Neural Networks

Joey White

Hephaestus Audio

The future of audio processing will move beyond the simple linear processing provided by the mainstays of digital filtering: the FIR and the IIR filter. *Neural networks* (NN's) contain all of the processing capability of linear digital filters and far more. Not only do they provide the ability to model arbitrary time-varying nonlinear functions, but also they are able to adapt in real time via a multitude of learning algorithms.

Feed-Forward Neural Networks

Feed-Forward Neural Networks (FFNN's) are the neural network "equivalent" of the FIR filter. Only feedforward terms are used (i.e. there is no feedback), so the transfer function is static. In other words only a static mapping of input to output is provided, although this mapping may now be nonlinear.

A *Time-Delay Neural Network* (TDNN) employs a tapped delay line, as does an FIR filter. This allows for the modeling of a *frequency-dependent* nonlinear function. As with a typical FFNN, *Backpropagation* is employed for training of the network. The only real difference is that the input vector now represents a time series.

Recurrent Neural Networks

Recurrent Neural Networks (RNN's) are the neural network "equivalent" of the IIR filter. As with an IIR filter there is great computational ability, however stability and training may be an issue. *Real Time Recurrent Learning* is one method of training full-connected RNN's, however it is computationally intensive and often yields slow convergence. Other training methods are available, however the application of RNN's is still relatively new and the algorithms are not "tried-and-true" as with FFNN's.

Expert System versus Adaptive System

Standard linear digital filters are examples of building block that may be used as part of an *expert system*. They must be set based upon knowledge of the system in question. Neural networks are examples of building block that may be used as part of an *adaptive system*, where the underlying nature of the system in question is not necessarily known, or perhaps not time-invariant.

Expert System

Pros:

- Simple to understand.
  - E.g. if the output should be a little higher at a certain level with a given load, then increase the input signal a small amount to compensate.
- Predictable in the sense that the parameters are fixed (it is time-invariant).

Cons:

- Must be tailored to a specific system.
- Requires knowledge of the system for all configurations and operating conditions.
- Any parameters that cause a change in the operation of the system must be measured and, via expert knowledge, compensated for.
  - E.g. temperature, load, component tolerance, etc.

Adaptive System

Pros:

- Does not require expert knowledge of the system.
- Can account for unforeseen variations in operating conditions.
- Can be used across a broad range of products with no modification.
  - E.g., it is not limited to a specific amplifier, or for that matter, amplifiers at all. It could be used for amplifier and loudspeaker combinations, etc.

Cons:

- Expert knowledge of the system under consideration is not required, however expert knowledge of adaptive systems is.
- The dynamics are not time-invariant, so care must be taken that the performance of the system is not somehow made worse.
  - This could take the form of a "cap" on the maximum error that should be corrected – e.g. the amplifier is allowed to clip without any attempt to "correct" it.

Backpropagation Algorithm for FFNN's

Transfer function – given the current set of network parameters, process the current input(s):

x_i	i^{th} Input node state	$i = 1,2,...,m$
$s_j^y = b_j^y + \sum_{i=1}^{m} \omega_{ji}^{yx} x_i$		
$y_j = f(s_j^y)$	j^{th} Hidden node state	$j = 1,2,...,n$
$s_k^z = \sum_{j=1}^{n} \omega_{kj}^{zy} y_j$		
$z_k = s_k^z$	k^{th} Output node state	$k = 1,2,..., p$
ω_{ji}^{yx}	Weight from node x_i to node y_j	
b_j^y	Bias for node y_j	
ω_{kj}^{zy}	Weight from node y_j to node z_k	

Parameter update – based upon the error of the resulting output, refine the network parameters:

μ	Gradient descent constant	
t_k	k^{th} Output node target	$k = 1,2,..., p$
$E = \frac{1}{2}\sum_{k=1}^{p}(t_k - z_k)^2$	Cost function to minimize	
$\alpha_j^y = \frac{\partial f(s_j^y)}{\partial s_j^y}$	Activation function derivative	
$\Delta\omega_{ji}^{yx} = \mu\alpha_j^y\left(\sum_{k=1}^{p}\omega_{kj}^{zy} e_k\right)x_i$	Weight update – input layer	
$\Delta b_j^y = \mu\alpha_j^y\left(\sum_{k=1}^{p}\omega_{kj}^{zy} e_k\right)$	Bias update – hidden node	
$\Delta\omega_{kj}^{zy} = \mu e_k y_j$	Weight update – output layer	

Real Time Recurrent Learning Algorithm for RNN's

Transfer function – given the current set of network parameters, process the current input(s):

$$y_i^{k+1} = f\left(x_i^k + b_i + \sum_{j=1}^{m} \omega_{ij} y_i^k \right)$$ i^{th} node state at time $k+1$

ω_{ij} Weight *from* node y_j^k *to* node y_i^k

b_i Bias for node y_i^k

x_i^k Input to node y_i^k

Parameter update – based upon the error of the resulting output, refine the network parameters:

μ Gradient descent constant

t_i^k Target output for node y_i^k

$$E^k = \frac{1}{2} \sum_{i=1}^{n} (t_i^k - y_i^k)^2$$ Cost function to minimize

$$\Delta\omega_{\alpha\beta} = \mu \cdot \sum_{i=1}^{n} \left((t_i^k - y_i^k) \cdot \frac{\partial y_i^k}{\partial \omega_{\alpha\beta}} \right)$$ Weight delta update

$$\frac{\partial y_i^k}{\partial \omega_{\alpha\beta}} = f'(net_i^{k-1}) \cdot \left(\delta_{i\alpha} y_\beta^{k-1} + \sum_{j=1}^{n} \left(\omega_{ij} \cdot \frac{\partial y_j^{k-1}}{\partial \omega_{\alpha\beta}} \right) \right)$$ Weight sensitivity expression

$$\Delta b_\alpha = \mu \cdot \sum_{i=1}^{n} \left((t_i^k - y_i^k) \cdot \frac{\partial y_i^k}{\partial b_\alpha} \right)$$ Bias delta update

$$\frac{\partial y_i^k}{\partial b_\alpha} = f'(net_i^{k-1}) \cdot \left(\delta_{i\alpha} + \sum_{j=1}^{n} \left(\omega_{ij} \cdot \frac{\partial y_j^{k-1}}{\partial b_\alpha} \right) \right)$$ Bias sensitivity expression