But each $n_i$ is a function of $n_1', n_2', \ldots, n_k'$.

The chain rule thus says

$$\frac{\partial J}{\partial n_j'} = \frac{\partial J}{\partial n_1} \cdot \frac{\partial n_1}{\partial n_j'} + \frac{\partial J}{\partial n_2} \cdot \frac{\partial n_2}{\partial n_j'} + \cdots + \frac{\partial J}{\partial n_m} \cdot \frac{\partial n_m}{\partial n_j'}.$$

But $n_i = w_{i1} f(n_1') + w_{i2} f(n_2') + \cdots + w_{ik} f(n_k')$ from the network structure.

So $\frac{\partial n_i}{\partial n_j'} = \frac{\partial}{\partial n_j'} (w_{i1} f(n_1') + w_{i2} f(n_2') + \cdots + w_{ik} f(n_k'))$

$$= w_{ij} f(n_j') \quad \text{(every term drops except the } j^{\text{th}} \text{)}$$

Thus $\frac{\partial J}{\partial n_j'} = \frac{\partial J}{\partial n_1} \cdot w_{1j} f(n_j') + \frac{\partial J}{\partial n_2} \cdot w_{2j} f(n_j') + \cdots + \frac{\partial J}{\partial n_m} \cdot w_{mj} f(n_j')$

i.e. $\frac{\partial J}{\partial n_j'} = w_{1j} f(n_j') \cdot \frac{\partial J}{\partial n_1} + w_{2j} f(n_j') \cdot \frac{\partial J}{\partial n_2} + \cdots + w_{mj} f(n_j') \cdot \frac{\partial J}{\partial n_m}$

(The $w$'s are the $j^{\text{th}}$ column of the weight matrix.)
Backward Propagation of Sensitivity

sensitivities

known

desired

sensitivities

w_i
Express desired sensitivity as a weighted sum of known sensitivities:

\[ s_i = f'(n) \sum w_j s_j \]

In effect, this uses the weight matrix transposed.

Biases do not play a role in this step.
3-Layer Net from Text (Fig. 6.4)
MSE = J = E[(desired-actual)^2] (The author multiples by a factor of 1/2 and does not average.) so gets, at the outputs $z_k$:

$$-\frac{\partial J}{\partial u_{ki}} = y_i (d_k - z_k) f'(\sum u_{ki} y_i) = y_i g_k$$

where variable $g_k = (d_k - z_k) f'(\sum u_{ki} y_i)$ is introduced and represents the **sensitivity** at the $k^{th}$ output unit.

This is implied in B.6.5.5, $\Delta u_{ki} = a g_k y_i$ (a is the learning rate).

Note: The subscripts $j$, $i$, and $k$ identify the units at the input, hidden, and output layers respectively.
● Using the chain rule, the sensitivities at the hidden layer $i$ are derived:

$$g_i = f'(\Sigma v_{ij} x_j) \Sigma g_k u_{ki}$$

This is B.6.5.11.

● The use of the above is in 6.5.4, the weight change for the hidden layer:

$$\Delta v_{ij} = a g_i x_j$$ (a is the learning rate).

● Again: The subscripts $j$, $i$, and $k$ identify the units at the input, hidden, and output layers respectively.
6.5.1 defines the error function.

6.5.2 and 6.5.3 define weight adjustments as negatives of gradients wrt weights.

6.5.4 and 6.5.5 expand 6.5.2 and 6.5.3 to use sensitivities $g_i$ and $g_k$ times unit inputs.

6.5.6 Derives output layer sensitivities $g_k$.

6.5.7-6.5.11 use the chain rule to derive hidden layer sensitivities $g_i$. 
Textbook’s Backpropagation Script backpropTrain.m is keyed to Fig. 6.4

Training is on-line, not batch.

TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 6.4
\[ a = 0.1; \]
\[ \text{tol} = 0.1; \]
\[ b = 1; \]
\[ \text{nIts} = 100000; \]
\[ \text{nHid} = 1; \]
\[ [\text{nPat}, \text{nIn}] = \text{size} (\text{InPat}); \]
\[ [\text{nPat}, \text{nOut}] = \text{size} (\text{DesOut}); \]
\[ \text{V} = \text{rand} (\text{nHid}, \text{nIn} + 1) \times 2 - 1; \]
\[ \text{U} = \text{rand} (\text{nOut}, \text{nHid} + 1) \times 2 - 1; \]
\[ \text{deltaV} = \text{zeros} (\text{nHid}, \text{nIn} + 1); \]
\[ \text{deltaU} = \text{zeros} (\text{nOut}, \text{nHid} + 1); \]
\[ \text{maxErr} = 10; \]

\[ \text{for} \; \text{c} = 1: \text{nIts}, \]
\[ \text{pIndx} = \text{ceil} (\text{rand} \times \text{nPat}); \]
\[ \text{d} = \text{DesOut} (\text{pIndx},:); \]
\[ \text{x} = [\text{InPat} (\text{pIndx},:) \; \text{b}]; \]
\[ \text{y} = 1. / (1 + \exp (-\text{V} \times \text{x})); \]
\[ \text{z} = 1. / (1 + \exp (-\text{U} \times \text{y})); \]
\[ \text{e} = \text{d} - \text{z}'; \]
\[ \text{if} \; \text{max} (\text{abs} (\text{e})) > \text{tol}, \]
\[ \text{x} = \text{x}'; \text{y} = \text{y}'; \text{z} = \text{z}'; \]
\[ \text{zg} = \text{e} \times (\text{z} \times (1 - \text{z})); \]
\[ \text{yg} = (\text{y} \times (1 - \text{y})) \times (\text{zg} \times \text{U}); \]
\[ \text{deltaU} = \text{a} \times \text{zg} \times \text{y}; \]
\[ \text{deltaV} = \text{a} \times \text{yg} (1: \text{nHid})' \times \text{x}; \]
\[ \text{U} = \text{U} + \text{deltaU}; \]
\[ \text{V} = \text{V} + \text{deltaV}; \]
\[ \text{end} \]
\[ \text{end} \]
### Important Steps

#### Input

\[
p\text{Indx} = \text{ceil}(\text{rand}\times n\text{Pat});
\]
\[
d = \text{DesOut}(p\text{Indx},:);
\]
\[
x = [\text{InPat}(p\text{Indx},:) \ b]';
\]

% choose pattern pair at random
% set desired output to chosen output
% append the bias to the input vector

#### Forward

\[
y = 1./(1+\exp(-V*x));
\]
\[
y = [y' \ b]';
\]
\[
z = 1./(1+\exp(-U*y));
\]
\[
e = d-z';
\]

% compute the hidden unit response
% append the bias to the hidden unit vector
% compute the output unit response
% find the error vector

#### Backward

\[
x = x'; y = y'; z = z';
\]
\[
zg = e.\ast(z.\ast(1-z));
\]
\[
yg = (y.\ast(1-y)).\ast(zg\ast U);
\]

% convert column to row vectors
% compute the output error signal
% compute hidden error signal

#### Adjust

\[
deltaU = a\ast zg'\ast y;
\]
\[
deltaV = a\ast yg(l:nHid)'\ast x;
\]
\[
U = U + \delta U;
\]
\[
V = V + \delta V;
\]

% compute the change in hidden-output weights
% compute change in input-hidden weights
% update the hidden-output weights
% update the input-hidden weights
Examples

- Try with xor, parity, etc.
Backward Propagation of Sensitivity Matrix Version

Express desired sensitivity as a weighted sum of known sensitivities:

\[ s_i = f'(n) \sum w_j s_j \]

If \( W \) is the weight matrix of the latter layer, and \( s \) is the corresponding vector of sensitivities, then the vector \( s^* \) of sensitivities at the former layer is computable by

\[ s^* = f'(n) \cdot (W^T s) \]

where \( \cdot \) is pointwise multiplication
Alternate Matrix-Vector Form expressed using dot for derivatives

Express desired as a weighted sum of known:

\[ s^m = \Phi^m (n^m) (W^{m+1})^T s^{m+1} \]

Note: over-dot means derivative.
Correctness

\[ s = f'(n) \sum w_j s_j \]

layer \( m+1 \) is layer after

\[ s = \frac{\partial J}{\partial n^m} \]

layer \( m \) is layer before

\[ = \sum (\frac{\partial n^{m+1}}{\partial n^m}) (\frac{\partial J}{\partial n^{m+1}}) \]

the vector form of the chain rule

\[ \mathbf{s}^m = \frac{\partial J}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial J}{\partial \mathbf{n}^{m+1}} \]

Vector Form for entire \( m^{th} \) layer:

\[ \mathbf{s}^m = \mathbf{F}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1} \]
In the vector form, the Jacobian matrix is given by:

$$ s^m = \frac{\partial J}{\partial n^m} = \left( \frac{\partial n^{m+1}}{\partial n^m} \right)^T \frac{\partial J}{\partial n^{m+1}} $$

where $s^m$ is the $s$ subscripts here refer to the size of the layer and are not related to sensitivities.
Vector Form

\[
s^m = \frac{\partial J}{\partial n^m} = \left(\frac{\partial n^{m+1}}{\partial n^m}\right)^T \frac{\partial J}{\partial n^{m+1}} = F^m(n^m)\left(W^{m+1}\right)^T \frac{\partial J}{\partial n^{m+1}}
\]

\[
s^m = F^m(n^m)\left(W^{m+1}\right)^T s^{m+1}
\]

\[
\frac{\partial n^{m+1}}{\partial n^m} = W^{m+1} F^m(n^m) \\
\hat{F}^m(n^m) = \begin{bmatrix}
    f^m(n_1^m) & 0 & \ldots & 0 \\
    0 & f^m(n_2^m) & \ldots & 0 \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \ldots & f^m(n_{s_m}^m)
\end{bmatrix}
\]
Fully-Subscripted Alternatives to the Vector Forms

\[ n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j} a_j^{m-1} + b_i^m \]

\[ \frac{\partial n_i^m}{\partial w_{i,j}} = a_j^{m-1} \quad \frac{\partial n_i^m}{\partial b_i^m} = 1 \]

Sensitivity

\[ s_i^m \equiv \frac{\partial J}{\partial n_i^m} \]

Gradient

\[ \frac{\partial J}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \quad \frac{\partial J}{\partial b_i^m} = s_i^m \]
Fully-Subscripted Alternatives to the Vector Forms

\[
\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial}{\partial n_j^m} \left( \sum_{l=1}^{S^m} w_{i,l}^{m+1} a_i^m + b_i^{m+1} \right) = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m}
\]

\[
\frac{\partial n_i^{m+1}}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1} f_j^m(n_j^m)
\]

\[
f_j^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}
\]
Vector Form for Last Layer, $M$

\[ s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (t - a)^T (t - a)}{\partial n_i^M} = \frac{\partial}{\partial n_i^M} \sum_{j=1}^{S^M} (t_j - a_j)^2 = 2(t_i - a_i) \frac{\partial a_i}{\partial n_i^M} \]

\[ \frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = f_i^M(n_i^M) \]

\[ s_i^M = -2(t_i - a_i) f_i^M(n_i^M) \]

\[ s^M = -2 \hat{F}^M(n^M)(t - a) \]
Backpropagation Training Cycle

- **Forward propagation**: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.

- **Compute the error** in the output.

- **Backpropagate** the error through the network to get “sensitivities” at each neuron. (The gradient approximation is derivable from the sensitivities.)

- Use the sensitivities to **derive weight changes**.

- Apply the weight changes.
Backpropagation (Sensitivities)

The sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer.

\[
\begin{align*}
\mathbf{s}^M & \rightarrow \mathbf{s}^{M-1} \rightarrow \ldots \rightarrow \mathbf{s}^2 \rightarrow \mathbf{s}^1 \\
\mathbf{s}^M &= -2 \mathbf{F}'^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}) \quad \text{basis} \\
\mathbf{s}^m &= \mathbf{F}'^m(\mathbf{n}^m)(\mathbf{W}^m + 1, \mathbf{T}^m m + 1) \mathbf{s}^{m+1} \quad \text{induction step}
\end{align*}
\]

\[
\mathbf{F}'^m(\mathbf{n}^m) = 
\begin{bmatrix}
    f'^m(n_1^m) & 0 & \ldots & 0 \\
    0 & f'^m(n_2^m) & \ldots & 0 \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \ldots & f'^m(n_{Sm}^m)
\end{bmatrix}
\]

diagonal matrix of activation function derivative values
Weight and Bias Update

(Here we are using $\alpha$ instead of $\eta$ for the learning rate.)

$$ W^m(k+1) = W^m(k) - \alpha s^m (a^{m-1})^T $$

Bias update is parallel to the above

$$ b^m(k+1) = b^m(k) - \alpha s^m $$
Fully-Subscripted Version of Weight Update

\[ w_{i,j}^m(k + 1) = w_{i,j}^m(k) - \alpha s_i a_j^{m-1} \]
\[ b_i^m(k + 1) = b_i^m(k) - \alpha s_i \]

\[ W^m(k + 1) = W^m(k) - \alpha s^m a^{m-1}^T \]
\[ b^m(k + 1) = b^m(k) - \alpha s^m \]

\[ s^m = \frac{\partial J}{\partial n^m} = \begin{bmatrix}
\frac{\partial J}{\partial n_1^m} \\
\frac{\partial J}{\partial n_2^m} \\
\vdots \\
\frac{\partial J}{\partial n_{S^m}^m}
\end{bmatrix} \]
Backpropagation Summary

Forward Propagation from Input Pattern:

\[ a^0 = p \]

\[ a^{m+1} = f^{m+1}(W^{m+1} a^m) \quad m = 0, 2, \ldots, M - 1 \]

\[ a = a^M \]

Backpropagation from Error:

\[ s^M = -2 \hat{F}^M(n^M)(t - a) \]

\[ s^m = \hat{F}^m(n^m)(W^{m+1})^T s^{m+1} \quad m = M - 1, \ldots, 2, 1 \]

Weight Update

\[ W^m(k + 1) = W^m(k) - \alpha s^m (a^{m-1})^T \]

\[ b^m(k + 1) = b^m(k) - \alpha s^m \]
BP Calculation Demo nnd11bc

Input: \( p = 1.0 \)
Target: \( t = 1.0 \sin(p / 4) = 1.707 \)

Simulate:
\[
\begin{align*}
a_1 &= \log\sin(W_1p + b_1) = [0.321, 0.368] \\
a_2 &= \text{purelin}(W_2a_1 + b_2) = 0.446 \\
e &= t - a_2 = 1.281
\end{align*}
\]

Backpropagate:
\( s_2 = ? \)
\( s_1 = ? \)

Update:
\( W_1 = ? \)
\( b_1 = ? \)
\( W_2 = ? \)
\( b_2 = ? \)
Worked Numeric Example from nnd11nf

1-2-1 Network

Input → Log-Sigmoid Layer → Linear Layer → $a$
Initial Conditions

\[
\text{Initial Conditions}
\]

\[
\begin{align*}
\mathbf{W}^1(0) &= \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} & \mathbf{b}^1(0) &= \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} & \mathbf{W}^2(0) &= \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} & \mathbf{b}^2(0) &= \begin{bmatrix} 0.48 \end{bmatrix}
\end{align*}
\]
Forward Propagation

\[ a^0 = p = 1 \]

\[ a^1 = f^1(W^1 a^0 + b^1) = \text{logsig} \left( \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} a^0 + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} \right) = \text{logsig} \left( \begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix} \right) \]

\[
\begin{bmatrix}
\frac{1}{1 + e^{-0.75}} \\
\frac{1}{1 + e^{-0.54}}
\end{bmatrix}
= \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}
\]

\[ a^2 = f^2(W^2 a^1 + b^2) = \text{purelin} \left( \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix} + \begin{bmatrix} 0.48 \end{bmatrix} \right) = \begin{bmatrix} 0.446 \end{bmatrix} \]

\[ e = t - a = \left\{ 1 + \sin \left( \frac{\pi}{4} p \right) \right\} - a^2 = \left\{ 1 + \sin \left( \frac{\pi}{4} 1 \right) \right\} - 0.446 = 1.261 \]
Transfer Function Derivatives

\[ f^1(n) = \frac{d}{dn} \left( \frac{1}{1 + e^{-n}} \right) = \frac{e^{-n}}{(1 + e^{-n})^2} = \left( 1 - \frac{1}{1 + e^{-n}} \right) \left( \frac{1}{1 + e^{-n}} \right) = (1 - a^1)(a^1) \]

\[ f^2(n) = \frac{d}{dn}(n) = 1 \]
Backpropagation of Sensitivities

\[ s^2 = -2 \dot{F}^2(n^2)(t - a) = -2 \left[ f^2(n^2) \right](1.261) = -2 \left[ 1 \right](1.261) = -2.522 \]

\[ s^1 = F^1(n)(W^2)^T s^2 = \begin{bmatrix} (1 - a_1^1)(a_1^1) & 0 \\ 0 & (1 - a_2^1)(a_2^1) \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} [-2.522] \]

\[ s^1 = \begin{bmatrix} (1 - 0.321)(0.321) \\ 0 \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} [-2.522] \]

\[ s^1 = \begin{bmatrix} 0.218 & 0 \\ 0 & 0.233 \end{bmatrix} \begin{bmatrix} -0.227 \\ 0.429 \end{bmatrix} = \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \]
Weight Update

\[ \alpha = 0.1 \]

\[ W^2(1) = W^2(0) - \alpha s^2(a^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \\ 0.321 \end{bmatrix} \begin{bmatrix} 0.368 \end{bmatrix} \]

\[ W^2(1) = \begin{bmatrix} 0.171 \\ -0.0772 \end{bmatrix} \]

\[ b^2(1) = b^2(0) - \alpha s^2 = \begin{bmatrix} 0.48 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} = \begin{bmatrix} 0.732 \end{bmatrix} \]

\[ W^1(1) = W^1(0) - \alpha s^1(a^0)^T = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix} \]

\[ b^1(1) = b^1(0) - \alpha s^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix} \]
Exercise

- Derive the backprop equations symbolically for a simple 2-layer network.

- Then use the equations to train the network.
Label the Levels

0 1 M = 2
Label the Signal Vectors or Lines

Vectors, superscript = level
Label the Signal Vectors or Lines

\[ a^0_1 \quad a^0_2 \quad a^1_1 \quad a^1_2 \quad a^2_1 \]

Lines, superscript = level,
Label the Net (Activation) Values
Label the Weights and Biases
Write the forward equations for activations

\[
\begin{align*}
n_1 &= w_{11} a_1 + w_{12} a_2 + b_1 \\
a_1 &= f(n_1) \\
n_2 &= w_{21} a_1 + w_{22} a_2 + b_2 \\
a_2 &= f(n_2) \\
n_1 &= w_{11} a_1 + w_{12} a_2 + b_1 \\
a_1 &= f(n_1) \\
n_2 &= w_{21} a_1 + w_{22} a_2 + b_2 \\
a_2 &= f(n_2)
\end{align*}
\]
Write the backward equations for sensitivities

\[ s^1_1 = w^{211}_1 s^2_1 f'(n^1_1) \]

\[ s^1_2 = w^{212}_1 s^2_1 f'(n^1_2) \]

\[ s^2_1 = -2(d^2_1 - a^2_1) f'(n^2_1) \]
The summations for the backpropagated sensitivities have only one term in this example, since the following layer has only one neuron.

Try working it out for, say, three neurons in the last layer.
Write the Equations for Weight and Bias Update

\[
\begin{align*}
\Delta w_{11} &= -\alpha s_1 a_1^0 \\
\Delta w_{12} &= -\alpha s_1 a_2^0 \\
\Delta b_1 &= -\alpha s_1 \\
\Delta w_{21} &= -\alpha s_2 a_1^1 \\
\Delta w_{22} &= -\alpha s_2 a_2^1 \\
\Delta b_2 &= -\alpha s_2
\end{align*}
\]
Note on Training vs. Use

- Discontinuous functions such as hardlim, hardlims, etc. don’t have derivatives.

- Therefore we train the network with continuous approximations to these functions, then replace them with the discontinuous versions during usage:
  - usage: hardlim, hardlims
  - train with: logsig, tansig
Cybenko’s Universal Approximation Theorem


Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of $n$ real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.
Cybenko’s Universal Approximation Theorem

Definition. We say that $\sigma$ is sigmoidal if

$$\sigma(t) \to \begin{cases} 1 & \text{as } t \to +\infty, \\ 0 & \text{as } t \to -\infty. \end{cases}$$

Theorem 1. Let $\sigma$ be any continuous discriminatory function. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$  (2)

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \varepsilon \quad \text{for all } x \in I_n.$$
Worst-Case Difficulty of Backprop

TRAINING A 3-NODE NEURAL NETWORK IS NP-COMPLETE

Avrim L. Blum*
MIT Laboratory for Computer Science
Cambridge, Mass. 02139 USA

Ronald L. Rivest†
MIT Laboratory for Computer Science
Cambridge, Mass. 02139 USA

Copyright 1992 Pergamon Press plc

Figure 1: The 3-Node Network.
Reduction is from Set-Splitting

The following problem, *Set-Splitting*, was proven to be NP-complete by Lovász (Garey and Johnson, 1979).

“Given a finite set $S$ and a collection $C$ of subsets $c_i$ of $S$, do there exist disjoint sets $S_1$, $S_2$ such that $S_1 \cup S_2 = S$ and $\forall i, c_i \not\subset S_1$ and $c_i \not\subset S_2$?”

The Set-Splitting problem is also known as 2-non-Monotone Colorability or Hypergraph 2-colorability. Our use of this problem is inspired by its use by Kearns, Li, Pitt, and Valiant to show that learning k-term DNF is NP-complete (Kearns et al., 1987) and the style of the reduction is similar.
We show for many simple two-layer networks whose nodes compute linear threshold functions of their inputs that training is NP-complete. For any training algorithm for one of these networks there will be some sets of training data on which it performs poorly, either by running for more than an amount of time polynomial in the input length, or by producing sub-optimal weights. Thus, these networks differ fundamentally from the perceptron in a worst-case computational sense.

The theorems and proofs are in a sense fragile; they do not imply that training is necessarily hard for networks other than those specifically mentioned. They do, however, suggest that one cannot escape computational difficulties simply by considering only very simple or very regular networks.

On a somewhat more positive note, we present two networks such that the second is both more powerful than the first and can be trained in polynomial time, even though the first is NP-complete to train. This shows that computational intractability does not depend directly on network power and provides theoretical support for the idea that finding an appropriate network and input encoding for one’s training problem is an important part of the training process.

An open problem is whether the NP-completeness results can be extended to neural networks that use the differentiable logistic linear functions. We conjecture that training remains NP-complete when these functions are used since it does not seem their use should too greatly alter the expressive power of a neural network (though Sontag (1989) has demonstrated some important differences between such functions and thresholds). Note that Judd (1990), for the networks he considers, shows NP-completeness for a wide variety of node functions including logistic linear functions.