netcom

Collection Editor:
shekhar maravi

netcom

Collection Editor:

shekhar maravi

Authors:

Don Johnson
Phil Repicky
Jacob Schrum

Online:

< http://cnx.org/content/col11181/1.1/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

Intoduction

1.1 Network architectures and interconnection[1]

The network structure—its architecture[2]—typifies what are known as **wide area networks** (WANs). The nodes, and users for that matter, are spread geographically over long distances. "Long" has no precise definition, and is intended to suggest that the communication links vary widely. The Internet is certainly the largest WAN, spanning the entire earth and beyond. **Local area networks**, LANs, employ a single communication link and special routing. Perhaps the best known LAN is Ethernet[3]. LANs connect to other LANs and to wide area networks through special nodes known as **gateways** (Figure 1.1). In the Internet, a computer's address consists of a four byte sequence, which is known as its **IP address** (Internet Protocol address). An example address is **128.42.4.32**: each byte is separated by a period. The first two bytes specify the computer's **domain** (here Rice University). Computers are also addressed by a more human-readable form: a sequence of alphabetic abbreviations representing institution, type of institution, and computer name. A given computer has both names (**128.42.4.32** is the same as **soma.rice.edu**). Data transmission on the Internet requires the numerical form. So-called **name servers** translate between alphabetic and numerical forms, and the transmitting computer requests this translation before the message is sent to the network.

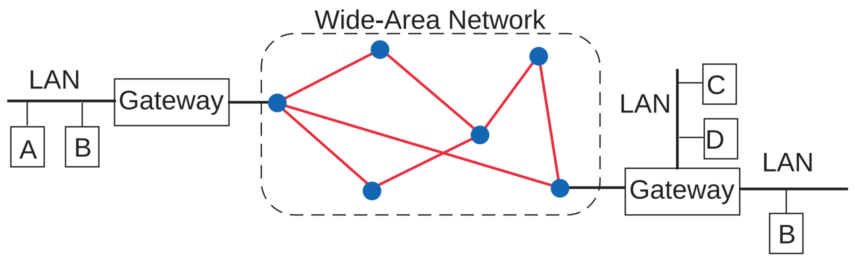

Figure 1.1: The gateway serves as an interface between local area networks and the Internet. The two shown here translate between LAN and WAN protocols; one of these also interfaces between two LANs, presumably because together the two LANs would be geographically too dispersed.

[1]This content is available online at <http://cnx.org/content/m0077/2.10/>.
[2]"Communication Networks", Figure 1 <http://cnx.org/content/m0075/latest/#commnetwork>
[3]"Ethernet" <http://cnx.org/content/m0078/latest/>

1.2 A Neural Network Model in ACL2[4]

Introduction

Artificial neural networks are biologically inspired tools that have proven useful in control tasks, pattern recognition, data mining, and other tasks. At an abstract level, a neural network is a tool for function approximation, and the ability of a special class of neural networks, namely multiplayer perceptrons, to approximate any function from an m-dimensional unit hyper cube to an n-dimensional unit hyper cube with arbitrary accuracy is actually proven by the universal approximation theorem, provided some mild constraints are met (Haykin 1999). Furthermore, such a function can be approximated using a multiplayer perceptron with a single hidden layer. Unfortunately, this theorem merely assures the existence of such a network, and does not provide a means of attaining it – a task that is left to one of the many diverse learning algorithms for multiplayer perceptrons.

The most popular learning algorithm for multiplayer perceptrons is the back-propagation algorithm, which attempts to find the best set of synaptic weights for a fixed network architecture (Haykin 1999), though there are also methods that adjust the network architecture during learning, such as cascade correlation (Fahlman 1991), and methods that search the space of available network architectures, such as some neuroevolution algorithms (Gomez 1997, Stanley 2004). These algorithms often deal with perceptrons having more than one hidden layer, in spite of the universal approximation theorem's assurance that a single hidden layer is sufficient. This is done because solutions with more hidden layers also exist, and may in fact be easier to discover with a given learning algorithm.

We therefore have a situation where a variety of network architectures represent the same function. Many undesirable functions will likewise have multiple representations within the space of architectures, and this can be problematic if architectures representing the desired solution do not densely populate the space. The ratio of desired architectures to undesired architectures in the space may drastically decrease as the size of the space increases. Even if this is not the case, the computational expense of searching a larger space can often be restrictive. Even when working with a fixed architecture, the organization of synaptic weights can give rise to equivalent networks, so this is actually a problem that affects any learning algorithm for multiplayer perceptrons.

This problem is addressed in this paper via the creation of a neural network model that allows one to prove whether or not a given set of networks is equivalent in terms of their input-output mappings. A model of multiplayer perceptrons is developed in ACL2, and several theorems about transformations on the networks that maintain the same input-output mapping are presented.

Model

The neural networks represented by this model are fully-connected feed-forward networks, also known as multiplayer perceptrons. Each network consists of at least one layer. The first layer is always the input layer, and the last layer is always the output layer (even if these two are the same). Any and all layers in between are known as hidden layers.

The basic unit of a layer is a neuron, which consists of several synapses, each with its own synaptic weight. In a computer, a synaptic weight is usually specified by a floating point value, but since ACL2 only supports rational numbers, a synapse in this model is represented by a rational number. The associated predicate is isSynapse. This means a neuron is represented by a list of such synapses. We also require that each neuron have at least one synapse. Both of these conditions are checked by the isNeuron function.

A single layer of a network is represented by a list of neurons. For consistency, all neurons within a given layer must have the same number of synapses. Furthermore, a layer must have at least one neuron. The isLayer function checks these conditions. Having defined a layer, we now define a network as a list of layers. The network must contain at least one layer, where the first layer of the list is the input layer, and the last layer of the list is the output layer. Furthermore, the number of outputs in each layer must match the number of inputs in the next. This means that for all layers but the output layer, the length of the layer equals the length shared by all neurons in the following layer. The predicate that identifies networks is isNetwork.

[4]This content is available online at <http://cnx.org/content/m15107/1.1/>.

Because it is challenging to specify a suitable structure meeting these requirements without making a mistake, there is a function makeNetwork that takes a flat list of synaptic weights and weaves them into the appropriate structure. It makes use of a function makeLayer, which does the same for layers. One may assume that organizing weights in such a list is almost as difficult to do properly as organizing the network itself, but since neural networks are often initialized with random weights prior to training, it simplifies matters to have a function that creates a proper network from a flat list of weights, the length of which may actually be longer than is necessary: as long as enough weights are in the list, it does not matter if there are extras. Besides the list of weights, the makeNetwork function also needs a list specifying the number of inputs to each layer of the network in sequence. The last number in this list is the number of outputs from the output layer. Likewise, every call to makeLayer must include the number of inputs to the layer and the number of outputs.

```
    Theorem 1: calling makeLayer with valid inputs (a list of weights
and two non-zero natural numbers) returns a valid layer recognized
by isLayer.
```

```
    Theorem 2: calling makeNetwork with valid inputs (a list of
non-zero natural numbers and a list of weights) returns a valid
network recognized by isNetwork.
```

Besides modeling the structure of neural networks, we also model their input-output behavior. A valid input vector is a list of rational numbers, and is identified by the isListOfWeights function, so called because it is also a helper function for isNeuron. Like neurons, input vectors are lists of rational numbers. A single neuron functions by outputting the sum of products of each input value with the corresponding weight on the synapse it is transmitted through. The behavior is handled by forwardNeuron, which takes an input vector and neuron as input, and returns the weighted sum as output.

The value outputted from each individual neuron is then passed through an activation function, which is normally defined as any real-valued function. Once again, use of ACL2 restricts us to rational numbers. Although it is not a requirement, activation functions also commonly have a restricted range, usually [0,1] or [-1,1]. The three activation functions modeled herein share the range [-1,1]: threshold, piecewise, and rational sigmoid approximation.

`(threshold x)`	`(if (equal x 0) 0 (if (< 0 x) 1 -1))`
`(piecewise x)`	`(if (<= x -1) -1 (if (<= 1 x) 1 x))`
`(rationalsigmoid x)`	`(/ (* 6 x) (+ 12 (* x x)))`

Table 1.1

The rationalsigmoid function was used because ACL2 does not support irrational numbers, and therefore does not support the more common full sigmoid function: `(exp(x) - 1)/(exp(x) + 1)`. The function presented here is a rational valued approximation of the sigmoid function using 2-2 Pade approximation (Boskovic 1997). This activation function does not meet the requirements of the basic universal approximation theorem, but additional work has shown that rational functions such as the one above do in fact satisfy the universal approximation property (Kainen 1994). Neither the threshold function nor piecewise function fulfill the universal approximation requirements, but are none-the-less common.

The activate function models the behavior of activation functions by applying forwardNeuron to the input vector and neuron, and then applying the appropriate activation function specified by a quoted symbol to the result. The activate function is used by the forwardLayer function, which models signal propagation through layers, and this function is in turn used by forwardNetwork, which models signal propagation through networks.

Structural Swapping

One of the main focuses of this paper is to provide proofs of the effects that several simple structural transformations have on neural networks. Many functions are defined to perform these transformations, and one of the most widely used is swap. The swap function takes a list and two indices in the list, and returns a list with the values at those two positions swapped. To assure the correctness of swap, several theorems are proven about it:

```
    Theorem 3: swap is commutative, meaning that swapping positions i
and j is the same as swapping positions j and i.
```

```
    Theorem 4: swap has an identity, in that swapping position i with
position i in x is equal to x.
```

```
    Theorem 5: swap is its own inverse, in that swapping positions i
and j twice returns the original list.
```

With the swap function, the structure of a neural network can be modified in many ways. We start first at the level of neurons and then work our way up to full networks.

First we prove that swapping the same positions in both a neuron and its input vector does not change the output. This makes sense, since a neuron outputs a weighted sum, and addition is both commutative and associative.

```
    Theorem 6: the output from a neuron is the same as the output generated
when the same positions are swapped in both the input vector and the
neuron's list of synapses.
```

```
    (implies (and (isNeuron n)
                  (isListOfWeights i)
                  (equal (len n) (len i))
                  (natp k)
                  (< k (len i))
                  (natp j)
                  (< j (len i)) )
         (equal (forwardNeuron i n)
                (forwardNeuron (swap i j k)
                               (swap n j k) ))
```

Since the activate function simply applies an extra function to the output of forwardNeuron, this leads directly to the following theorem, which can be considered a corollary.

```
    Theorem 7: the activation of a neuron is the same as the activation
generated when the same positions are swapped in both the input vector
and the neuron's list of synapses.
```

```
    (implies (and (isNeuron n)
                  (isListOfWeights i)
                  (equal (len n) (len i))
                  (natp k)
                  (< k (len i))
                  (natp j)
                  (< j (len i)) )
         (equal (activate f i n)
                (activate f (swap i j k)
                            (swap n j k)) ) )
```

Moving up to the level of layers, we observe that swapping neurons in a layer only affects the order of outputs from the layer, because the order of synapses in each neuron of the layer remains the same, as shown in figure 1. This fact is proven by two similar theorems, the second of which is presented in its entirety below.

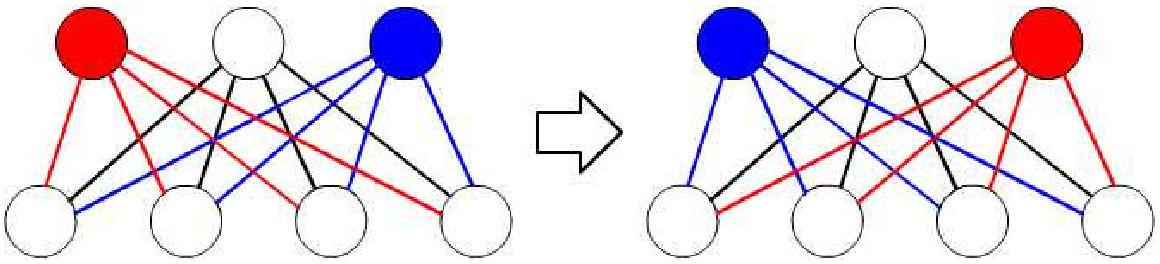

Figure 1.2: A single layer with neurons swapped. The bottom row represents inputs to the layer.

```
    Theorem 8: The output of a layer with two of its neurons swapped equals
the result of swapping those same positions in the unaltered layer's output.

    Theorem 9: The output of a layer remains the same when the same positions are
swapped in both the layer and its final output vector.

    (implies (and (isLayer l)
                  (isListOfWeights i)
                  (equal (len (car l)) (len i))
                  (natp k)
                  (< k (len l))
                  (natp j)
                  (< j (len l)) )
         (equal (forwardLayer f i l)
                (swap (forwardLayer f i (swap l j k)) j k)) )
```

Proving further facts about manipulations at the layer level requires an extra function, namely swapNeuron-SynapsesInLayer, which swaps the synapses at the same two positions for each neuron in a given layer. The effect of this function on a network layer is demonstrated in figure 2.

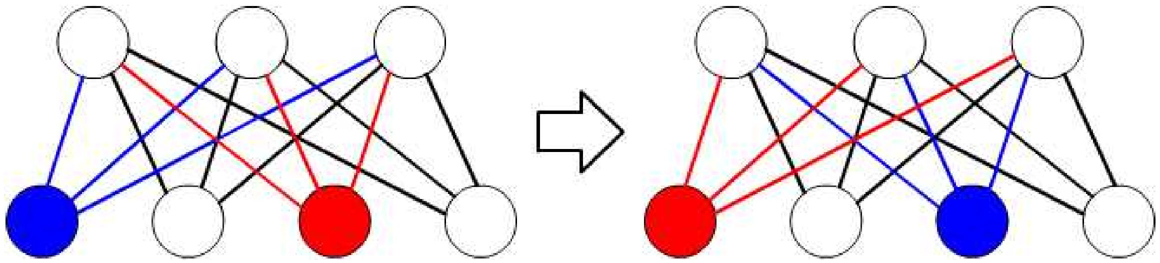

Figure 1.3: A single layer with synapses of each neuron swapped. The bottom row represents inputs to the layer, which must also be swapped.

Use of this function allows us to prove the following theorem:

```
    Theorem 10: the output of a layer remains the same if the same two positions are
swapped in all neurons of a layer, along with the same two positions in the
input vector.


    (implies (and (isLayer l)
                  (isListOfWeights i)
                  (equal (len (car l)) (len i))
                  (natp k)
                  (< k (len (car l)))
                  (natp j)
                  (< j (len (car l))) )
         (equal (forwardLayer f i l)
                (forwardLayer f (swap i j k)
                                (swapNeuronSynapsesInLayer l j k)) ) )
```

At the network level we finally see instances of identical input-output mappings for different network structures. Doing so requires yet another function, swapLayerNeuronsInNetwork, which swaps two neurons within the same layer of a network. Additionally, the function also applies swapNeuronSynapsesInLayer to the following layer (provided there is one) so that the neurons that were swapped still output to the same synapses in the following layer. As long as the neuron swap occurs in a layer before the final layer, this operation does not change the output that a neural network produces.

```
    Theorem 11: swapping the positions of two neurons in either the input layer or a hidden
layer of a network with two or more layers, and also swapping the corresponding synapses
in each neuron of the following hidden layer, leaves the output of a network
completely unchanged.


    (implies (and (isNetwork n)
                  (isListOfWeights i)
                  (equal (len (caar n)) (len i))
                  (natp l)
                  (< l (- (len n) 1))
                  (natp k)
                  (< k (len (nth l n)))
                  (natp j)
                  (< j (len (nth l n))) )
         (equal (forwardNetwork f i n)
                (forwardNetwork f i (swapLayerNeuronsInNetwork n l j k)) ) )
```

When the swapLayerNeuronsInNetwork function is applied to the output layer of a network, then the same positions in the output vector must be swapped to obtain the original output.

```
    Theorem 12: swapping the positions of two neurons in the output layer of a network leaves the
resulting output the same, except that the resulting output also has its values swapped at
the same two positions.


    (implies (and (isNetwork n)
                  (isListOfWeights i)
                  (equal (len (caar n)) (len i))
                  (natp k)
                  (< k (len (nth (- (len n) 1) n)))
```

```
              (natp j)
              (< j (len (nth (- (len n) 1) n))) )
        (equal (forwardNetwork f i n)
               (swap
                  (forwardNetwork f i
                                  (swapLayerNeuronsInNetwork n (- (len n) 1) j k))
                  j k) ) )
```

The set of proofs described so far essentially capture the idea of interchange equivalence (Kainen 1994). Permuting neurons of a network within their hidden layers, and modifying the synaptic connections in the next layer accordingly, produces interchange equivalent networks.

Dead Neurons

Besides rearranging existing components of a neural network, we can also introduce and eliminate certain components. There are certain cases when these operations have no effect on the behavior of the neural network, and the most readily identifiable of these cases involve dead neurons, which shall be defined to be neurons with nothing but zeroes for all synaptic weights. It therefore makes intuitive sense that such neurons transmit no data, and can therefore be freely added to and removed from a neural network without affecting its behavior. Dead neurons have a recognizer function isDeadNeuron and a constructor function makeDeadNeuron, which constructs a dead neuron with a given number of synapses. It should be noted that the following theorems about dead neurons depend on the fact that all available activation functions map zero to zero. This is commonly true for activation function in the range [-1,1], but is not a hard requirement for activation functions, which can be any real-valued function. If the value of a given activation function at zero was not zero, then the above definition of a dead neuron would not make sense, because information would in fact be transmitted.

In order to insert dead neurons into a network, we define a more general function that inserts any given neuron into a network. The addNeuronToLayerOfNetwork, function adds a neuron to the front of a specified layer in a network. Due to theorems 11 and 12 above, we know that it is sufficient to have a function that only adds a new neuron to the front of a layer, since we could use any given number of swap operations to reorganize the layer into any order we wanted without changing the fundamental input-output behavior of the network. However, it is important that the new neuron have the same size as all the neurons in the layer into which it is being inserted.

When adding a new neuron to a layer, we must also add new connections between that neuron and the neurons of the following layer (unless there is none). Therefore besides taking a neuron, a network and the index of a layer as input, the addNeuronToLayerOfNetwork function also takes a list of weights as input. Each neuron of the following layer receives one synaptic weight from this list. The addSynapsesToNeuronsOfLayer helper function performs this operation and assures that the newly inserted synapses line up with the newly added neuron. Given these functions, we can now describe the effect of adding a dead neuron to a neural network.

```
    Theorem 13: Adding a dead neuron to the input layer or a hidden layer of a network with at
least 2 layers does not change its output.


    (implies (and (isNetwork net)
                  (< 0 (len i))
                  (isListOfWeights i)
                  (isListOfWeights s)
                  (equal (len (nth (+ 1 l) net)) (len s))
                  (equal (len i) (len (caar net)))
                  (natp l)
                  (< l (- (len net) 1)) )
        (equal (forwardNetwork f i net)
               (forwardNetwork
                    f i (addNeuronToLayerOfNetwork
                            (makeDeadNeuron (len (car (nth l net)))) s l net)) ) )
```

`Theorem 14: Adding a dead neuron to the output layer of a network only changes the output in that it now contains an additional value, which is 0.`

```
    (implies (and (isNetwork net)
                  (< 0 (len i))
                  (isListOfWeights i)
                  (isListOfWeights s)
                  (equal (len i) (len (caar net))) )
         (equal (cons 0 (forwardNetwork f i net))
                (forwardNetwork
                  f i (addNeuronToLayerOfNetwork
                         (makeDeadNeuron (len (car (nth (- (len net) 1) net))))
                         s (- (len net) 1) net)) ) )
```

We can link the concept of dead neurons to lesioning/pruning, which is the destruction of select neurons of a neural network. The function that removes a neuron from a network is removeFromNetwork, which takes a network, the index of a layer within the network, and the index of a neuron within that layer as input. When removing a neuron from the network, we must also adjust the synapses of neurons in the next layer so that they still match up. The removeFromNetwork function calls repairRemainder to perform these adjustments, which are only necessary if the neuron was not removed from the output layer. It should be noted that the result of removeFromNetwork is only a valid network if the layer from which the neuron was removed had at least two neurons. Otherwise the layer would be empty after the removal, making the network invalid.

With these functions we can now prove that removing a dead neuron from a non-output layer of a network has no effect on its output. This theorem is the mirror to theorem 13.

`Theorem 15: removing a dead neuron from the input layer or a hidden layer in the network with at least 2 layers does not change its output.`

```
    (implies (and (isNetwork n)
                  (isDeadNeuron (nth j (nth l n)))
                  (isListOfWeights i)
                  (equal (len i) (len (caar n)))
                  (<= 2 (len (car (nth (+ 1 l) n))))
                  (natp l)
                  (natp j)
                  (< l (- (len n) 1))
                  (<= 2 (len (nth l n)))
                  (< j (len (nth l n))) )
         (equal (forwardNetwork f i n)
                (forwardNetwork f i (removeFromNetwork l j n))) )
```

It makes intuitive sense that the effect of lesioning/pruning can also be achieved by simply replacing the neuron marked for removal with a dead neuron, and this is the next theorem that we will prove. Once again, we must first build up some functions for replacement of neurons with other neurons in a network. We therefore define replaceNeuronInLayerOfNetwork, which takes as input a network, the index of a layer in the network, the index of a neuron within that layer, and a neuron designated to replace the neuron at that position. The resulting network is only valid if the new neuron is the same size as the neuron it replaces. Given this new function, we now state and prove the desired theorem.

```
 Theorem 16: removing a neuron from the input layer or a hidden layer in a network with at least 2
layers is equivalent to replacing it with a dead neuron.

 (implies (and (isNetwork n)
               (isListOfWeights i)
               (natp l)
               (natp j)
               (< l (- (len n) 1))
               (< j (len (nth l n)))
               (<= 2 (len (nth l n)))
               (equal (len i) (len (caar n))) )
      (equal (forwardNetwork f i (removeFromNetwork l j n))
             (forwardNetwork
                  f i (replaceNeuronInLayerOfNetwork
                          j l (makeDeadNeuron (len (nth j (nth l n)))) n)) ) )
```

There are several neural network algorithms that depend on lesioning, such as the optimal brain damage algorithm (Russell 2003) and the Enforced Subpopulation neuroevolution algorithm (Gomez 1997). Either the removal or replacement method can be used to perform the lesioning, though often both are used in the following way: first the normal activations of several candidate neurons for lesioning are replaced one at a time with zeroes, effectively replacing the neuron with an equal sized dead neuron; then the neuron whose lesioning resulted in the best increase in performance is actually removed from the network. Theorem 16 validates this methodology, by showing that the two operations result in equivalent network outputs.

Conclusion and Future Work

The functions presented in this paper provide a useful model of fully-connected feed-forward neural networks in ACL2. The theorems presented have shown that networks with widely different representations are in fact equivalent in terms of their input-output mappings. This is the first step towards creating neural network search algorithms that search in a reduced space with fewer or no duplicate network representations of the same input-output mapping.

One method for doing this might involve deciding on some canonical form for neural network representations, and then creating a search algorithm that only searched the space of canonical forms. The validity of such an algorithm could be proven via the following means:

First define a function, say canonicalp, that identified whether or not a network was in canonical form. One would need to show that each transformation that could be performed on a neural network during the search would return a network for which canonicalp would return t.

The validity of such an algorithm would also depend on whether or not this canonical form was actually canonical. That is, one would need to define a function, say toCanonical, and prove that for every value recognized by isNetwork, the toCanonical function would return a value recognized by canonicalp. The theorems provided in this paper would likely be very useful in such a proof. Another theorem that would be required to demonstrate that a given form was actually canonical would be a theorem proving that two canonical forms with the same input-output mapping are in fact equal in accordance with the ACL2 equal function.

This model could also be extended to model existing learning/search algorithms for neural networks. One extension that would be required to model certain algorithms is addition of recurrent network connections. Another extension that would be even more challenging to integrate would be support for networks of arbitrary connectivity. Such support would perhaps require a completely different model. However, even if this were the case, equivalence between networks of arbitrary connectivity and the fully-connected networks of this model could perhaps be proven in ACL2. For example, it seems likely that any gaps in a sparsely connected neural network could be represented in a fully-connected network by dead neurons. Naturally, the many theorems and lemmas about dead neurons created as part of this model would be helpful in proving such a fact. Furthermore, if in the arbitrarily connected model a synapse skipped over two layers, this could perhaps be modeled in the fully-connected model by the introduction of "identity neurons", which could be defined as neurons where one synapse had a weight of 1, and for which the identity function was used instead of the normal activation function. Therefore a single synapse over two layers would be represented as two synapses, one of which had the same weight as the original, and the other of which would be the weight 1 synapse of an identity neuron.

Clearly, many extensions are possible. This model provides a good basis for studying some low level aspects of the theory behind neural networks, especially concerning their structure. Furthermore, the functions provided serve as a good start towards modeling any of several neural network search algorithms. Adding neurons, replacing existing neurons (or individual synapses) and removing neurons are basic low level operations needed for searching the space of neural network architectures. The hope is that this model can serve as a platform for stud of such extended models.

Bibliography

Jovan D. Boskovic: A stable neural network-based adaptive control scheme for a class of nonlinear plants, Proceedings of the 36th Conference on Decision & Control: 472-477, 1997.

Scott E. Fahlman, Christian Lebiere: The Cascade-Correlation Learning Architecture, Report CMU-CS-90-100, School of Computer Science at Carnegie Mellon University, Pittsburgh, PA, 1991.

F. Gomez, R. Miikkulainen: Incremental Evolution of Complex General Behavior, Adaptive Behavior 5:317–342, 1997. Also Available from http://nn.cs.utexas.edu/.

Simon Haykin: Neural Networks: A Comprehensive Foundation. Prentice Hall, 1999.

P. C. Kainen, V. Kurková, V. Kreinovich, O. Sirisengtaksin: Uniqueness of network parameterization and faster learning, Neural, Parallel and Scientific Computations 2: 459-466, 1994.

Stuart J. Russell, Peter Norvig: Artificial Intelligence: A Modern Approach. Prentice Hall, 2003.

K. O. Stanley, R. Miikkulainen: Competitive Coevolution through Evolutionary Complexification, Journal of Artificial Intelligence Research 21:63-100, 2004.

1.3 Neural Network Design[5]

To implement our neural network we used the Neural Network Toolbox in MATLAB. The neural network is built up of layers of neurons. Each neuron can either accept a vector or scalar input (p) and gives a scalar output (a). The inputs are weighted by W and given a bias b. This results in the inputs becoming Wp + b. The neuron transfer function operates on this value to generate the final scalar output a.

[5]This content is available online at <http://cnx.org/content/m13228/1.1/>.

A MATLAB Neuron that Accepts a Vector Input

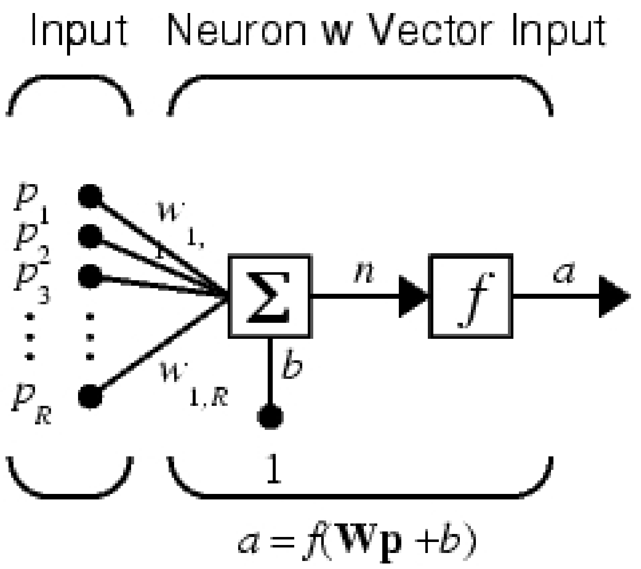

Figure 1.4

Our network used three layers of neurons, one of which is required by the toolbox. The final layer, output layer, is required to have neurons equal to the size of the output. We tested five accents, so our final layer has 5 neurons. We also added two "hidden" layers, which operate on the inputs before they are prepared as outputs, each of which have 20 neurons.

In addition to configuring the network parameters, we had to build the network training set. In our training set we had 42 speakers: 8 Northern, 9 Texan, 9 Russian, 9 Farsi, and 7 Mandarin. An accent profile was created for each of these speakers as discussed and compacted into a matrix. Each profile was a column vector, so the size was 42 x 28. For each speaker we also generated an answer vector. For example, the desired answer for a Texan accent is [0 1 0 0 0]. These answer vectors were also combined into an answer matrix. The training matrix and the desired answer matrix were given to the neural network which trained using traingda (gradient descent with adaptive learning rate backpropogation). We set the goal for the training function to be a mean square error of .005.

We originally configured our neural network to use neurons with a linear transfer function (purelin), however when using more than three accents at a time we could not reduce the mean square error to .005 The error approached a limit, which increased as the number of accents we included increased.

Linear Neuron Transfer Function

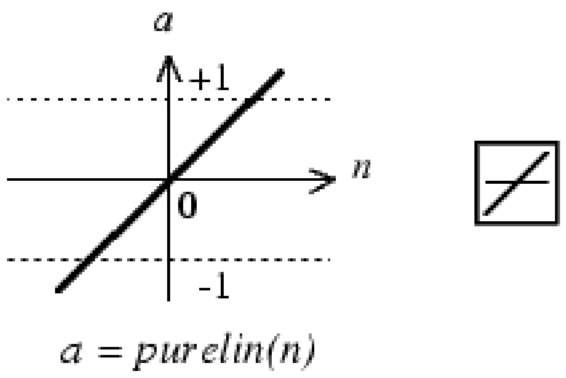

Figure 1.5

Linear Neurons Training Curve

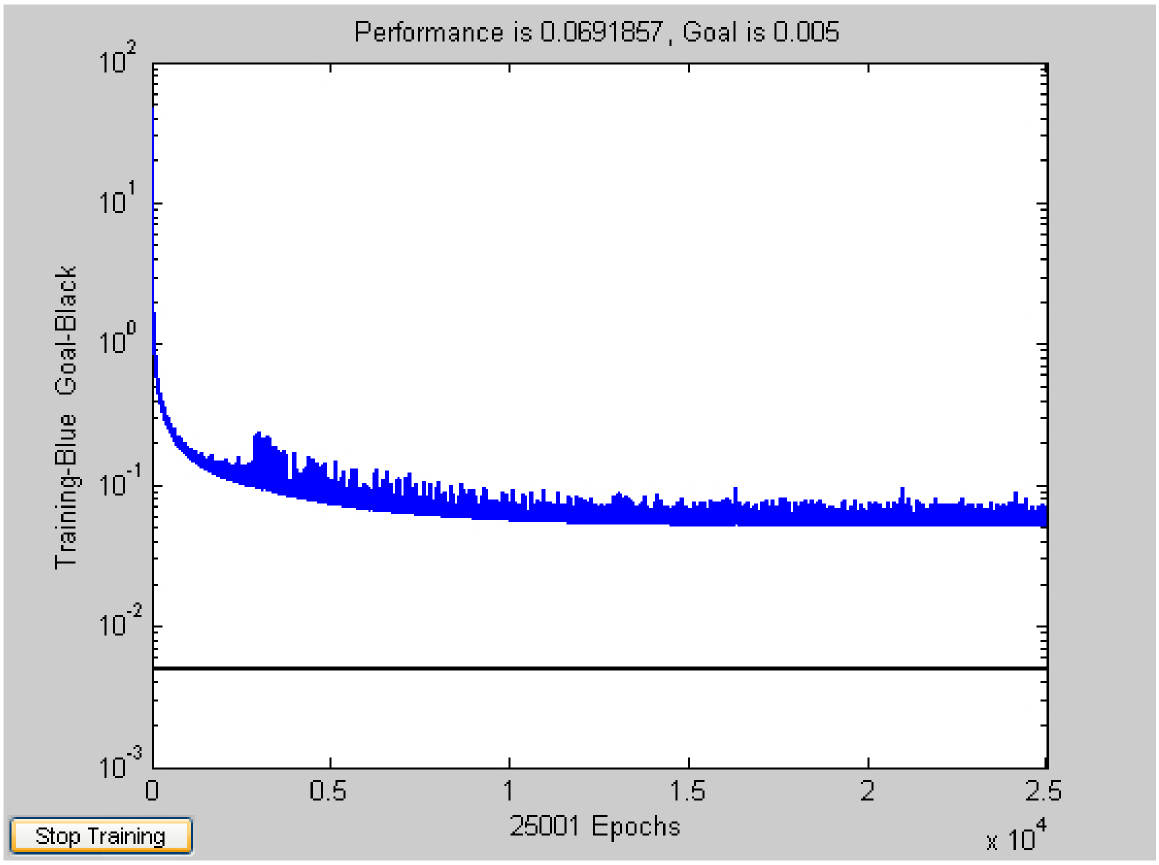

Figure 1.6

So, at this point we redesigned our network to use non-linear neurons (tansig).

Tansig Neuron Transfer Function

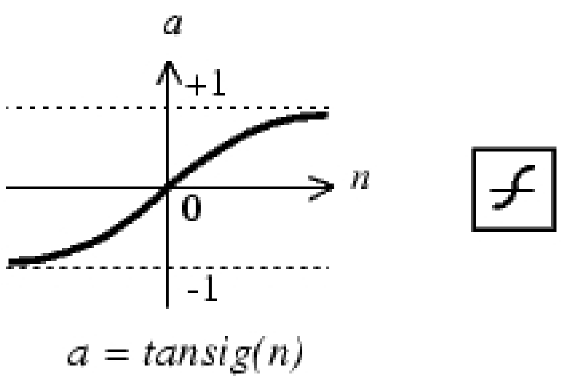

Figure 1.7

Tansig Neurons Training Curve

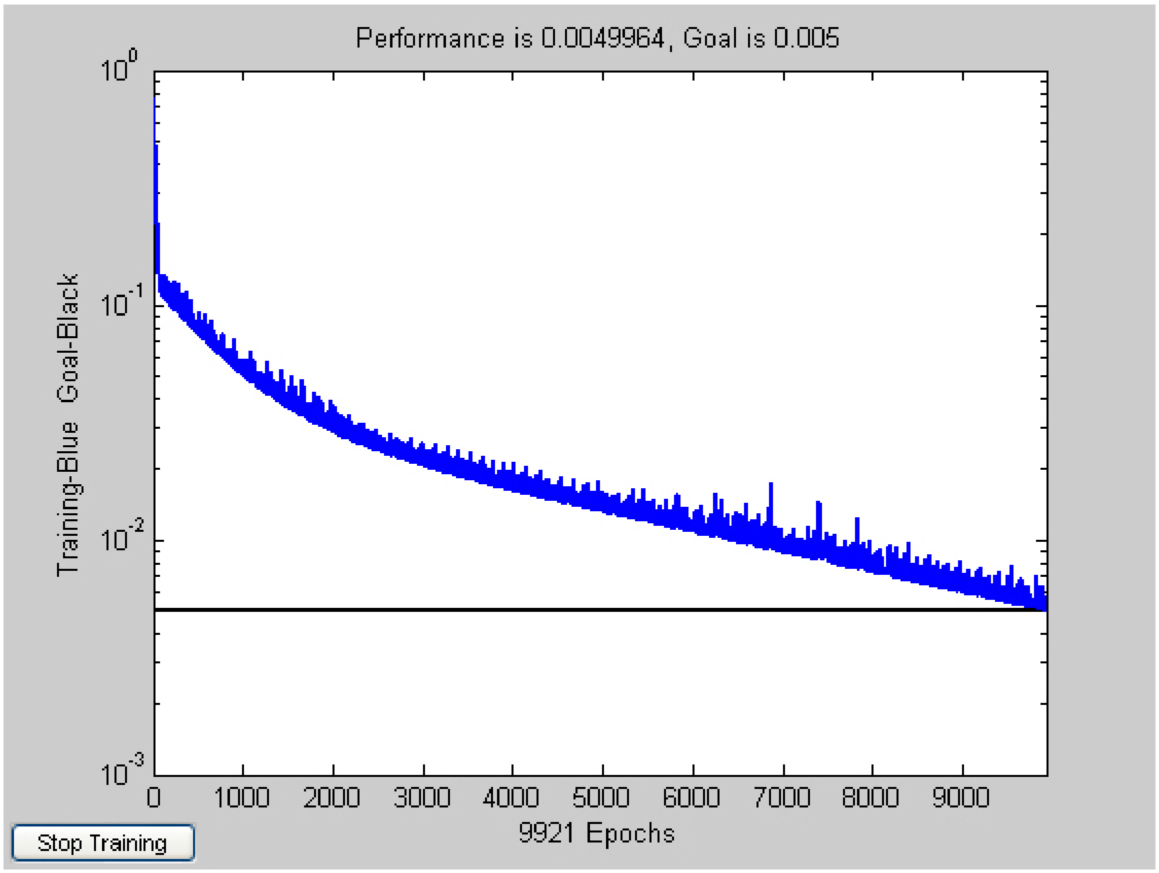

Figure 1.8

After the network was trained we refined our set of training samples by looking at the network's output when given the training matrix again. We removed a handful of speakers to arrive at our present number of 42 because they included an accent we weren't explicitly testing for. These consisted of speakers who sounded as if they did not learn American English, but British English.

These final two figures show an image representation of the answer matrix and the answers given by the trained matrix. In the images, grey is 0 and white is one. Colors darker than grey represent negative numbers.

Answer Matrix

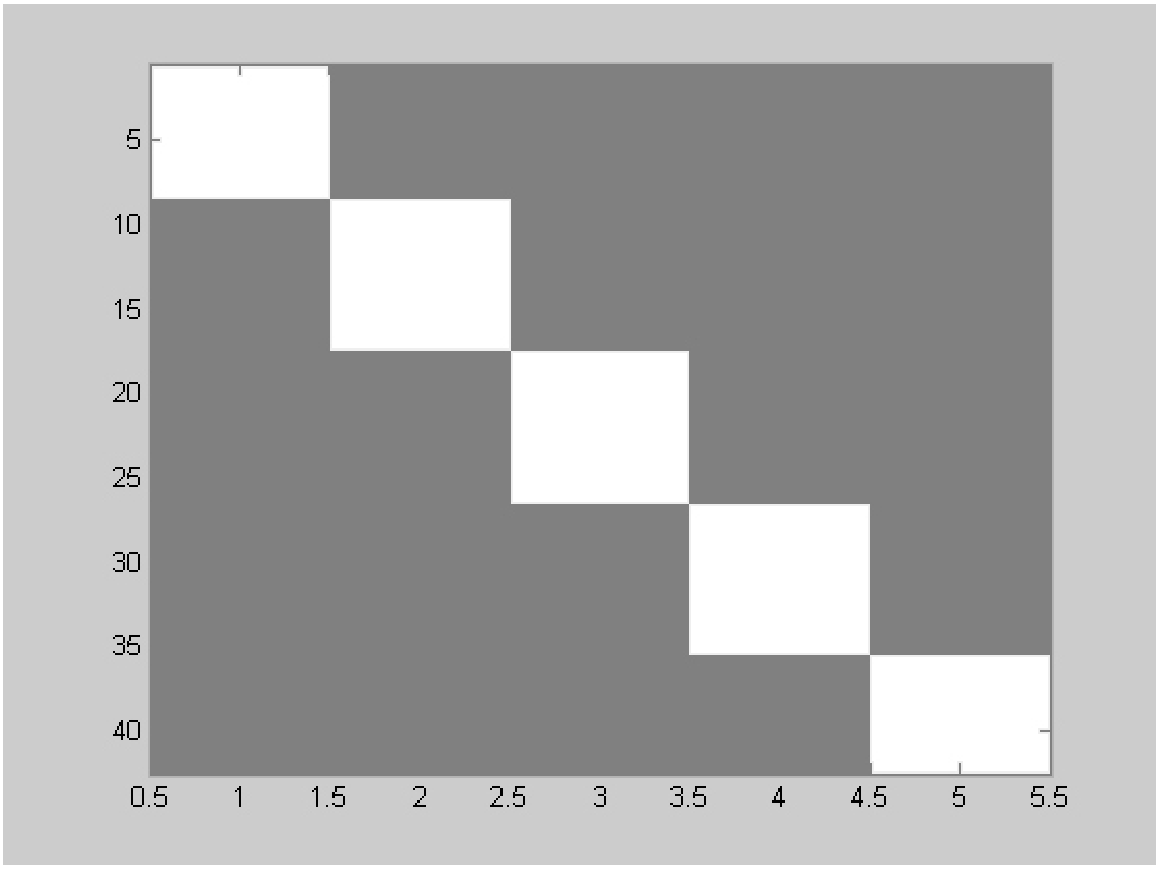

Figure 1.9

Trained Answers

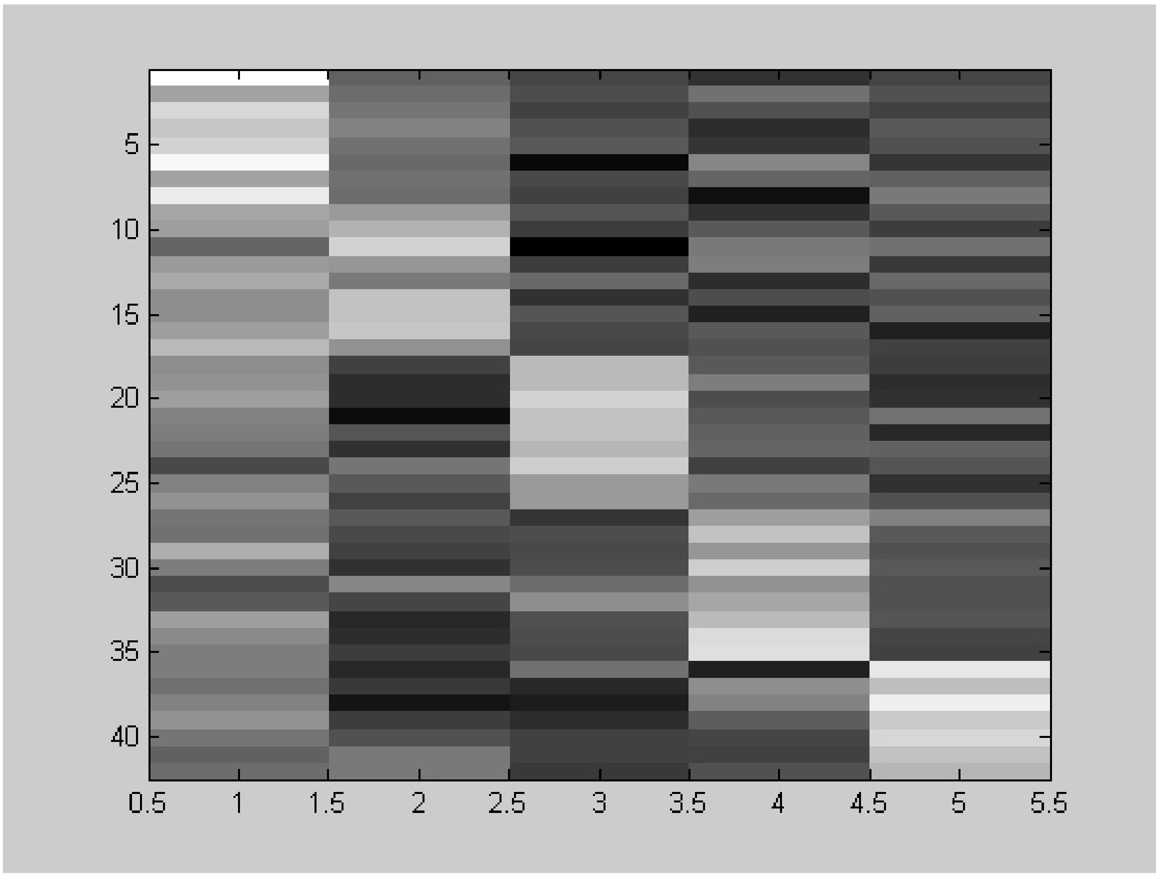

Figure 1.10

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

netcom
network theory