

A Neural Network Model in ACL2

By Jacob Schrum (schrum2@cs.utexas.edu)

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`.

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`.
 ("layers.lisp")

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`.
 ("networks.lisp")

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.

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

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 . ("swap.lisp")

Theorem 4: swap has an identity, in that swapping position i with position i in x is equal to x . ("swap.lisp")

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

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. ("form1.lisp")

```
(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. ("form1.lisp")

```
(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.

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. ("form4.lisp")

```
(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. ("form4.lisp")

```
(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. ("dead.lisp")

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

```
(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. ("prune.lisp")

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

Appendix

```
****
neurons.lisp
****
; Neurons Book
(in-package "ACL2")

;-----

; An activation function takes a rational number as input and
; returns a rational number as output.

; The threshold function returns -1, 0, or 1 for negative values, zero, and
; positive values respectively.

(defun threshold (x) (if (equal x 0) 0 (if (< 0 x) 1 -1)))

; The piecewise activation function returns the input value within the
; range [-1,1], but returns 1 for values greater than 1 and -1 for values
; less than -1.

(defun piecewise (x)
  (if (<= x -1)
      -1
      (if (<= 1 x)
          1
          x)))

; The rationalsigmoid function is a rational valued approximation of the
; standard real-valued sigmoid function.

(defun rationalsigmoid (x) (/ (* 6 x) (+ 12 (* x x)) )

;-----

; Synapse - represented by a rational number for the synaptic weight.

(defun isSynapse (x) (rationalp x))

;-----

; Neuron - list of synapses entering the neuron.

(defun isListOfWeights (x)
  (if (consp x)
      (if (isSynapse (car x))
          (isListOfWeights (cdr x))
          nil)
      (not x)))

(defun isNeuron (x)
  (if (consp x)
      (if (isSynapse (car x))
          (isListOfWeights (cdr x))
          nil)
      nil))

; Signals are passed through neuron by multiplying incoming signals by
; synaptic weights and outputting the sum of these weighted signals.
; Function works for a list of inputs i with length equal to the number
; of synapses in neuron n.

(defun forwardNeuron (i n)
  (if (consp i)
      (if (consp n)
          (+ (* (car i) (car n))
              (forwardNeuron (cdr i) (cdr n)))
          0)
      0))
```

```

; Outputs of neurons are restricted by an activation function f.
; Function works for a list of inputs i with length equal to the number
; of synapses in neuron n, along with a function f.

(defun activate (f i n)
  (if (equal f `threshold)
      (threshold (forwardNeuron i n))
      (if (equal f `piecewise)
          (piecewise (forwardNeuron i n))
          (rationalsigmoid (forwardNeuron i n) ) ) ) )

;-----
****

layers.lisp
****

; Layers Book
(in-package "ACL2")

(include-book "neurons")

;-----

; Layer - list of neurons in the layer, where each neuron contains
; the same number of synapses. Layer must have at least one neuron.

(defun isLayer2 (x i)
  (if (consp x)
      (if (isNeuron (car x))
          (if (equal (len (car x)) i)
              (isLayer2 (cdr x) i)
              nil)
          nil)
      (not x)))

(defun isLayer (x)
  (if (consp x)
      (if (isNeuron (car x))
          (isLayer2 (cdr x) (len (car x)))
          nil)
      nil))

; Signals are passed through a layer by passing the same set of signals through
; each of its neurons. The result is a list of outputs, each coming from
; a neuron of the layer. Function works for a list of inputs i with length equal
; to the number of synapses in each neuron of layer l.

(defun forwardLayer (f i l)
  (if (consp l)
      (cons (activate f i (car l))
            (forwardLayer f i (cdr l)))
      nil))

; A layer can be made from a flat list of weights if the number of inputs
; and outputs (neurons) is specified. The makeLayer function works if the number
; of inputs i times the number of outputs o is the length of w, a list of
; synaptic weights.

(defun front (i w)
  (if (zp i)
      nil
      (cons (car w) (front (- i 1) (cdr w))) ) )

(defun after (i w)
  (if (zp i)
      w
      (after (- i 1) (cdr w)) ) )

(defun makeLayer (i o w)
  (if (zp o)

```

```

nil
  (cons (front i w) (makeLayer i (- o 1) (after i w))))
;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defthmd trans-<=
  (implies (and (<= a b) (<= b c)) (<= a c)) )

(defthmd len-1
  (iff (consp i)
    (<= 1 (len i)) ) )

(defthmd neuron-subset
  (implies (and (isListOfWeights w)
    (natp i)
    (<= 1 i)
    (<= i (len w)) )
    (isNeuron (front i w)) )

  :hints (("Goal"
    :in-theory (enable trans-<=
      len-1)
    :induct (front i w) ) ) )

(defthmd after-subset
  (implies (and (isListOfWeights w)
    (<= i (len w)) )
    (isListOfWeights (after i w)) )

  :hints (("Goal"
    :induct (after i w) ) ) )

(defthmd after-len
  (implies (and (natp i)
    (<= i (len w)) )
    (equal (len (after i w)) (- (len w) i)) )

  :hints (("Goal"
    :induct (after i w) ) ) )

(defthmd front-len
  (implies (and (<= i (len w))
    (natp i) )
    (equal (len (front i w)) i) )

  :hints (("Goal"
    :induct (front i w) ) ) )

(defun times (x y)
  (if (zp x) 0 (+ y (times (- x 1) y)) ) )

(defthmd times-*
  (implies (and (natp x) (natp y))
    (equal (* x y) (times x y)) ) )

(defthmd greater-times
  (implies (and (natp i)
    (natp o)
    (< 0 o) )
    (<= i (times i o)) ) )

(defthmd greater-*
  (implies (and (natp i)
    (natp o)
    (< 0 o) )
    (<= i (* i o)) )

  :hints (("Goal"

```

```

      :in-theory (enable times-*) ) ) )

(defthm inputs-within-weights
  (implies (and (natp i)
                (natp o)
                (< 0 o)
                (<= (* i o) (len w)) )
            (<= i (len w)) )

  :hints (("Goal"
           :use ((:instance greater-* (i i) (o o) )
                 (:instance trans-<= (a i) (b (* i o)) (c (len w)) ) ) ) ) )

(defthmd len-cdr
  (implies (< 0 (len w))
            (equal (len (cdr w)) (- (len w) 1)) ) )

(defthmd remove-neuron-from-layer
  (implies (and (<= (* i o) (len w))
                (natp i)
                (natp o)
                (< 0 i)
                (< 0 o) )
            (<= (* i (- o 1)) (len (after i w)) ) )

  :hints (("Goal"
           :use ((:instance len-cdr (w w) )
                 (:instance after-len (i (- i 1)) (w (cdr w)) )
                 (:instance times-* (x o) (y i) ) ) ) ) )

(defthmd makeLayer-isLayer2
  (implies (and (isListOfWeights w)
                (<= (* i o) (len w))
                (natp i)
                (natp o)
                (< 0 i)
                (< 0 o) )
            (isLayer2 (makeLayer i o w) i) )

  :hints (("Goal"
           :induct (makeLayer i o w)
           :in-theory (enable inputs-within-weights
                              remove-neuron-from-layer
                              neuron-subset
                              after-subset
                              after-len
                              front-len) ) ) )

(defthmd layer-size-inequality
  (implies (and (<= (* i o) (len w))
                (natp o)
                (natp i)
                (< 0 o)
                (< 0 i) )
            (not (< (len (after i w)) (- (* i o) i) ) ) )

  :hints (("Goal"
           :in-theory (enable after-len) ) ) )

(defthmd neg-*
  (equal (- a b) (+ (* -1 b) a)))

(defthmd layer-size-inequality-2
  (implies (and (<= (* i o) (len w))
                (natp o)
                (natp i)
                (< 0 o)
                (< 0 i) )
            (not (< (len (after i w)) (+ (* -1 i) (* i o)) ) ) )

  :hints (("Goal"

```

```

      :use ((:instance neg-* (a (* i o)) (b i) )
            (:instance layer-size-inequality (i i) (o o) (w w) ) ) ) )
;-----

; Theorem 1: calling makeLayer with valid inputs returns a valid
; layer recognized by isLayer.

(defthmd makeLayer-isLayer
  (implies (and (<= (* i o) (len w))
                (isListOfWeights w)
                (natp o)
                (natp i)
                (< 0 o)
                (< 0 i) )
            (isLayer (makeLayer i o w)) )

  :hints (("Goal"
           :use ((:instance makeLayer-isLayer2 (i i) (o (- o 1)) (w (after i w)) ))
           :in-theory (enable neuron-subset
                             layer-size-inequality-2
                             neg-*
                             front-len
                             greater-*
                             after-subset
                             inputs-within-weights
                             trans-<=) ) ) )

;-----

****
networks.lisp
****
; Networks Book
(in-package "ACL2")

(include-book "layers")

;-----

; Network - list of layers from input layer to output layer, where
; the number of neurons in any given layer matches the number of
; synapses in each neuron of the following layer. Network must have
; at least one layer.

(defun isNetwork2 (x i)
  (if (consp x)
      (if (isLayer (car x))
          (if (equal (len (caar x)) i)
              (isNetwork2 (cdr x) (len (car x)))
              nil)
          nil)
      (not x)))

(defun isNetwork (x)
  (if (consp x)
      (if (isLayer (car x))
          (isNetwork2 (cdr x) (len (car x)))
          nil)
      nil))

; Signals are passed through a network by passing the signals through
; each layer in succession, with the output of one layer being the
; input to the next. The forwardNetwork function works for a list of
; inputs i with length equal to the number of synapses in each neuron
; of the first layer of network n.

(defun forwardNetwork (f i n)
  (if (consp n)
      (forwardNetwork f (forwardLayer f i (car n)) (cdr n))

```

```

    i))

; A network can be made from a flat list of weights if the number of
; inputs and outputs to each layer is specified. Since the number of inputs
; to every layer after the input layer equals the number of outputs from the
; previous layer, this information can be encoded by a single list whose first
; entry is the number of inputs to the input layer, and the following entries
; are the number of outputs for each layer.

(defun makeNetwork (i w)
  (if (consp i)
      (if (consp (cdr i))
          (cons (makeLayer (car i) (cadr i) (front (* (car i) (cadr i)) w))
                (makeNetwork (cdr i) (after (* (car i) (cadr i)) w)))
          nil)
      nil))

;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defun countSynapses (x)
  (if (consp x)
      (if (consp (cdr x))
          (+ (* (car x) (cadr x))
             (countSynapses (cdr x)))
          0)
      0))

(defun posnats (x)
  (if (consp x)
      (if (not (zp (car x)))
          (posnats (cdr x))
          nil)
      (not x)))

(defthmd cdr-countSynapses
  (implies (and (consp i)
                (consp (cdr i)) )
           (equal (countSynapses (cdr i)) (- (countSynapses i) (* (car i) (cadr i))))))

(defthmd first-two-countSynapses
  (implies (posnats i)
           (<= (* (car i) (cadr i)) (countSynapses i))))

(defthmd nats-in-posnats
  (implies (and (consp i)
                (consp (cdr i))
                (posnats i) )
           (and (< 0 (car i))
                (< 0 (cadr i))
                (natp (car i))
                (natp (cadr i)) ) ) )

(defthmd remove-layer
  (implies (and (<= (countSynapses i) (len w))
                (posnats i)
                (consp i)
                (consp (cdr i)) )
           (<= (countSynapses (cdr i)) (len (after (* (car i) (cadr i)) w))))

:hints ("Goal"
        :do-not-induct t
        :in-theory (enable nats-in-posnats)
        :use ( (:instance first-two-countSynapses (i i) )
                (:instance after-len (i (* (car i) (cadr i))) (w w) )
                (:instance trans-<= (a (* (car i) (cadr i))
                                       (b (countSynapses i))
                                       (c (len w)) )
                (:instance cdr-countSynapses (i i) ) ) ) )

```

```

(defthmd more-weights-than-two-layers
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 1 (len i))
                (<= (countSynapses i) (len w)) )
    (not (< (len w) (* (car i) (cadr i)))) )

  :hints (("Goal"
    :do-not-induct t
    :use ( (:instance trans-<= (a (* (car i) (cadr i)))
                                (b (countSynapses i))
                                (c (len w)) )
          (:instance first-two-countSynapses (i i) )
          :in-theory (enable nats-in-posnats) ) ) )

(defthmd isLayer-premise-1
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 1 (len i))
                (<= (countSynapses i) (len w)) )
    (<= (* (car i) (cadr i))
      (len (front (* (car i) (cadr i)) w))) )

  :hints (("Goal"
    :do-not-induct t
    :use ( (:instance front-len (i (* (car i) (cadr i))) (w w) )
          :in-theory (enable more-weights-than-two-layers) ) ) )

(defthmd isLayer-premise-2
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 1 (len i))
                (<= (countSynapses i) (len w)) )
    (isListOfWeights (front (* (car i) (cadr i)) w))) )

  :hints (("Goal"
    :use ( (:instance neuron-subset (i (* (car i) (cadr i))) (w w) )
          (:instance first-two-countSynapses (i i) )
          (:instance trans-<= (a (* (car i) (cadr i)))
                              (b (countSynapses i))
                              (c (len w)) )
          :do-not-induct t) ) )

(defthmd isLayer-premises-3+
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 2 (len i))
                (<= (countSynapses i) (len w)) )
    (and (natp (car i))
          (natp (cadr i))
          (< 0 (car i))
          (< 0 (cadr i)) ) )

  :hints (("Goal"
    :in-theory (enable nats-in-posnats)
    :do-not-induct t) ) )

(defthmd makeLayer-in-makeNetwork-isLayer
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 2 (len i))
                (<= (countSynapses i) (len w)) )
    (isLayer (makeLayer (car i) (cadr i) (front (* (car i) (cadr i)) w))) )

  :hints (("Goal"
    :do-not-induct t
    :use ( (:instance makeLayer-isLayer (i (car i)) (o (cadr i))
                                        (w (front (* (car i) (cadr i)) w))) )
    :in-theory (enable isLayer-premise-1
                      isLayer-premise-2) ) )

```

```

isLayer-premises-3+)) ) )

(defthmd not-zp-makeLayer
  (implies (not (zp o))
    (equal (makeLayer i o w)
      (cons (front i w) (makeLayer i (- o 1) (after i w))) ) ) )

(defthmd inputs-per-layer
  (implies (and (natp o)
    (< 0 o)
    (<= (* i o) (len w))
    (< 0 i)
    (natp i))
    (equal (len (car (makeLayer i o w))) i) )

  :hints (("Goal"
    :use ((:instance not-zp-makeLayer (i i) (o o) (w w) )
      (:instance front-len (i i) (w w) )
      (:instance greater-* (i i) (o o) )
      (:instance trans-<= (a i) (b (* i o)) (c (len w)) ) )
    :do-not-induct t)) )

(defthmd isLayer-after
  (implies (and (isListOfWeights w)
    (posnats i)
    (<= 2 (len i))
    (<= (countSynapses i) (len w)) )
    (isListOfWeights (after (* (car i) (cadr i)) w)))

  :hints (("Goal"
    :use ((:instance after-subset (i (* (car i) (cadr i))) (w w) )
      (:instance first-two-countSynapses (i i) )
      (:instance trans-<= (a (* (car i) (cadr i))) (b (countSynapses i))
        (c (len w)) ) )
    :do-not-induct t)) )

(defthmd def-makeNetwork
  (implies (and (isListOfWeights w)
    (posnats i)
    (<= 2 (len i))
    (<= (countSynapses i) (len w)) )
    (equal (makeNetwork i w)
      (cons (makeLayer (car i) (cadr i) (front (* (car i) (cadr i)) w))
        (makeNetwork (cdr i) (after (* (car i) (cadr i)) w))) ) )

  :hints (("Goal"
    :do-not-induct t)) )

(defthmd minus-1
  (implies (and (natp i)
    (natp o)
    (equal i o) )
    (equal (- i 1) (- o 1) ) )

  :hints (("Goal"
    :do-not-induct t)) )

(defthmd len-0
  (implies (equal (len i) 0)
    (not (consp i)) ) )

(defthmd len-only-1
  (implies (equal 1 (len i))
    (not (consp (cdr i))) )

  :hints (("Goal"
    :use ((:instance len-0 (i (cdr i)) )
      (:instance minus-1 (i 1) (o (len i)) ) )
    :do-not-induct t)) )

(defthmd def-makeNetwork-2

```



```

      (<= 2 (len (cdr i)))
      (<= (countSynapses (cdr i))
         (len (after (* (car i) (cadr i)) w))) )
(isNetwork2 (makeNetwork (cdr i)
                        (after (* (car i) (cadr i)) w)
                        (car (cdr i)))) )
(isNetwork2 (makeNetwork (cdr i) (after (* (car i) (cadr i)) w)
                        (car (cdr i)))) )

:hints ("Goal"
       :use (:instance remove-layer (i i) (w w) )
           (:instance isLayer-after (i i) (w w) ))
       :do-not-induct t)) )

(defthmd front-inequality
  (implies (and (<= i (len w))
                (natp i) )
           (not (< (len (front i w)) i) ) )

  :hints ("Goal"
         :in-theory (enable front-len) )) )

(defthmd front-layer-size-inequality
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 2 (len i))
                (<= (countSynapses i) (len w)) )
           (not (< (len (front (* (car i) (cadr i)) w)) (* (car i) (cadr i)) )) )

  :hints ("Goal"
         :use (:instance front-inequality (i (* (car i) (cadr i))) (w w) )
             (:instance ineq-*-countSynapses-len (i i) (w w) ))
         :do-not-induct t)) )

(defthmd makeNetwork-isNetwork2
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 2 (len i))
                (<= (countSynapses i) (len w)) )
           (isNetwork2 (makeNetwork i w) (car i)) )

  :hints ("Goal"
         :induct (makeNetwork i w) )
         ("Subgoal *1/1"
          :use (:instance inputs-per-layer (i (car i)) (o (cadr i))
              (w (front (* (car i) (cadr i)) w)))
              (:instance len-2-makeNetwork-isNetwork2 (i i) (w w) )
              (:instance front-layer-size-inequality (i i) (w w) )
              (:instance inductive-premises-makeNetwork-isNetwork2 (i i) (w w) )
              (:instance ineq-*-countSynapses-len (i i) (w w) )
              (:instance front-len (i (* (car i) (cadr i))) (w w) )
              (:instance nats-in-posnats (i i) )
              (:instance makeLayer-in-makeNetwork-isLayer (i i) (w w) ))
          :do-not-induct t)
         ("Subgoal *1/1.2"
          :use (:instance isLayer-premise-2 (i i) (w w) )
              (:instance number-outputs-in-layer (i (car i)) (o (cadr i))
              (w (front (* (car i) (cadr i)) w))))
          :do-not-induct t)
         ("Subgoal *1/1.1"
          :use (:instance isLayer-premise-2 (i i) (w w) )
              (:instance number-outputs-in-layer (i (car i)) (o (cadr i))
              (w (front (* (car i) (cadr i)) w))))
          :do-not-induct t)) )

(defthmd natp-countSynapses
  (implies (posnats i)
           (natp (countSynapses i)) )

  :hints ("Goal"
         :induct (countSynapses i) )) )

```

```

(defthmd mini-ineq
  (implies (and (posnats i4)
                (< 0 i3)
                (integerp i3)
                (integerp i1)
                (< 0 i1) )
            (<= (* i1 i3)
                 (+ (* i1 i3)
                     (countSynapses i4)
                     (* i3 (car i4)))) )

  :hints (("Goal"
           :use ((:instance nats-in-posnats (i i4) )
                 (:instance natp-countSynapses (i i4) ))
           :do-not-induct t)) )

;-----

; Theorem 2: calling makeNetwork with valid inputs returns a valid
; network recognized by isNetwork.

(defthmd makeNetwork-isNetwork
  (implies (and (isListOfWeights w)
                (posnats i)
                (<= 3 (len i))
                (<= (countSynapses i) (len w)) )
            (isNetwork (makeNetwork i w)) )

  :hints (("Goal"
           :use ((:instance isLayer-after (i i) (w w) )
                 (:instance def-makeNetwork (i i) (w w) )
                 (:instance makeLayer-in-makeNetwork-isLayer (i i) (w w) )
                 (:instance remove-layer (i i) (w w) )
                 (:instance makeNetwork-isNetwork2 (i (cdr i)) (w (after (* (car i)
(cadr i)) w)) ))
           :do-not-induct t)
          ("Subgoal 2"
           :use ((:instance isLayer-premise-2 (i i) (w w) )
                 (:instance nats-in-posnats (i i) )
                 (:instance number-outputs-in-layer (i (car i)) (o (cadr i))
                                                       (w (front (* (car i) (cadr i)) w))))
           :do-not-induct t)
          ("Subgoal 2.1"
           :use ((:instance front-inequality (i (* i1 i3)) (w w) )
                 (:instance mini-ineq (i1 i1) (i3 i3) (i4 i4) )
                 (:instance trans-<= (a (* i1 i3)
                                         (b (+ (* i1 i3)
                                                (countSynapses i4)
                                                (* i3 (car i4))))
                                         (c (len w)) ))
           :do-not-induct t)) )

;-----
****
swap.lisp
****
; Swap Book
(in-package "ACL2")

(include-book "layers")

;-----

; A function for swapping two elements of a list at given indices.
; Works properly when both indices are within the list.
; Arguments are a list x, then the two indices i and j.

(defun swap2 (x i j)
  (if (equal i j)

```

```

x
(let* ( (a (front i x))
        (bc (after i x))
        (b (front (- j i) bc))
        (c (after (- j i) bc)) )
      (append a
              (cons (car c)
                    (append (cdr b)
                            (cons (car b)
                                  (cdr c)) ) ) ) ) )

(defun swap (x i j) (swap2 x (min i j) (max i j)))

;-----
; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defthmd swap2-same
  (implies (and (natp i)
                (< i (len x)) )
            (equal (swap2 x i i) x) )

  :hints (("Goal"
           :do-not-induct t)) )

(defthmd front-app
  (implies (<= i (len x))
            (equal (front i (append (front i x) b))
                   (front i x)) ) )

(defthmd not-zp++
  (implies (and (integerp a)
                (<= 0 a)
                (not (zp b)) )
            (not (zp (+ a b))) ) )

(defthmd after-step
  (implies (not (zp i))
            (equal (after i x) (after (- i 1) (cdr x))) ) )

(defthmd +-assoc
  (equal (+ a b c) (+ b a c)) )

(defthmd after-nil
  (equal (after i nil) nil) )

(defthmd after-after
  (implies (and (true-listp x)
                (natp a)
                (natp b) )
            (equal (after a (after b x)) (after (+ a b) x)) )

  :hints (("Goal"
           :in-theory (enable after-nil) )
           ("Subgoal *1/5'4'"
            :use ( (:instance not-zp++ (a a) (b b) )
                  (:instance +-assoc (a -1) (b a) (c b) )
                  (:instance after-step (i (+ a b)) (x (cons x1 x2)) ) ) ) ) )

(defthmd cdr-after
  (implies (< 0 (len x))
            (equal (cdr x) (after 1 x)) ) )

(defthmd after-append-front
  (implies (<= i (len x))
            (equal (after i (append (front i x) y)) y) ) )

(defthmd +-inverse-3
  (implies (natp j)
            (equal (+ i (- i) j) j)) )

```

```

(defthmd cdr-after2
  (implies (and (true-listp x)
                (natp i) )
           (equal (cdr (after i x))
                  (after (+ i 1) x) ) )

  :hints (("Goal"
           :use ( (:instance not-zp+ (a 1) (b i) ) )
           :in-theory (enable after-nil) ) ) )

(defthmd after-cons
  (implies (not (zp i))
           (equal (after i (cons e x))
                  (after (- i 1) x) ) ) )

(defthmd car-front
  (implies (not (zp i))
           (equal (car (front i x))
                  (car x) ) ) )

(defthmd cdr-front
  (implies (not (zp i))
           (equal (cdr (front i x))
                  (front (- i 1) (cdr x)) ) ) )

(defthmd len-after-helper
  (implies (and (natp i)
                (natp j)
                (< i (len x))
                (< j (len x)) )
           (<= (+ -1 (- i) j)
                (len (after (+ 1 i) x)) ) )

  :hints (("Goal"
           :use ( (:instance after-len (i (+ 1 i)) (w x) ) ) ) )

(defthmd not-zp
  (implies (not (zp j))
           (and (natp j)
                (<= 1 j) ) ) )

(defthmd trans-<=<-<
  (implies (and (<= a b) (< b c) (< a c) ) )

(defthmd len-<-1
  (implies (< 1 (len x))
           (consp x) ) )

(defthmd combine-afters
  (implies (and (true-listp x)
                (not (zp j))
                (< j (len x)) )
           (equal (after j x)
                  (cons (car (after j x))
                        (after (+ 1 j) x) ) ) )

  :hints (("Goal"
           :do-not-induct t
           :use (not-zp
                 len-<-1
                 (:instance after-len (i j) (w x) )
                 (:instance trans-<=<-< (a 1) (b j) (c (len x)) )
                 (:instance cdr-after2 (i j) (x x) ) ) ) ) )

(defthmd append-front-after
  (implies (and (natp i)
                (true-listp x)
                (< i (len x)) )
           (equal (append (front i x) (after i x)) x) ) )

```

```

(defthmd assoc+-
  (equal (+ -1 a (- (+ -1 b)))
    (+ a (- b)) ) )

(defthmd app-front-after-after
  (implies (and (true-listp x)
    (< b a)
    (< a (len x))
    (< b (len x))
    (natp a)
    (natp b) )
    (equal (after b x)
      (append (front (- a b) (after b x))
        (after a x)) ) )

  :hints (("Goal"
    :use (assoc+-)
    :in-theory (enable append-front-after) )
    ("Subgoal *1/2.1"
    :use (assoc+-) ) ) )

(defthmd zero+-
  (equal (+ -1 (- i) 1 i) 0) )

(defthmd duh+-
  (equal (+ j (- (+ 1 i))) (+ -1 (- i) j)) )

(defthmd swap2-double
  (implies (and (true-listp x)
    (<= i j)
    (natp i)
    (natp j)
    (< i (len x))
    (< j (len x)) )
    (equal (swap2 (swap2 x i j) i j) x) )

  :hints (("Goal"
    :do-not-induct t
    :in-theory (enable +-inverse-3
      front-app
      after-after
      after-cons
      cdr-after
      cdr-after2
      after-append-front
      car-front
      cdr-front
      swap2-same) )
    ("Subgoal 1'"
    :use (len-after-helper
      (:instance after-append-front (i (+ -1 (- i) j))
        (x (after (+ 1 i) x))
        (y (cons (car (after i x))
          (after (+ 1 j) x))) ) ) )
    ("Subgoal 1'4'"
    :use (combine-afters) )
    ("Subgoal 1'6'"
    :use (zero+-
      (:instance app-front-after-after (a j) (b (+ 1 i)) (x x) ) ) )
    ("Subgoal 1.2'"
    :use ((:instance combine-afters (j i) (x x) ) ) )
    ("Subgoal 1.2.2"
    :use (cdr-after) )
    ("Subgoal 1.2.1"
    :use (append-front-after) )
    ("Subgoal 1.1"
    :use (zero+-
      (:instance app-front-after-after (a j) (b (+ 1 i)) (x x) ) ) )
    ("Subgoal 1.1'"
    :use (duh+-) )
    ("Subgoal 1.1'4'"

```

```

      :use (:instance combine-afters (j i) (x x) ) )
      ("Subgoal 1.1.1'5'"
       :use (append-front-after) ) )

;-----

; Interesting Facts about swap

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

(defthmd swap-commutative
  (implies (and (natp i)
                (natp j)
                (< i (len x))
                (< j (len x)) )
           (equal (swap x i j) (swap x j i)) )

  :hints (("Goal"
           :do-not-induct t)) )

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

(defthmd swap-same
  (implies (and (natp i)
                (< i (len x)) )
           (equal (swap x i i) x) )

  :hints (("Goal"
           :in-theory (enable swap2-same)
           :do-not-induct t)) )

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

(defthmd swap-double
  (implies (and (true-listp x)
                (natp i)
                (natp j)
                (<= 0 i)
                (<= 0 j)
                (< i (len x))
                (< j (len x)) )
           (equal (swap (swap x i j) i j) x) )

  :hints (("Goal"
           :use (swap2-double
                 (:instance swap2-double (j i) (i j) (x x) ))
           :do-not-induct t)) )

;-----
****
form1.lisp
****
; Form Book 1 - Rearrange Neural Networks
(in-package "ACL2")

(include-book "networks")
(include-book "swap")

;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defthmd isListOfWeights-true-listp
  (implies (isListOfWeights i)
           (true-listp i) ) )

```

```

(defthmd isNeuron-true-listp
  (implies (isNeuron n)
    (true-listp n) )

  :hints (("Goal"
    :in-theory (enable isListOfWeights-true-listp) )) )

(defthmd k-zero1
  (equal (+ k j (- k)) (+ 0 j)) )

(defthmd k-zero2
  (equal (+ 1 k j (- k)) (+ 1 j)) )

(defthmd cons-front
  (implies (not (zp k))
    (equal (front k i)
      (cons (car i) (front (- k 1) (cdr i))) ) ) )

(defthmd obvious+-
  (equal (+ -1 -1 j (- (+ -1 k)))
    (+ -1 j (- k)) ) )

(defthmd commute+-
  (equal (+ 1 -1 k) (+ -1 1 k)) )

(defthmd forwardNeuron-over-app
  (implies (and (isNeuron b)
    (isNeuron d)
    (isListOfWeights a)
    (isListOfWeights c)
    (equal (len a) (len c))
    (equal (len b) (len d)) )
    (equal (forwardNeuron (append a b)
      (append c d))
      (+ (forwardNeuron a c)
        (forwardNeuron b d)) ) ) )

(defthmd forwardNeuron-over-app2
  (implies (and (isNeuron b)
    (isNeuron d)
    (isListOfWeights a)
    (isListOfWeights c)
    (equal (len a) (len c))
    (equal (len b) (len d)) )
    (equal (forwardNeuron (append a b)
      (append c d))
      (+ (forwardNeuron b d)
        (forwardNeuron a c)) ) ) )

(defun app3-help (a c)
  (if (consp a)
    (app3-help (cdr a) (cdr c))
    c))

(defthmd forwardNeuron-over-app3-helper
  (implies (and (isNeuron b)
    (isNeuron d)
    (isListOfWeights a)
    (isListOfWeights c)
    (equal (len a) (len c))
    (equal (len b) (len d)) )
    (equal (+ (* (car b) (car d))
      (forwardNeuron (append a (cdr b))
        (append c (cdr d))))
      (forwardNeuron (append a b)
        (append c d)) ) )

  :hints (("Goal"
    :induct (app3-help a c) )) )

(defthmd forwardNeuron-over-app3

```

```

(implies (and (isNeuron b)
              (isNeuron d)
              (consp (cdr b))
              (consp (cdr d))
              (isListOfWeights a)
              (isListOfWeights c)
              (equal (len a) (len c))
              (equal (len b) (len d)))
         (equal (+ (* (car b) (car d))
                  (forwardNeuron (append a (cdr b))
                                    (append c (cdr d))))
              (+ (* (cadr b) (cadr d))
                  (forwardNeuron a c)
                  (* (car b) (car d))
                  (forwardNeuron (caddr b) (caddr d))))))

:hints (("Goal"
        :do-not-induct t
        :use (forwardNeuron-over-app3-helper
              forwardNeuron-over-app
              forwardNeuron-over-app2) )) )

(defthmd front-after-components
  (implies (and (true-listp n)
                (natp j)
                (< j (len n)))
           (equal (append (front j n) (after j n)) n) ) )

(defthmd j-len-helper
  (implies (and (equal (len n) (len i))
                (integerp j)
                (< j (len i)))
           (<= (+ 1 j) (len n)) ) )

(defthmd after-len-i-n-helper
  (implies (and (equal (len n) (len i))
                (true-listp i)
                (natp j)
                (< j (len i)))
           (equal (len (after (+ 1 j) i))
                  (len (after (+ 1 j) n))) ) )

:hints (("Goal"
        :do-not-induct t
        :use (j-len-helper
              (:instance after-len (i (+ 1 j)) (w i) )
              (:instance after-len (i (+ 1 j)) (w n) )))) )

(defthmd another-len-helper
  (implies (and (equal (len n) (len i))
                (true-listp i)
                (natp j)
                (< j (len i)))
           (equal (len (cons (car i) (after (+ 1 j) n)))
                  (len (cons (car i) (after (+ 1 j) n)))) ) )

:hints (("Goal"
        :use (after-len-i-n-helper) )) )

(defthmd unequal-j-helper
  (implies (and (natp j)
                (< 0 (len i))
                (< j (len i)))
           (not (equal (len (after 1 i)) (+ -1 j))) ) )

:hints (("Goal"
        :in-theory (enable cdr-after) )) )

(defthmd not-<=-j-helper
  (implies (and (natp j)
                (< 0 (len i)))

```

```

      (< j (len i)) )
      (not (<= (+ 1 (len (after 1 i))) j)) )

:hints (("Goal"
        :in-theory (enable cdr-after) )) )

(defthmd <=-step
  (equal (<= (+ 1 k) j)
         (<= k (+ -1 j)) ) )

(defthmd not-<-j-helper
  (implies (and (natp j)
                (< 0 (len i))
                (< j (len i)) )
           (not (< (len (after 1 i)) (+ -1 j))) )

:hints (("Goal"
        :in-theory (enable cdr-after) )) )

(defthmd not-<-n-helper
  (implies (and (natp j)
                (equal (len i) (len n))
                (< 0 (len i))
                (< j (len i)) )
           (not (< (len (after 1 n)) (+ -1 j))) )

:hints (("Goal"
        :in-theory (enable cdr-after) )) )

(defthmd not-<-j-helper2
  (implies (and (natp j)
                (< 0 (len i))
                (< j (len i)) )
           (not (< (+ 1 (len (after 1 i))) (+ 1 j))) )

:hints (("Goal"
        :in-theory (enable cdr-after) )) )

(defthmd overall-j-len-helper
  (implies (and (natp j)
                (equal (len i) (len n))
                (< 0 (len i))
                (< j (len i)) )
           (and (not (equal (len (after 1 i)) (+ -1 j)))
                (not (<= (+ 1 (len (after 1 i))) j))
                (not (< (len (after 1 i)) (+ -1 j)))
                (not (< (len (after 1 n)) (+ -1 j)))
                (not (< (+ 1 (len (after 1 i))) (+ 1 j))) ) )

:hints (("Goal"
        :do-not-induct t
        :use (unequal-j-helper
              not-<=-j-helper
              not-<-j-helper
              not-<-j-helper2
              not-<-n-helper) )) )

(defthmd front-subset
  (implies (and (isListOfWeights w)
                (natp i)
                (<= i (len w)) )
           (isListOfWeights (front i w)) )

:hints (("Goal"
        :in-theory (enable trans-<=
                      len-1)
        :induct (front i w) )) )

(defthmd isListOfWeights-helper
  (implies (and (isListOfWeights i)
                (natp j)

```

```

      (< j (len i)) )
      (isListOfWeights (after 1 (cons (car i) (after (+ 1 j) i)))) )

:hints (("Goal"
        :do-not-induct t
        :in-theory (enable after-subset)
        :use ( (:instance cdr-after (x (cons (car i) (after (+ 1 j) i))) ) ) ) )

(defthmd isListOfWeights-helper2
  (implies (and (isListOfWeights (cdr n))
                (natp j)
                (< j (len i))
                (equal (len i) (len n)) )
            (isListOfWeights (front (+ -1 j) (after 1 n))) )

:hints (("Goal"
        :do-not-induct t
        :use ( (:instance front-subset (i (+ -1 j)) (w (after 1 n)) ) )
        :in-theory (enable cdr-after) ) )

(defthmd after-true-listp
  (implies (true-listp i)
            (true-listp (after j i)) ) )

(defthmd after-0
  (equal (after 0 x) x) )

(defun after-helper (i j x)
  (if (or (zp i) (zp j))
      x
      (after-helper (- i 1) (- j 1) (cdr x)) ) )

(defthmd len-not-equal
  (implies (not (equal (len x) (len y)))
            (not (equal x y)) ) )

(defthmd consp-duh
  (implies (consp x)
            (not (equal x (cdr x))) ) )

(defthmd zp-step-after
  (implies (and (consp x)
                (< j (len x))
                (equal x (after j x)))
            (zp j) )

:hints (("Subgoal *1/"
        :do-not-induct t
        :use (consp-duh
              (:instance after-len (i j) (w x) )
              (:instance len-not-equal (x x) (y (after j x)) ) ) ) )

(defthmd zp-after-or
  (implies (and (zp i)
                (natp i)
                (consp x)
                (< j (len x))
                (equal (after i x) (after j x)) )
            (zp j) )

:hints (("Goal"
        :use (after-0
              zp-step-after) ) )

(defthmd <-duh-i
  (not (< i i)) )

(defthmd after-arg-equal
  (implies (and (equal (after i x) (after j x))
                (< j (len x))
                (< i (len x))

```

```

      (natp i)
      (natp j) )
    (equal (nfix i) j) )

:hints ("Goal"
  :in-theory (enable after-0
    <-duh-i
    zp-after-or)
  :induct (after-helper i j x) )
("Subgoal *1/1"
  :use (zp-after-or
    (:instance zp-after-or (i j) (j i) (x x) )) ))

(defthmd after-j-1
  (implies (and (consp n)
    (natp j)
    (< j (len n))
    (< 0 (len n))
    (equal (after (+ -1 j) (cdr n)) (cdr n)) )
    (<= j 1) )

  :hints ("Goal"
    :use ((:instance after-arg-equal (i 0) (j (+ -1 j)) (x (cdr n)))
      (:instance after-0 (x (cdr n))))
    :do-not-induct t)) )

(defthmd not-consp-after-helper
  (implies (and (not (consp (after k x)))
    (<= k (len x))
    (natp k) )
    (equal (len x) k) ) )

(defthmd rationalp-after
  (implies (and (isListOfWeights w)
    (natp k)
    (< k (len w)) )
    (rationalp (car (after k w))) ) )

(defthmd isListOfWeights-after-helper
  (implies (and (isListOfWeights w)
    (< k (len w)) )
    (isListOfWeights (cdr (after k w))) ) )

(defthmd consp-after-helper
  (implies (and (< k (len w))
    (natp k) )
    (consp (after k w)) ) )

(defthmd equal-len-after
  (implies (and (equal (len i) (len n))
    (natp j)
    (< j (len i)) )
    (equal (len (after j i))
      (len (after j n))) ) )

(defthmd decompose-forwardNeuron
  (implies (and (isNeuron n)
    (isListOfWeights i)
    (natp j)
    (equal (len n) (len i))
    (< 0 j)
    (< j (len i)) )
    (equal (forwardNeuron (after 1 i) (after 1 n))
      (+ (forwardNeuron (front (+ -1 j) (after 1 i))
        (front (+ -1 j) (after 1 n)))
        (forwardNeuron (after j i)
          (after j n)) ) ) )

:hints ("Goal"
  :do-not-induct t
  :use (isListOfWeights-true-listp

```

```

      (:instance isListOfWeights-true-listp (i n) ))
:in-theory (enable after-true-listp
                front-after-components
                forwardNeuron-over-app
                after-step
                cdr-after) )

("Goal''")
:in-theory (disable front-after-components
                  forwardNeuron-over-app)
:use (overall-j-len-helper
      after-j-1
      (:instance not-consp-after-helper (x (cdr i)) (k (+ -1 j)) )
      (:instance after-step (i j) (x i) )
      (:instance <==step (j j) (k (len (after 1 i))) )
      (:instance after-true-listp (i i) (j 1) )
      (:instance front-after-components (j (+ -1 j)) (n (after 1 i)) )
      (:instance front-after-components (j (+ -1 j)) (n (after 1 n)) )) )

("Subgoal 6")
:use ((:instance front-subset (i (+ -1 j)) (w (cdr i)) )
      (:instance front-subset (i (+ -1 j)) (w (cdr n)) )
      (:instance front-len (i (+ -1 j)) (w (cdr i)) )
      (:instance front-len (i (+ -1 j)) (w (cdr n)) )
      (:instance consp-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr i)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr i)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr n)) )) )

("Subgoal 6'")
:use ((:instance equal-len-after (i (cdr i)) (n (cdr n)) (j (+ -1 j)) )
      (:instance forwardNeuron-over-app2 (b (after (+ -1 j) (cdr i)))
                                           (d (after (+ -1 j) (cdr n)))
                                           (a (front (+ -1 j) (cdr i)))
                                           (c (front (+ -1 j) (cdr n)))) )
      (:instance front-after-components (j (+ -1 j)) (n (cdr n)) )
      (:instance front-after-components (j (+ -1 j)) (n (cdr i)) )) )

("Subgoal 4")
:use ((:instance equal-len-after (i (cdr i)) (n (cdr n)) (j (+ -1 j)) )
      (:instance front-subset (i (+ -1 j)) (w (cdr i)) )
      (:instance front-subset (i (+ -1 j)) (w (cdr n)) )
      (:instance front-len (i (+ -1 j)) (w (cdr i)) )
      (:instance front-len (i (+ -1 j)) (w (cdr n)) )
      (:instance consp-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr i)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr i)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr n)) )
      (:instance forwardNeuron-over-app2 (b (after (+ -1 j) (cdr i)))
                                           (d (after (+ -1 j) (cdr n)))
                                           (a (front (+ -1 j) (cdr i)))
                                           (c (front (+ -1 j) (cdr n)))) )) )

("Subgoal 2")
:use ((:instance equal-len-after (i (cdr i)) (n (cdr n)) (j (+ -1 j)) )
      (:instance front-subset (i (+ -1 j)) (w (cdr i)) )
      (:instance front-subset (i (+ -1 j)) (w (cdr n)) )
      (:instance front-len (i (+ -1 j)) (w (cdr i)) )
      (:instance front-len (i (+ -1 j)) (w (cdr n)) )
      (:instance consp-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr i)) )
      (:instance isListOfWeights-after-helper (k (+ -1 j)) (w (cdr n)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr i)) )
      (:instance rationalp-after (k (+ -1 j)) (w (cdr n)) )
      (:instance forwardNeuron-over-app2 (b (after (+ -1 j) (cdr i)))
                                           (d (after (+ -1 j) (cdr n)))
                                           (a (front (+ -1 j) (cdr i)))
                                           (c (front (+ -1 j) (cdr n)))) )) )

(defun faa-helper (j k i n)
  (if (zp k)
      (cons j (cons i n))
      (faa-helper (- j 1) (- k 1) (cdr i) (cdr n)) ))

```



```

                                                                    (after (+ 1 j) n))) ) )
("Subgoal *1/1.1.3"
 :expand ((len i) )
("Subgoal *1/1.1.2"
 :use ( (:instance cdr-after (x i) ) ) )
("Subgoal *1/1.1.1"
 :in-theory (enable after-step)
 :use (isNeuron-true-listp
       (:instance after-subset (i (+ -1 j)) (w (cdr i)) )
       (:instance cdr-after2 (i j) (x n) )
       (:instance cdr-after2 (i j) (x i) )
       (:instance front-after-components (j (+ -1 j)) (n (after 1 i)) )
       (:instance front-after-components (j (+ -1 j)) (n (cdr n)) )
       (:instance forwardNeuron-over-app3 (a (front (+ -1 j) (after 1 i)))
                                           (b (cons (car i)
                                                  (after (+ -1 j)
                                                       (after 1 i))))
                                           (c (front (+ -1 j) (cdr n)))
                                           (d (cons (car n)
                                                  (after (+ -1 j)
                                                       (cdr n)))))) ) ) )
;-----
; 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.

(defthmd forwardNeuron-swap-swap
  (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) ) ) )

  :hints ( ("Goal"
            :cases ((< j k) (< k j))
            :expand ((swap i j k)
                    (swap n j k) )
            :use (isListOfWeights-true-listp
                  isNeuron-true-listp
                  k-zero1
                  k-zero2)
            :in-theory (enable front-app
                                after-after
                                after-cons
                                cdr-after
                                cdr-after2
                                after-append-front
                                car-front
                                cdr-front
                                not-zp+
                                after-nil
                                +-inverse-3
                                after-cons
                                not-zp
                                append-front-after)

                      :do-not-induct t)
            ("Subgoal 2"
            :hands-off (isNeuron)
            :use ( (:instance cdr-after2 (i k) (x n) )
                  (:instance forwardNeuron-app-app (j k) (k j) (n n) (i i) ) )
            :expand ((swap i j k)
                    (swap n j k)
                    (swap2 i j k)
                    (swap2 n j k) ) ) )

```

```

      ("Subgoal 1"
       :hands-off (isNeuron)
       :use ((:instance cdr-after2 (i j) (x n) )
            forwardNeuron-app-app)
       :expand ((swap i j k)
                (swap n j k)
                (swap2 i k j)
                (swap2 n k j) ) ) )

; 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.

(defthmd synapse-swap-in-neuron
  (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)) ) )

  :hints (("Goal"
           :use (forwardNeuron-swap-swap
                 isListOfWeights-true-listp
                 isNeuron-true-listp)
           :in-theory (enable front-app
                              after-after
                              after-cons
                              cdr-after
                              cdr-after2
                              after-append-front
                              car-front
                              cdr-front
                              not-zp+
                              after-nil
                              +-inverse-3
                              after-cons
                              not-zp
                              append-front-after)
           :do-not-induct t)) )

;-----
****
form2.lisp
****
; Form Book 2 - More Theorems for Rearranging Neural Networks
(in-package "ACL2")

(include-book "form1")

;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defthmd forwardLayer-over-app
  (implies (and (isLayer a)
                (isLayer b)
                (isListOfWeights i)
                (equal (len (car a)) (len i))
                (equal (len (car b)) (len i)) )
            (equal (forwardLayer f i (append a b))
                  (append (forwardLayer f i a)
                          (forwardLayer f i b)) ) )

  :hints (("Goal"

```

```

      :induct (append a b) )) )

(defthmd front-consp-zp
  (implies (and (not (consp (front k l)))
                (consp l) )
            (zp k) ) )

(defthmd isLayer2-front
  (implies (and (isLayer2 l j)
                (natp i)
                (<= i (len l)) )
            (isLayer2 (front i l) j) ) )

(defthmd isLayer-front
  (implies (and (isLayer l)
                (natp i)
                (< 0 i)
                (<= i (len l)) )
            (isLayer (front i l)) )

  :hints (("Goal"
           :in-theory (enable isLayer2-front) )) )

(defthmd isLayer-true-listp
  (implies (isLayer l)
            (true-listp l) ) )

(defthmd ineq-duh
  (implies (and (natp j)
                (natp k)
                (< k j) )
            (not (< (+ j (- k)) 0)) ) )

(defthmd isLayer-after2
  (implies (and (isLayer l)
                (natp j)
                (< j (len l)) )
            (isLayer (after j l)) ) )

(defthmd not-zp-helper
  (implies (and (natp k)
                (natp j)
                (< k j) )
            (and (not (zp (+ j (- k))))
                 (< 0 (+ j (- k))))
                 (natp (+ j (- k)))) ) ) )

(defthmd isLayer2-append
  (implies (and (isLayer2 x i)
                (isLayer2 y i) )
            (isLayer2 (append x y) i) ) )

(defthmd isLayer-after-equal-neuron-len
  (implies (and (isLayer l)
                (natp k)
                (< k (len l)) )
            (equal (len (car l))
                   (len (car (after k l)))) ) ) )

(defthmd isLayer-after-equal-neuron-len2
  (implies (and (isLayer l)
                (natp k)
                (natp j)
                (< k (len l))
                (< j (len l)) )
            (equal (len (car (after j l)))
                   (len (car (after k l)))) ) )

  :hints (("Goal"
           :use (isLayer-after-equal-neuron-len
                 (:instance isLayer-after-equal-neuron-len (k j) (l l) ))
           )) )

```

```

      :do-not-induct t)) )

(defthmd consp-car-isLayer
  (implies (isLayer l)
    (consp (car l)) ) )

(defthmd lens-from-layers
  (implies (and (isLayer l)
    (natp j)
    (natp k)
    (< k (len l))
    (< j (len l))
    (< k j) )
    (equal (len (car (front (+ j (- k)) (after k l))))
      (+ 1 (len (cdar (after j l)))) ) )

  :hints (("Goal"
    :hands-off (isLayer)
    :use (isLayer-after2
      isLayer-after-equal-neuron-len2
      (:instance consp-car-isLayer (l (after j l)) )
      (:instance isLayer-after2 (j k) (l l) ) )
    :in-theory (enable after-len
      car-front)
    :do-not-induct t)) )

(defthmd isLayer-isLayer2-duh
  (iff (isLayer l)
    (and (isLayer2 (cdr l) (len (car l)))
      (isNeuron (car l)) ) ) )

(defthmd forwardLayer-len
  (implies (and (isLayer l)
    (isListOfWeights i)
    (equal (len (car l)) (len i)) )
    (equal (len l)
      (len (forwardLayer f i l)) ) ) )

(defthmd len-1-cdr-not-consp
  (implies (equal (len x) 1)
    (not (consp (cdr x))) )

  :hints (("Goal"
    :use (:instance len-0 (i (cdr x)) ) ) ) )

(defthmd isLayer-not-isLayer-cdr
  (implies (and (isLayer l)
    (not (isLayer (cdr l))) )
    (equal (len l) 1) ) )

(defthmd len-parts-+1
  (implies (and (equal (len a) (len b))
    (equal (+ 1 (len (cdr a))) 1) )
    (equal (+ 1 (len (cdr b))) 1) ) )

(defthmd zp-1+0
  (implies (equal (+ 1 a) 1)
    (and (<= a 0)
      (zp a)) ) )

(defthmd equal-len-equal-front-len
  (implies (equal (len a) (len b))
    (equal (len (front j a)) (len (front j b))) ) )

(defthmd len-parts-+1-front
  (implies (and (equal (len a) (len b))
    (< j (len a))
    (equal (+ 1 (len (cdr (front j a)))) 1) )
    (equal (+ 1 (len (cdr (front j b)))) 1) )

  :hints (("Goal"

```

```

      :use (equal-len-equal-front-len
            (:instance zp-1+0 (a (len (cdr (front j a)))) )
            (:instance len-parts+1 (a (front j a) (b (front j b)) ))
      :in-theory (enable front-len)
      :do-not-induct t)) )

(defthmd isLayer-consp
  (implies (isLayer l)
    (consp l) ) )

(defthmd add-neurons-to-layer
  (implies (and (isLayer l)
    (equal (len n) (len (car l)))
    (isNeuron n) )
    (isLayer (cons n l)) ) )

(defthmd add-neurons-to-layer2
  (implies (and (isLayer2 l i)
    (equal (len n) i)
    (isNeuron n) )
    (isLayer (cons n l)) ) )

(defthmd same-neuron-len-in-after
  (implies (and (isLayer l)
    (natp j)
    (< j (len l))
    (isLayer (after j l)) )
    (equal (len (car l))
      (len (car (after j l))) ) ) )

(defthmd same-neuron-len-in-front
  (implies (and (isLayer l)
    (natp j)
    (< j (len l))
    (isLayer (front j l)) )
    (equal (len (car l))
      (len (car (front j l))) ) ) )

(defthmd undo-app
  (equal (cons a (append b c))
    (append (cons a b) c) ) )

(defthmd not-isLayer-cdr-nil
  (implies (and (isLayer l)
    (not (isLayer (cdr l))) )
    (equal (cdr l) nil) ) )

(defthmd list-car-layer
  (implies (isLayer l)
    (isLayer (list (car l))) ) )

(defthmd not-front-cdr-helper
  (implies (and (not (cdr (front j l)))
    (natp j)
    (< j (len l))
    (isListOfWeights i)
    (equal (len (car l)) (len i))
    (isLayer l) )
    (not (cdr (front j (forwardLayer f i l)))) )

:hints (("Goal"
  :hands-off (forwardLayer)
  :use ((:instance front-len (i j) (w l) )
    (:instance len-0 (i (cdr (front j l))) )
    isLayer-true-listp)
  :do-not-induct t)
  "Subgoal 1"
  :use (forwardLayer-len
    (:instance len-0 (i (cdr (front 1 (forwardLayer f i l)))) )
    (:instance zp-1+0 (a (len (cdr (front 1 (forwardLayer f i l)))) )
    (:instance front-len (i l) (w (forwardLayer f i l)) ) ) ) )

```

```

(defthmd forwardLayer-over-list*
  (implies (and (isLayer a)
                (isLayer b)
                (isLayer c)
                (isListOfWeights i)
                (equal (len (car a)) (len i))
                (equal (len (car a)) (len (car b)))
                (equal (len (car b)) (len (car c))) )
            (equal (forwardLayer f i (list* (car a)
                                             (car b)
                                             (cdr c)))
                  (list* (car (forwardLayer f i a))
                          (car (forwardLayer f i b))
                          (cdr (forwardLayer f i c))) ) ) )

(defthmd forwardLayer-over-after
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len (car l)) (len i))
                (natp j)
                (< j (len l)) )
            (equal (forwardLayer f i (after j l))
                  (after j (forwardLayer f i l))) ) )

(defthmd forwardLayer-true-listp
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len (car l)) (len i)) )
            (true-listp (forwardLayer f i l)) ) )

(defthmd forwardLayer-over-front
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len (car l)) (len i))
                (natp k)
                (< k (len l)) )
            (equal (forwardLayer f i (front k l))
                  (front k (forwardLayer f i l))) ) )

(defthmd forwardLayer-step-car
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len (car l)) (len i)) )
            (equal (forwardLayer f i (cons (car l) x))
                  (cons (car (forwardLayer f i l))
                        (forwardLayer f i x)) ) ) )

(defthmd car-append
  (implies (consp a)
            (equal (car (append a b))
                  (car a)) ) )

(defthmd app-isLayer
  (implies (and (isLayer a)
                (isLayer b)
                (equal (len (car a)) (len (car b))) )
            (isLayer (append a b)) )

  :hints (("Goal"
           :induct (append a b) )
          ("Subgoal *1/2.1"
           :use ( (:instance car-append (a (cdr a)) (b b) ) ) ) )

(defthmd then-cdr-front-isLayer
  (implies (and (isLayer x)
                (cdr x) )
            (isLayer (cdr x)) ) )

(defthmd isLayer-all-neurons-same
  (implies (and (isLayer x)

```

```

        (isLayer (cdr x)) )
      (equal (len (car x))
             (len (cadr x)) ) ) )

(defthmd forwardLayer-over-cdr
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len (car l)) (len i)) )
           (equal (cdr (forwardLayer f i l))
                  (forwardLayer f i (cdr l)) ) ) )

(defthmd forwardLayer-over-cons
  (implies (and (isLayer a)
                (isLayer b)
                (isListOfWeights i)
                (equal (len (car a)) (len i))
                (equal (len (car a)) (len (car b))) )
           (equal (forwardLayer f i (cons (car a) (cdr b)) )
                  (cons (car (forwardLayer f i a))
                        (cdr (forwardLayer f i b)) ) ) ) )

(defthmd forwardLayer-over-cons2
  (implies (and (isLayer a)
                (isLayer b)
                (isListOfWeights i)
                (equal (len (car a)) (len i))
                (equal (len (car a)) (len (car b))) )
           (equal (forwardLayer f i (cons (car a) b) )
                  (cons (car (forwardLayer f i a))
                        (forwardLayer f i b) ) ) ) )

(in-theory (disable CAR-CDR-ELIM))

(defthmd swap-over-forwardLayer-2.1-sub-3
  (IMPLIES (AND (ISLAYER (AFTER J L))
                (< 0 J)
                (ISLAYER L)
                (ISLISTOFWEIGHTS I)
                (EQUAL (LEN (CAR (AFTER J L))) (LEN I))
                (< 0 (LEN L))
                (INTEGERP J)
                (<= 0 J)
                (< J (LEN L)))
           (EQUAL (FORWARDLAYER F I (CONS (CAR (AFTER J L))
                                         (APPEND (CDR (FRONT J L))
                                                (CONS (CAR L) (CDR (AFTER J L))))))
                  (CONS (CAR (AFTER J (FORWARDLAYER F I L)))
                        (APPEND (CDR (FRONT J (FORWARDLAYER F I L)))
                                (CONS (CAR (FORWARDLAYER F I L))
                                      (CDR (AFTER J (FORWARDLAYER F I L)))))))

:hints ("Goal"
        :cases ( (NOT (CDR (FRONT J L))) )
        :hands-off (forwardLayer
                    isLayer
                    activate)
        :do-not-induct t
        :use (forwardLayer-over-after
              (:instance forwardLayer-step-car (f f)
                        (i i)
                        (l (after j l))
                        (x (append (cdr (front j l))
                                  (cons (car l)
                                        (cdr (after j l))))))
              (:instance forwardLayer-over-app (a (cdr (front j l)) )
                        (b (cons (car l)
                                (cdr (after j l)))) )
                        (i i)
                        (f f) ))

:in-theory (enable swap-same
              front-consp-zp

```

```

isLayer-isLayer2-duh
isLayer-consp
list-car-layer
same-neuron-len-in-after
same-neuron-len-in-front
add-neurons-to-layer
car-front
isLayer-isLayer2-duh
not-isLayer-cdr-nil
isLayer-after2
not-front-cdr-helper
forwardLayer-over-after
forwardLayer-over-list*
after-after
add-neurons-to-layer
add-neurons-to-layer2
app-isLayer
forwardLayer-over-front
isLayer-true-listp
forwardLayer-true-listp
forwardLayer-over-app
forwardLayer-step-car
isLayer-front) )

("Subgoal 2"
 :in-theory (disable isLayer-front
                    same-neuron-len-in-after)
 :use ((:instance then-cdr-front-isLayer (x (front j l)) )
       same-neuron-len-in-after
       (:instance add-neurons-to-layer2 (i (len (car (after j l))))
                  (l (cdr (after j l)))
                  (n (car l)) )
       (:instance isLayer-isLayer2-duh (l (after j l)) )
       isLayer-isLayer2-duh
       (:instance isLayer-front (i j) (l l) )
       same-neuron-len-in-front
       (:instance isLayer-all-neurons-same (x (front j l)) )
       (:instance forwardLayer-over-front (f f) (i i) (k j) (l l) )
       (:instance forwardLayer-over-cdr (f f) (i i) (l (front j l)) )
       (:instance forwardLayer-over-cons (f f) (i i) (a l) (b (after j l)))) )

("Subgoal 1"
 :use ((:instance forwardLayer-over-list* (a (after j l)
                                           (b l)
                                           (c (after j l)) )) ) )

(defthmd isLayer-list*
  (implies (and (isLayer a)
                (isLayer b)
                (isLayer c)
                (equal (len (car a)) (len (car b)))
                (equal (len (car b)) (len (car c))) )
            (isLayer (list* (car a)
                            (car b)
                            (cdr c)) ) ) )

(defthmd front-difference-isLayer
  (implies (and (< k j)
                (not (equal (+ j (- k)) l))
                (natp j)
                (natp k)
                (isLayer l)
                (< (+ j (- k)) (len l)) )
            (isLayer (cdr (front (+ j (- k)) l))) )

  :hints (("Goal"
          :use ((:instance isLayer-front (i (+ j (- k))) (l l) )
                (:instance isLayer-isLayer2-duh (l (cdr (front (+ j (- k)) l))) )
                (:instance front-len (i (+ j (- k))) (w l) ))
          :do-not-induct t)) )

(defthmd car-append
  (implies (consp a)

```

```

(equal (car (append a b))
       (car a) ) )

(defthmd add-neurons-to-layer-premises
  (implies (AND (< 1 (+ J (- K)))
               (EQUAL (AFTER (+ K J (- K)) L) (AFTER J L))
               (ISLAYER (FRONT K L))
               (EQUAL (AFTER (+ K J (- K)) (FORWARDLAYER F I L))
                      (AFTER J (FORWARDLAYER F I L)))
               (ISLAYER (AFTER J L))
               (EQUAL (LEN (CAR (FRONT K L))) (LEN I))
               (< K J)
               (ISLAYER L)
               (ISLISTOFWEIGHTS I)
               (EQUAL (LEN (CAR (AFTER J L))) (LEN I))
               (INTEGERP K)
               (<= 0 K)
               (< K (LEN L))
               (INTEGERP J)
               (<= 0 J)
               (< J (LEN L))
               (<= K J)
               (NOT (EQUAL K J)) )
           (AND (ISLAYER (APPEND (CDR (FRONT (+ J (- K)) (AFTER K L)))
                                (CONS (CAR (AFTER K L))
                                       (CDR (AFTER J L))))))
                (EQUAL (LEN (car (AFTER J L)))
                       (LEN (CAR (APPEND (CDR (FRONT (+ J (- K)) (AFTER K L))
                                             (CONS (CAR (AFTER K L))
                                                  (CDR (AFTER J L)))))))
                (ISNEURON (car (AFTER J L)) ) )

: hints ("Goal"
        :hands-off (forwardLayer
                    isLayer
                    activate)
        :use ((:instance isLayer-after2 (j k) (l l) )
              (:instance car-append (a (CDR (FRONT (+ J (- K)) (AFTER K L)))
                                     (b (CONS (CAR (AFTER K L))
                                             (CDR (AFTER J L)))) )
              (:instance same-neuron-len-in-after (j k) (l l) )
              (:instance isLayer-all-neurons-same (x (front (+ j (- k))
                                                            (after k l))) )
              (:instance after-len (i k) (w l) )
              same-neuron-len-in-after
              (:instance isLayer-isLayer2-duh (l (after j l)) )
              (:instance isLayer-isLayer2-duh (l (after k l)) )
              (:instance same-neuron-len-in-front (j (+ j (- k)) (l (after k l)) )
              (:instance front-difference-isLayer (l (after k l)) (j j) (k k) )
              (:instance add-neurons-to-layer2 (i (len (car (after k l)))
                                                (l (cdr (after j l)))
                                                (n (car (after k l))) )
              (:instance app-isLayer (a (cdr (front (+ j (- k)) (after k l)))
                                       (b (cons (car (after k l))
                                             (cdr (after j l)))) )
              (:instance isLayer-front (i (+ j (- k)) (l (after k l)) ) )
        :do-not-induct t)) )

(defthmd swap-over-forwardLayer-2.1-1.2
  (IMPLIES (AND (< 1 (+ J (- K)))
               (EQUAL (AFTER (+ K J (- K)) L) (AFTER J L))
               (ISLAYER (FRONT K L))
               (EQUAL (AFTER (+ K J (- K)) (FORWARDLAYER F I L))
                      (AFTER J (FORWARDLAYER F I L)))
               (ISLAYER (AFTER J L))
               (EQUAL (LEN (CAR (FRONT K L))) (LEN I))
               (< K J)
               (ISLAYER L)
               (ISLISTOFWEIGHTS I)
               (EQUAL (LEN (CAR (AFTER J L))) (LEN I))
               (INTEGERP K)

```

```

(<= 0 K)
(< K (LEN L))
(INTEGERP J)
(<= 0 J)
(< J (LEN L))
(<= K J)
(NOT (EQUAL K J))
(EQUAL (FORWARDLAYER F I (APPEND (FRONT K L)
                                   (CONS (CAR (AFTER J L))
                                           (APPEND (CDR (FRONT (+ J (- K))
                                                         (AFTER K L))))
                                           (CONS (CAR (AFTER K L))
                                                 (CDR (AFTER J L)))))))
        (APPEND (FRONT K (FORWARDLAYER F I L))
                 (CONS (CAR (AFTER J (FORWARDLAYER F I L))
                       (APPEND (CDR (FRONT (+ J (- K))
                                         (AFTER K
                                           (FORWARDLAYER F I L))))
                               (CONS (CAR (AFTER K (FORWARDLAYER F I L))
                                     (CDR (AFTER J (FORWARDLAYER F I L))))))))))

: hints ("Goal"
        : hands-off (forwardLayer
                    isLayer
                    activate)
        : use (add-neurons-to-layer-premises
              forwardLayer-over-front
              (:instance forwardLayer-over-front (f f) (i i) (k (+ j (- k)))
              (l (after k l)) )
              (:instance forwardLayer-over-cons2 (a (after j l))
              (b (APPEND (CDR (FRONT (+ J (- K))
                                   (AFTER K L))))
                (CONS
                 (CAR (AFTER K L))
                 (CDR (AFTER J L)))) )
              (f f)
              (i i) )
              (:instance forwardLayer-over-cons (a (after k l)) (b (after j l))
              (f f) (i i) )
              (:instance isLayer-all-neurons-same (x (after k l)) )
              (:instance isLayer-all-neurons-same (x (front (+ j (- k))
              (after k l))) )
              (:instance isLayer-after2 (j k) (l l) )
              (:instance isLayer-front (i (+ j (- k))) (l (after k l)) )
              (:instance forwardLayer-over-cdr (f f) (i i) (l (front (+ j (- k))
              (after k l))) )
              same-neuron-len-in-after
              (:instance same-neuron-len-in-after (j k) (l l) )
              (:instance same-neuron-len-in-front (j (+ j (- k))) (l (after k l)) )
              (:instance isLayer-isLayer2-duh (l (after j l)) )
              (:instance isLayer-isLayer2-duh (l (after k l)) )
              (:instance add-neurons-to-layer2 (i (len (car (after k l)))
              (l (cdr (after j l)))
              (n (car (after k l)))) )
              (:instance forwardLayer-over-after (l l) (i i) (f f) (j k) )
              forwardLayer-over-after
              (:instance after-len (i k) (w l) )
              (:instance front-difference-isLayer (l (after k l)) (j j) (k k) )
              (:instance forwardLayer-over-app (a (cdr (front (+ j (- k))
              (after k l))) )
              (b (cons (car (after k l))
              (cdr (after j l)) ) )
              (i i)
              (f f) )
              (:instance forwardLayer-over-app (a (front k l) )
              (b (cons (car (after j l))
              (append
               (cdr (front (+ j (- k))
                           (after k l)))
               (cons (car (after k l))
                     (cdr

```

```

                                                    (after j l))))))
      (i i)
      (f f)
      (:instance add-neurons-to-layer (n (car (after j l)))
        (l (append (cdr (front (+ j (- k))
          (after k l)))
            (cons (car (after k l))
              (cdr (after j
                l)))))) ))
      :do-not-induct t)) )
(defthmd true-listp-front
  (true-listp (front i x)) )
(defthmd is-nil-duh
  (implies (and (equal (len x) 0)
    (true-listp x) )
    (null x)) )
(defthmd nil-cdr-front-1
  (implies (consp x)
    (equal (cdr (front 1 x)) nil) )
  :hints (("Goal"
    :use ((:instance is-nil-duh (x (cdr (front 1 x))) )
      (:instance true-listp-front (i 1) (x x) )
      (:instance zp-1+0 (a (len (cdr (front 1 x)))) )
      (:instance len-1 (i x) )
      (:instance front-len (i 1) (w x) )) ) )
(defthmd after-consp-duh
  (implies (and (natp k)
    (< k (len l)) )
    (consp (after k l)) ) )
(defthmd swap-over-forwardLayer-2.1
  (implies (and (< k j)
    (isLayer l)
    (isListOfWeights i)
    (equal (len (car l)) (len i))
    (integerp k)
    (<= 0 k)
    (< k (len l))
    (integerp j)
    (<= 0 j)
    (< j (len l))
    (<= k j)
    (not (equal k j)))
    (equal (forwardLayer f i (append (front k l)
      (cons (car (after (+ k j (- k)) l))
        (append (cdr (front (+ j (- k))
          (after k l)))
            (cons (car (after k l))
              (cdr
                (after (+ k
                  j
                    (- k))
                  l))))))
      (append (front k (forwardLayer f i l))
        (cons (car (after (+ k j (- k)) (forwardLayer f i l)))
          (append (cdr (front (+ j (- k))
            (after k (forwardLayer f i l))))
              (cons (car (after k (forwardLayer f i l)))
                (cdr (after (+ k j (- k))
                  (forwardLayer f i l))))))))))
  :hints (("Goal"
    :hands-off (forwardLayer
      isLayer
      activate)
    :do-not-induct t

```

```

:use (swap-over-forwardLayer-2.1-sub-3
      (:instance after-after (a (+ j (- k))) (b k) (x l) )
      (:instance isLayer-front (i k) (l l) )
      (:instance after-after (a (+ j (- k))) (b k) (x (forwardLayer f i l)) )
      isLayer-after2
      same-neuron-len-in-after)
:in-theory (enable swap-same
                front-consp-zp
                isLayer-isLayer2-duh
                isLayer-consp
                list-car-layer
                same-neuron-len-in-after
                same-neuron-len-in-front
                add-neurons-to-layer
                car-front
                isLayer-isLayer2-duh
                not-isLayer-cdr-nil
                isLayer-after2
                not-front-cdr-helper
                forwardLayer-over-after
                forwardLayer-over-list*
                after-after
                add-neurons-to-layer
                add-neurons-to-layer2
                app-isLayer
                forwardLayer-over-front
                isLayer-true-listp
                forwardLayer-true-listp
                forwardLayer-over-app
                forwardLayer-step-car
                isLayer-front) )

("Subgoal 1"
 :cases ( (< 1 (- j k))
          (equal 1 (- j k)) ) )
("Subgoal 1.2"
 :use (swap-over-forwardLayer-2.1-1.2))
("Subgoal 1.1"
 :use (after-consp-duh
      forwardLayer-over-front
      (:instance same-neuron-len-in-after (j k) (l l) )
      (:instance isLayer-list* (a (after j l)) (b (after k l))
                            (c (after j l)) )
      (:instance forwardLayer-over-app (a (front k l) )
                                       (b (LIST* (CAR (AFTER J L))
                                                (CAR (AFTER K L))
                                                (CDR (AFTER J L))))
                                       (i i)
                                       (f f) )
      (:instance forwardLayer-over-list* (a (after j l))
                                       (b (after k l))
                                       (c (after j l)) )
      (:instance forwardLayer-over-after (l l) (i i) (f f) (j k) )
      forwardLayer-over-after
      forwardLayer-len
      (:instance after-consp-duh (k k) (l (forwardLayer f i l)) )
      (:instance nil-cdr-front-1 (x (after k l)) )
      (:instance isLayer-after2 (j k) (l l) )
      (:instance nil-cdr-front-1 (x (after k (forwardLayer f i l))) ) ) ) )

(defthmd zero-j-duh
  (implies (natp k)
    (equal (+ (- j) j k) k) ) )

(defthmd ineq-jk-duh
  (implies (< j k)
    (< 0 (+ (- j) k)) ) )

(defthmd isLayer-<-2
  (implies (and (isLayer l)
    (<= 2 (len l)) )
    (isLayer (cdr l)) ) )

```

```

(defthmd trans-<
  (implies (and (< a b) (< b c)) (< a c)) )

(defthmd cdr-after-isLayer
  (implies (and (isLayer (after j l))
                (natp j)
                (< j k)
                (natp k)
                (< k (len l)))
            (isLayer (cdr (after j l)))) )

  :hints (("Goal"
           :in-theory (enable cdr-after
                               after-len
                               len-1
                               len-0)
           :do-not-induct t)) )

(defthmd cdr-front-1-nil
  (equal (cdr (front 1 x)) nil)

  :hints (("Goal"
           :cases ( (consp x) )
           :expand ( (front 1 x) )
           :in-theory (enable car-front
                               front-len
                               true-listp-front
                               len-0)
           :do-not-induct t)) )

(defthmd j-zero-duh
  (equal (+ (- j) 1 j) 1) )

(defthmd specific-cdr-front-1-nil
  (equal (cdr (front (+ (- j) 1 j) (after j l))) nil)

  :hints (("Goal"
           :do-not-induct t
           :use (j-zero-duh
                 (:instance cdr-front-1-nil (x (after j l)))))) )

(defthmd list-isLayer
  (implies (and (isNeuron x)
                (isNeuron y)
                (equal (len x) (len y)))
            (isLayer (list x y)) ) )

(defthmd list-isLayer1
  (implies (isNeuron x)
            (isLayer (list x)) ) )

(defthmd car-after-cdr-helper
  (implies (and (isLayer l)
                (natp j)
                (< j (len l)))
            (iff (EQUAL (LEN (CAR (AFTER (+ 1 J) L)))
                       (LEN (CAR (AFTER J L))))
                 (EQUAL (LEN (CAR (AFTER J L)))
                       (LEN (CADR (AFTER J L)))))) ) )

  :hints (("Goal"
           :use (isLayer-true-listp
                 (:instance cdr-after2 (i j) (x l)))
           :do-not-induct t)) )

(defthmd list-cdr-isLayer-helper
  (implies (and (isLayer l)
                (natp j)
                (< j (len l)))
            (iff (ISLAYER (LIST (CAR (AFTER (+ 1 J) L)))

```

```

                (CAR (AFTER J L)))
      (ISLAYER (LIST (CADR (AFTER J L))
                    (CAR (AFTER J L)))) ) )

:hints ("Goal"
       :use (isLayer-true-listp
             (:instance cdr-after2 (i j) (x l) ))
       :do-not-induct t))

(defthmd swap-isLayer-2.3.1
  (IMPLIES (AND (TRUE-LISTP L)
                (EQUAL (+ (- J) J K) K)
                (< 0 (+ (- J) K))
                (EQUAL (AFTER (+ (- J) K) (AFTER J L))
                       (AFTER K L))
                (ISLAYER (AFTER K L))
                (< 2 K)
                (< J K)
                (ISLAYER L)
                (INTEGERP K)
                (<= 0 K)
                (< K (LEN L))
                (INTEGERP J)
                (<= 0 J)
                (< J (LEN L))
                (<= J K)
                (NOT (EQUAL J K)))
            (ISLAYER (APPEND (FRONT J L)
                             (CONS (CAR (AFTER K L))
                                   (APPEND (CDR (FRONT (+ (- J) K) (AFTER J L)))
                                           (CONS (CAR (FRONT (+ (- J) K) (AFTER J L)))
                                               (CDR (AFTER K L))))))))))

:hints ("Goal"
       :in-theory (enable car-front
                        after-len)
       :hands-off (isLayer
                  len)
       :cases ( (equal j 0) )
       :use (isLayer-<-2
             (:instance isLayer-<-2 (l (front k l) )
                                   (:instance trans-< (a 2) (b k) (c (len l)) )
                                   (:instance same-neuron-len-in-after (j k) (l l) )
                                   (:instance same-neuron-len-in-front (j k) (l l) )
                                   (:instance isLayer-front (i j) (l l) )
                                   (:instance isLayer-front (i k) (l l) )
                                   same-neuron-len-in-front
                                   (:instance isLayer-isLayer2-duh (l (after k l)) )
                                   (:instance front-len (i k) (w l) )
                                   isLayer-isLayer2-duh)
             :do-not-induct t)
       ("Subgoal 2"
        :use (isLayer-after2
              same-neuron-len-in-after
              (:instance after-len (i j) (w l) )
              (:instance front-len (i (+ (- j) k)) (w (after j l)) )
              (:instance isLayer-isLayer2-duh (l (after j l)) )
              (:instance add-neurons-to-layer2 (n (car (after j l)))
                                                (i (len (car (after j l))))
                                                (l (cdr (after k l))))))
        ("Subgoal 2.2"
         :cases ( (>= 1 (+ (- j) k)) )
         :use (cdr-after-isLayer
               (:instance isLayer-isLayer2-duh (l (after (+ 1 j) l)) )
               (:instance list-isLayer (x (CAR (AFTER (+ 1 J) L))
                                         (y (CAR (AFTER J L))))
               (:instance isLayer-all-neurons-same (x (after j l)) )
               (:instance isLayer-all-neurons-same (x (after k l)) )
               (:instance cdr-after2 (i j) (x l) )
               (:instance isLayer-front (i (+ (- j) k)) (l (after j l)) )
               (:instance app-isLayer (a (front j l))

```

```

(b (LIST (CAR (AFTER (+ 1 J) L))
         (CAR (AFTER J L))) ) ) )
"Subgoal 2.2.2"
:use (car-after-cdr-helper
      list-cdr-isLayer-helper
      (:instance front-difference-isLayer (j k) (k j) (l (after j l)) )
      (:instance list-isLayer1 (x (car (after j l))) )
      (:instance car-append (a (CDR (FRONT (+ (- J) K) (AFTER J L))) )
                            (b (cons (CAR (AFTER J L))
                                       (cdr (after k l))) ) )
      (:instance app-isLayer (a (CDR (FRONT (+ (- J) K) (AFTER J L))) )
                              (b (cons (CAR (AFTER J L))
                                       (cdr (after k l))) ) )
      (:instance add-neurons-to-layer (n (CAR (AFTER K L)) )
                                       (l (APPEND (CDR (FRONT (+ (- J) K)
                                                           (AFTER J L)))
                                                (cons (CAR (AFTER J L))
                                                    (cdr (after k l)))))) )
      (:instance app-isLayer (a (front j l))
                              (b (CONS (CAR (AFTER K L))
                                       (APPEND (CDR (FRONT (+ (- J) K)
                                                           (AFTER J L)))
                                                (cons (CAR (AFTER J L))
                                                    (cdr (after k l)))))))) )
"Subgoal 2.2.1"
:use (specific-cdr-front-l-nil
      (:instance isLayer-list* (a (cdr (after j l)))
                               (b (after j l))
                               (c (cdr (after k l))) )
      (:instance isLayer-list* (a (after (+ 1 j) l))
                               (b (after j l))
                               (c (after (+ 1 j) l)) )
      (:instance app-isLayer (a (front j l))
                              (b (LIST* (CAR (AFTER (+ 1 J) L))
                                         (CAR (AFTER J L))
                                         (CDR (AFTER (+ 1 J) L))) ) ) )
"Subgoal 1"
:use ((:instance isLayer-isLayer2-duh (l (front k l)) )
      (:instance isLayer-isLayer2-duh (l (cdr (front k l))) )
      (:instance add-neurons-to-layer2 (n (car l) )
                                       (i (len (car (after k l))) )
                                       (l (cdr (after k l))) ) )
      (:instance isLayer-all-neurons-same (x (front k l)) )
      (:instance app-isLayer (a (cdr (front k l))
                              (b (cons (car l)
                                       (cdr (after k l))) ) )
      (:instance car-append (a (CDR (FRONT K L))
                              (b (CONS (CAR L)
                                       (CDR (AFTER K L)))) ) )
      (:instance add-neurons-to-layer (n (car (after k l))
                                       (l (APPEND (CDR (FRONT K L))
                                                (CONS
                                                  (CAR L)
                                                  (CDR (AFTER K L))))))) ) ) )
(defthmd swap-isLayer-sub
  (implies (and (isLayer l)
                (< j k)
                (natp k)
                (< k (len l))
                (natp j)
                (< j (len l)) )
           (isLayer (swap l j k)) )
:hints (("Goal"
:hands-off (isLayer)
:do-not-induct t
:cases ( (< 2 k)
         (equal k 2)
         (equal k 1) )
:use (isLayer-true-listp

```

```

zero-j-duh
ineq-jk-duh
(:instance after-after (a (+ (- j) k)) (b j) (x l) )
(:instance isLayer-after2 (j k) (l l) )
swap-isLayer-2.3.1) )
("Subgoal 2"
:use ((:instance isLayer-front (i j) (l l) )
      isLayer-after2)
:cases ( (equal j 0) ) )
("Subgoal 2.2"
:use ((:instance isLayer-front (i 1) (l (after 1 l)) )
      (:instance cdr-front-1-nil (x (after 1 l)) )
      (:instance same-neuron-len-in-after (j 2) (l l) )
      (:instance same-neuron-len-in-after (j 1) (l l) )
      (:instance same-neuron-len-in-front (j 1) (l l) )
      (:instance same-neuron-len-in-front (j 1) (l (after 1 l)) )
      (:instance after-len (i 1) (w l) )
      (:instance isLayer-list* (a (after 2 l) )
                              (b (FRONT 1 (AFTER 1 L)) )
                              (c (AFTER 2 L) ) )
      (:instance app-isLayer (a (front 1 l) )
                              (b (LIST* (CAR (AFTER 2 L))
                                           (CAR (FRONT 1 (AFTER 1 L)))
                                           (CDR (AFTER 2 L))) ) ) )
      )
("Subgoal 2.1"
:in-theory (enable car-front)
:use ((:instance front-len (i 2) (w l) )
      (:instance isLayer-<-2 (l (front 2 l)) )
      (:instance isLayer-all-neurons-same (x (front 2 l)) )
      (:instance true-listp-front (i 2) (x l) )
      (:instance isLayer-front (i 2) (l l) )
      isLayer-isLayer2-duh
      (:instance isLayer-isLayer2-duh (l (after 2 l)) )
      (:instance same-neuron-len-in-after (j 2) (l l) )
      (:instance same-neuron-len-in-front (j 2) (l l) )
      (:instance then-cdr-front-isLayer (x (front 2 l)) )
      (:instance add-neurons-to-layer2 (n (car l))
                                       (i (len (car (after 2 l))) )
                                       (l (cdr (after 2 l)) ) )
      (:instance app-isLayer (a (cdr (front 2 l)) )
                              (b (CONS (CAR L)
                                       (CDR (AFTER 2 L))) ) )
      (:instance car-append (a (cdr (front 2 l)) )
                              (b (CONS (CAR L)
                                       (CDR (AFTER 2 L))) ) )
      (:instance add-neurons-to-layer (n (car (after 2 l)))
                                       (l (APPEND (CDR (FRONT 2 L))
                                                  (CONS
                                                    (CAR L)
                                                    (CDR (AFTER 2 L)))) ) ) )
      )
("Subgoal 1"
:in-theory (enable car-front)
:use ((:instance isLayer-front (i 1) (l l) )
      (:instance isLayer-isLayer2-duh (l (after 1 l)) )
      (:instance cdr-front-1-nil (x l) )
      (:instance same-neuron-len-in-after (j 1) (l l) )
      (:instance same-neuron-len-in-front (j 1) (l l) )
      (:instance isLayer-list* (a (after 1 l) )
                              (b l)
                              (c (after 1 l)) )
      (:instance add-neurons-to-layer2 (n (car l))
                                       (i (len (car (after 1 l))) )
                                       (l (cdr (after 1 l)) ) ) )
      )
(defthmd swap-isLayer
  (implies (and (isLayer l)
                (natp k)
                (< k (len l))
                (natp j)
                (< j (len l)) )
           (isLayer (swap l j k)) )

```

```

:hints ("Goal"
  :use (swap-isLayer-sub
    (:instance swap-isLayer-sub (j k) (k j) (l l) ))
  :hands-off (isLayer)
  :do-not-induct t)) )

(defthmd same-len-neuron-in-swap
  (implies (and (isLayer l)
    (natp k)
    (< k (len l))
    (natp j)
    (< j (len l)) )
    (equal (len (car l))
      (len (car (swap l j k))) ) ) )

:hints ("Goal"
  :hands-off (isLayer)
  :use (:instance isLayer-front (i k) (l l) )
    (:instance isLayer-front (i j) (l l) )
    isLayer-after2
    (:instance isLayer-after2 (j k) (l l) )
    zero-j-duh
    same-neuron-len-in-after
    (:instance same-neuron-len-in-after (j k) (l l) )
    isLayer-true-listp
    (:instance after-after (a (+ (- j) k) (b j) (x l) )
      same-neuron-len-in-front
      (:instance same-neuron-len-in-front (j k) (l l) )
      (:instance car-append (a (FRONT J L))
        (b (CONS (CAR (AFTER (+ (- J) K) (AFTER J L)))
          (APPEND (CDR (FRONT (+ (- J) K)
            (AFTER J L)))
              (CONS (CAR (FRONT (+ (- J) K)
                (AFTER J L)))
                  (CDR (AFTER
                    (+ (- J) K)
                    (AFTER J L)))))) ) )
          (:instance car-append (a (FRONT K L))
            (b (CONS (CAR (AFTER (+ J (- K)) (AFTER K L)))
              (APPEND (CDR (FRONT (+ J (- K))
                (AFTER K L)))
                  (CONS (CAR (FRONT (+ J (- K))
                    (AFTER K L)))
                      (CDR (AFTER
                        (+ J (- K))
                        (AFTER K L)))))) ) )
          :do-not-induct t)) ) )

(defthmd append-len
  (equal (len (append x y))
    (+ (len x)
      (len y) ) ) )

(defthmd neg-neg
  (implies (natp j)
    (equal (- (- j)) j) ) )

(defthmd zero-k-duh
  (implies (natp j)
    (equal (+ K J (- K)) J) ) )

(defthmd swap-len
  (implies (and (true-listp l)
    (natp j)
    (natp k)
    (< j (len l))
    (< k (len l)) )
    (equal (len l)
      (len (swap l j k)) ) ) )

```

```

:hints ("Goal"
  :in-theory (enable append-len
                    after-after
                    front-len
                    after-len)

  :use (neg-neg
        zero-k-duh
        (:instance len-cdr (w (after j l)) )
        (:instance len-cdr (w (FRONT (+ J (- K)) (AFTER K L))) )
        (:instance front-len (i k) (w l) )
        (:instance len-cdr (w (after k l)) )
        (:instance len-cdr (w (FRONT (+ (- J) K) (AFTER J L))) )
        (:instance zero-j-duh (k k) (j (- j)) )
        (:instance front-len (i j) (w l) ))

  :do-not-induct t
  :cases ( (< j k)
           (< k j) )) )

;-----

; 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.

(defthmd swap-over-forwardLayer
  (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 (swap l j k))
                   (swap (forwardLayer f i l) j k) ) )

  :hints ("Goal"
    :hands-off (forwardLayer
                isLayer
                activate)
    :cases ( (< k j)
             (< j k) )
    :in-theory (enable swap-same
                      front-consp-zp
                      isLayer-isLayer2-duh
                      isLayer-consp
                      list-car-layer
                      same-neuron-len-in-after
                      same-neuron-len-in-front
                      add-neurons-to-layer
                      car-front
                      isLayer-isLayer2-duh
                      not-isLayer-cdr-nil
                      isLayer-after2
                      not-front-cdr-helper
                      forwardLayer-over-after
                      forwardLayer-over-list*
                      after-after
                      add-neurons-to-layer
                      add-neurons-to-layer2
                      app-isLayer
                      forwardLayer-over-front
                      isLayer-true-listp
                      forwardLayer-true-listp
                      forwardLayer-over-app
                      forwardLayer-step-car
                      isLayer-front)

    :do-not-induct t)
  ("Subgoal 2.1"
   :use (swap-over-forwardLayer-2.1) )
  ("Subgoal 1.1"
   :use ((:instance swap-over-forwardLayer-2.1 (l l) (f f)

```

```
(i i) (k j) (j k) ))))
```

```
; 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.
```

```
(defthmd neuron-swap-in-layer  
  (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)) )  
  
  :hints (("Goal"  
          :hands-off (forwardLayer  
                     isLayer  
                     swap  
                     activate)  
          :use ((:instance swap-over-forwardLayer (f f) (i i)  
                                                       (j j) (k k) (l (swap l j k)) )  
              isLayer-true-listp  
              swap-len  
              same-len-neuron-in-swap  
              swap-isLayer  
              (:instance swap-double (x l) (i j) (j k) ))  
          :do-not-induct t)) )
```

```
;-----
```

```
****
```

form3.lisp

```
****
```

```
; Form Book 3 - More Theorems for Rearranging Neural Networks  
(in-package "ACL2")
```

```
(include-book "form2")
```

```
;-----
```

```
; swapNeuronSynapsesInLayer swaps the synapses in two positions j and k  
; for each neuron in a given layer l.
```

```
(defun swapNeuronSynapsesInLayer (l j k)  
  (if (consp l)  
      (cons (swap (car l) j k)  
            (swapNeuronSynapsesInLayer (cdr l) j k))  
      nil))
```

```
;-----
```

```
; The following are several uninteresting lemmas. Skip ahead  
; to find interesting theorems.
```

```
(defthmd len-2  
  (implies (consp (cdr l))  
           (<= 2 (len l)) )  
  
  :hints (("Goal"  
          :in-theory (enable len-0  
                             len-1) )) )
```

```
(defthmd synapse-swap-in-neurons-of-layer-sub-3  
  (IMPLIES (AND (< J K)  
                (CONSP L)  
                (CONSP (CAR L))  
                (RATIONALP (CAAR L))  
                (ISLISTOFWEIGHTS (CDAR L))
```

```

      (ISLAYER L)
      (ISLISTOFWEIGHTS I)
      (EQUAL (LEN (CAR L)) (LEN I))
      (INTEGERP K)
      (<= 0 K)
      (< K (LEN (CAR L)))
      (INTEGERP J)
      (<= 0 J)
      (< J (LEN (CAR L)))
    (EQUAL (FORWARDLAYER F I L)
           (FORWARDLAYER F (SWAP I J K)
                           (SWAPNEURONSYNAPSESINLAYER L J K))))

:hints (("Goal"
        :hands-off ( swap
                     isLayer )
        :in-theory (enable after-after
                          after-len
                          front-len)
        :induct (swapNeuronSynapsesInLayer l j k) )
 ("Subgoal *1/"
 :expand ( (SWAPNEURONSYNAPSESINLAYER L J K) )
 :cases ( (consp (cdr l)) )
 :use (isLayer-isLayer2-duh
       (:instance forwardNeuron-swap-swap (i i) (n (car l)) (j j) (k k) ))
 :do-not-induct t)
 ("Subgoal *1/2"
 :use ((:instance swap-same (i j) (x i) )
       (:instance swap-same (i j) (x (car l)) )) )
 ("Subgoal *1/1"
 :use (isLayer-<-2
       len-2
       (:instance isLayer-isLayer2-duh (l (cdr l)) )) ) )

(defthmd commutative-swapNeuronSynapsesInLayer
  (implies (and (isLayer l)
                (natp k)
                (< k (len (car l)))
                (natp j)
                (< j (len (car l))) )
            (equal (swapNeuronSynapsesInLayer l j k)
                   (swapNeuronSynapsesInLayer l k j)) )

  :hints (("Goal"
          :induct (swapNeuronSynapsesInLayer l j k) )) )

(defthmd swapNeuronSynapsesInLayer-same
  (implies (and (isLayer l)
                (natp j)
                (< j (len (car l))) )
            (equal (swapNeuronSynapsesInLayer l j j) l) )

  :hints (("Goal"
          :expand ( (swapNeuronSynapsesInLayer l j j) )
          :induct (swapNeuronSynapsesInLayer l j j) )
 ("Subgoal *1/1"
 :do-not-induct t
 :use ((:instance isLayer-all-neurons-same (x l) )
       (:instance swap-same (i j) (x (car l)) )) ) )

;-----

; 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.

(defthmd synapse-swap-in-neurons-of-layer
  (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)) ) )

:hints ("Goal"
  :hands-off ( forwardLayer
              isLayer
              swap )
  :use (isLayer-isLayer2-duh
      swapNeuronSynapsesInLayer-same
      (:instance swapNeuronSynapsesInLayer-same (l (cdr l)) (j j) )
      (:instance isLayer-isLayer2-duh (l (cdr l)) )
      (:instance isLayer-all-neurons-same (x l) )
      (:instance swap-commutative (x i) (i j) (j k) )
      (:instance swap-commutative (x (car l)) (i j) (j k) )
      (:instance synapse-swap-in-neurons-of-layer-sub-3 (f f) (i i)
        (l l) (j k) (k j) )
      (:instance commutative-swapNeuronSynapsesInLayer (k k) (j j)
        (l (cdr l)) )

      synapse-swap-in-neurons-of-layer-sub-3)
  :cases ( (< j k)
          (< k j)
          (equal j k) )
  :do-not-induct t)
("Subgoal 1"
  :use ((:instance swap-same (x i) (i j) )
      swapNeuronSynapsesInLayer-same) ) )

;-----
****
form4.lisp
****
; Form Book 4 - More Theorems for Rearranging Neural Networks
(in-package "ACL2")

(include-book "form3")

;-----

; swapLayerNeuronsInNetwork swaps the neurons at positions j and k
; within the l-th layer (from 0) of a given network n, and updates
; the following layer (if there is one) so that the outputs of the
; swapped neurons still go to the same inputs in the next layer.

(defun swapLayerNeuronsInNetwork2 (n j k)
  (if (consp n)
      (cons (swapNeuronSynapsesInLayer (car n) j k)
            (cdr n))
      nil))

(defun swapLayerNeuronsInNetwork (n l j k)
  (if (zp l)
      (cons (swap (car n) j k)
            (swapLayerNeuronsInNetwork2 (cdr n) j k))
      (cons (car n)
            (swapLayerNeuronsInNetwork (cdr n) (- l 1) j k)) ) )

;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defthmd isNetwork-isNetwork2-duh
  (iff (isNetwork n)
      (and (consp n)
           (isLayer (car n))
           (isNetwork2 (cdr n) (len (car n))) ) ) )

```

```

(defthmd swapLayerNeuronsInNetwork2-input-swap
  (implies (and (isNetwork n)
                (isListOfWeights i)
                (equal (len (caar n)) (len i))
                (natp k)
                (< k (len (car (nth 0 n))))
                (natp j)
                (< j (len (car (nth 0 n)))) )
            (equal (forwardNetwork f i n)
                  (forwardNetwork f (swap i j k)
                                     (swapLayerNeuronsInNetwork2 n j k)) ) ) )

  :hints (("Goal"
           :hands-off ( swap
                        isNetwork
                        isLayer
                        forwardLayer )
           :expand ( (forwardNetwork f i n)
                     (swapLayerNeuronsInNetwork2 n j k) )
           :use (isNetwork-isNetwork2-duh
                 (:instance synapse-swap-in-neurons-of-layer (j j) (k k) (f f) (i i)
                          (l (car n)) ) )
           :do-not-induct t)) )

(defthmd isNetwork-cdr
  (implies (and (isNetwork n)
                (< 0 (- (len n) 1)) )
            (isNetwork (cdr n)) )

  :hints (("Goal"
           :do-not-induct t)) )

(defthmd isSynapse-forwardNeuron
  (implies (and (isNeuron n)
                (isListOfWeights i)
                (equal (len i) (len n)) )
            (isSynapse (forwardNeuron i n)) ) )

(defthmd isSynapse-activate
  (implies (and (isNeuron n)
                (isListOfWeights i)
                (equal (len i) (len n)) )
            (isSynapse (activate f i n)) )

  :hints (("Goal"
           :use (isSynapse-forwardNeuron) )) )

(defthmd isListOfWeights-forwardLayer
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len i) (len (car l)))) )
            (isListOfWeights (forwardLayer f i l)) )

  :hints (("Goal"
           :hands-off ( activate )
           :expand ( (forwardLayer f i l) )
           :induct (forwardLayer f i l) )
          ("Subgoal *1/1"
           :use ((:instance isSynapse-activate (f f) (i i) (n (car l)) ) )
           :do-not-induct t)) )

(defthmd neuron-swap-in-first-layer-of-network
  (implies (and (isNetwork n)
                (isListOfWeights i)
                (equal (len (caar n)) (len i))
                (< 0 (- (len n) 1))
                (natp k)
                (< k (len (nth 0 n)))
                (natp j)
                (< j (len (nth 0 n)))) )

```

```

(equal (forwardNetwork f i n)
       (forwardNetwork f i (swapLayerNeuronsInNetwork n 0 j k)) ) )

:hints ("Goal"
       :expand ( (forwardNetwork f i n) )
       :use (isNetwork-isNetwork2-duh
            isNetwork-cdr
            (:instance forwardLayer-len (f f) (i i) (l (car n)) )
            (:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) )
            (:instance isNetwork-isNetwork2-duh (n (cdr n)) )
            (:instance swap-over-forwardLayer (f f) (i i) (j j) (k k) (l (car n)) )
            (:instance swapLayerNeuronsInNetwork2-input-swap
            (f f) (i (forwardLayer f i (car n))) (j j) (k k) (n (cdr n)) ) )
       :hands-off ( swap
                   swapLayerNeuronsInNetwork2
                   isNetwork
                   isLayer
                   forwardLayer )
       :do-not-induct t ) )

(defthmd nth-1-cdr
  (implies (and (consp n)
                (< 1 (len n))
                (not (zp 1))
                (natp 1))
           (equal (nth 1 n) (nth (+ -1 1) (cdr n))) ) )

(defthmd 1-ineq-duh
  (implies (and (integerp 1)
                (< 0 1) )
           (<= 0 (- 1 1)) ) )

(defthmd 1-is-1-helper
  (implies (and (<= (+ -1 L) 0)
                (INTEGERP L)
                (< 0 L)
                (< 1 (- (len n) 1)) )
           (and (<= 1 1)
                (<= 1 1)) ) )

(defthmd front-after-swapLayerNeuronsInNetwork
  (implies (and (isNetwork n)
                (natp 1)
                (< 1 (- (len n) 1))
                (natp k)
                (< k (len (nth 1 n)))
                (natp j)
                (< j (len (nth 1 n))) )
           (equal (swapLayerNeuronsInNetwork n 1 j k)
                  (append (front 1 n)
                           (cons (swap (nth 1 n) j k)
                                   (cons (swapNeuronSynapsesInLayer (nth (+ 1 1) n) j k)
                                         (after (+ 1 2) n)))) ) )

:hints ("Goal"
       :induct (swapLayerNeuronsInNetwork n 1 j k)
       :hands-off (swap
                   swapNeuronSynapsesInLayer
                   isNetwork
                   isLayer) )
("Subgoal *1/2"
 :use (nth-1-cdr
       1-is-1-helper
       1-ineq-duh
       isNetwork-cdr
       isNetwork-isNetwork2-duh)
 :do-not-induct t )
("Subgoal *1/1"
 :do-not '(generalize)
 :use (isNetwork-cdr
       isNetwork-isNetwork2-duh)

```

```

      :do-not-induct t )) )

(defthmd isNetwork2-true-listp
  (implies (isNetwork2 n i)
    (true-listp n) ) )

(defthmd isNetwork-true-listp
  (implies (isNetwork n)
    (true-listp n) )

  :hints (("Goal"
    :use ( (:instance isNetwork2-true-listp (n n) (i (len (car n))) ) ) ) ) )

(defthmd l-<-1
  (implies (and (NOT (EQUAL L 1))
    (NOT (ZP L))
    (NATP L) )
    (< 1 1) ) )

(defthmd isNetwork-to-isNetwork2
  (implies (isNetwork n)
    (isNetwork2 n (len (caar n))) ) )

(defthmd add-layer-to-network
  (implies (and (isLayer l)
    (isNetwork n)
    (equal (len l) (len (caar n))) )
    (isNetwork (cons l n)) ) )

(defthmd front-isNetwork
  (implies (and (natp l)
    (< 0 l)
    (<= 1 (len n))
    (isNetwork n) )
    (isNetwork (front l n)) )

  :hints (("Goal"
    :induct (front l n) )
    ("Subgoal *1/2"
    :cases ( (equal l 1)
      (not (equal l 1)) ) )
    :do-not-induct t)
    ("Subgoal *1/2.1"
    :do-not '(generalize)
    :in-theory (enable car-front)
    :hands-off (isNetwork
      isLayer)
    :use (l-<-1
      isNetwork-cdr
      (:instance add-layer-to-network (l (car n))
        (n (front (+ -1 l) (cdr n))) )
      isNetwork-isNetwork2-duh))
    ("Subgoal *1/1"
    :do-not-induct t)) ) )

(defthmd after-isNetwork
  (implies (and (natp l)
    (< 1 (len n))
    (isNetwork n) )
    (isNetwork (after l n)) ) )

(defthmd isNetwork-cdr-nil
  (implies (and (isNetwork n)
    (not (isNetwork (cdr n))) )
    (equal (cdr n) nil) )

  :hints (("Goal"
    :use (isNetwork-true-listp) ) ) )

(defthmd cdr-nth
  (implies (and (natp l)

```

```

        (< 0 1)
        (< 1 (len a))
        (consp a) )
    (equal (nth 1 a)
           (nth (- 1 1) (cdr a)) ) ) )

(defthmd just-len-cdr
  (equal (+ -1 1 (LEN (CDR A)))
         (LEN (CDR A)) ) )

(defthmd isNetwork-app
  (implies (and (isNetwork a)
                (isNetwork b)
                (equal (len (nth (- (len a) 1) a))
                       (len (caar b)) ) )
           (isNetwork (append a b)) )

  :hints (("Goal"
           :induct (append a b) )
           ("Subgoal *1/2"
            :in-theory (enable len-0
                                car-append
                                len-1)
            :hands-off (isNetwork
                        isLayer)
            :use ((:instance isNetwork-isNetwork2-duh (n a) )
                  (:instance cdr-nth (a a) (1 (- (len a) 1)) )
                  just-len-cdr
                  (:instance add-layer-to-network (1 (car a))
                                                    (n (APPEND (CDR A) B)) )
                  (:instance isNetwork-cdr-nil (n a) )
                  :do-not '(generalize)
                  :do-not-induct t)
           ("Subgoal *1/1"
            :do-not-induct t)) )

(defthmd cdr-append
  (implies (consp a)
           (equal (append (cdr a) b)
                  (cdr (append a b)) ) ) )

(defthmd forwardNetwork-over-app
  (implies (and (isNetwork a)
                (isNetwork b)
                (isNetwork (append a b))
                (isListOfWeights i)
                (equal (len i) (len (caar a))) )
           (equal (forwardNetwork f i (append a b))
                  (forwardNetwork f (forwardNetwork f i a) b)) )

  :hints (("Goal"
           :induct (forwardNetwork f i a) )
           ("Subgoal *1/2"
            :do-not-induct t)
           ("Subgoal *1/1"
            :expand ( (ISNETWORK2 (CDR A) (LEN (CAR A))) )
            :use ((:instance isListOfWeights-forwardLayer (f f) (i i) (1 (car a)) )
                  (:instance isNetwork-isNetwork2-duh (n a) )
                  cdr-append
                  (:instance forwardLayer-len (f f) (i i) (1 (car a)) )
                  (:instance isNetwork-cdr (n (append a b)) )
                  (:instance isNetwork-cdr-nil (n a) )
            :hands-off (forwardLayer
                        isNetwork
                        isLayer)
            :do-not-induct t)) )

(defthmd after-cdr
  (implies (and (natp i)
                (<= (+ 1 i) (len n))
                (consp n) )

```

```

      (equal (after (+ 1 i) n)
             (after i (cdr n)) ) )

:hints (("Goal"
        :expand ( (AFTER (+ 1 I) N) )
        :use ( (:instance zero-j-duh (j 1) (k i) ) )
        :do-not-induct t ) )

(defthmd after-all-nil
  (implies (true-listp n)
           (equal (after (len n) n) nil) )

  :hints (("Goal"
          :induct (len n) )
          ("Subgoal *1/2"
           :do-not-induct t )
          ("Subgoal *1/1"
           :use ( (:instance after-cdr (i (len (cdr n))) (n n) ) )
           :do-not-induct t ) ) )

(defthmd front-isNeuron
  (implies (and (isNeuron n)
                (natp j)
                (< 0 j)
                (< j (len n)) )
           (isNeuron (front j n)) ) )

(defthmd after-isNeuron
  (implies (and (isNeuron n)
                (natp k)
                (< k (len n)) )
           (isNeuron (after k n)) ) )

(defthmd app-isNeuron
  (implies (and (isNeuron a)
                (isNeuron b) )
           (isNeuron (append a b)) )

  :hints (("Goal"
          :induct (append a b) ) ) )

(defthmd app-isNeuron2
  (implies (and (isListOfWeights a)
                (isNeuron b) )
           (isNeuron (append a b)) )

  :hints (("Goal"
          :induct (append a b) ) ) )

(defthmd jk-trans-duh
  (implies (and (natp j)
                (natp k)
                (<= 0 j)
                (< j k) )
           (<= 1 k)) )

(defthmd isNeuron-true-listp
  (implies (isNeuron n)
           (true-listp n)) )

(defthmd isNeuron-duh
  (iff (isNeuron n)
       (and (consp n)
            (isSynapse (car n))
            (isListOfWeights (cdr n)) ) ) )

(defthmd add-synapse-to-neuron
  (implies (and (isListOfWeights n)
                (isSynapse s) )
           (isNeuron (cons s n)) ) )

```

```

(defthmd add-synapse-to-neuron2
  (implies (and (isNeuron n)
                (isSynapse s) )
           (isNeuron (cons s n)) ) )

(defthmd list*-isNeuron
  (implies (and (isSynapse a)
                (isSynapse b)
                (isListOfWeights c) )
           (isNeuron (list* a b c)) ) )

(defthmd cdr-isNeuron
  (implies (and (isNeuron n)
                (< 1 (len n)) )
           (isNeuron (cdr n)) ) )

(defthmd k-<-duh
  (implies (and (<= 1 k)
                (not (equal k 1)) )
           (< 1 k) ) )

(defthmd swap-isNeuron-sub2.1-2.1
  (IMPLIES (AND (ISNEURON (AFTER K N))
                (<= 1 (+ (- K) 1 1 (LEN (CDDR N))))
                (<= 1 (+ 1 (LEN (CDR (AFTER K N)))))
                (RATIONALP (CAR (AFTER K N)))
                (ISLISTOFWEIGHTS (CDR (AFTER K N)))
                (ISNEURON (LIST* (CADR N) (CAR N) (CDDR N)))
                (ISNEURON (CONS (CAR N) (CDR (AFTER K N))))
                (<= 1 (LEN (CDR N)))
                (<= 1 (+ 1 (LEN (CDDR N))))
                (ISNEURON (AFTER (+ 1 (LEN (CDDR N))) N))
                (NOT (EQUAL K 1))
                (TRUE-LISTP (CDDR N))
                (RATIONALP (CAR N))
                (RATIONALP (CADR N))
                (ISLISTOFWEIGHTS (CDDR N))
                (< 0 K)
                (ISNEURON N)
                (< 0 (+ 1 1 (LEN (CDDR N))))
                (INTEGERP K)
                (<= 1 (LEN N))
                (<= 1 (+ 1 1 (LEN (CDDR N))))
                (< K (+ 1 1 (LEN (CDDR N))))
                (<= 0 K))
           (ISNEURON (CONS (CAR (AFTER K N))
                           (APPEND (CDR (FRONT K N))
                                   (CONS (CAR N) (CDR (AFTER K N)))))))

:hints ("Goal"
       :use ((:instance cdr-isNeuron (n (front k n)) )
             (:instance front-len (i k) (w n) )
             k-<-duh
             (:instance front-isNeuron (j k) (n n) ) )
       ("Subgoal 1"
       :use ((:instance add-synapse-to-neuron2 (s (car (after k n)))
                                                (n (APPEND (CDR (FRONT K N))
                                                            (CONS
                                                                (CAR N)
                                                                (CDR (AFTER K N)))))) )
             (:instance app-isNeuron (a (CDR (FRONT K N)))
                                     (b (CONS (CAR N) (CDR (AFTER K N)))) ) ) )

(defthmd swap-isNeuron-sub2.1-1.2
  (IMPLIES
   (AND (EQUAL (+ (- J) K) 1)
        (ISNEURON (FRONT J N))
        (EQUAL (+ (- J) J K) K)
        (TRUE-LISTP (CDR N))
        (RATIONALP (CAR N))
        (ISLISTOFWEIGHTS (CDR N))

```

```

(ISNEURON (AFTER K N))
(NOT (EQUAL J 0))
(< J K)
(ISNEURON N)
(INTEGERP J)
(<= 0 J)
(< J (+ 1 (LEN (CDR N))))
(INTEGERP K)
(<= 0 K)
(<= 1 (LEN N))
(<= 1 (+ 1 (LEN (CDR N))))
(< K (+ 1 (LEN (CDR N))))
(<= J K)
(NOT (EQUAL J K)))
(ISNEURON (APPEND (FRONT J N)
  (CONS (CAR (AFTER 1 (AFTER J N)))
    (APPEND (CDR (FRONT 1 (AFTER J N)))
      (CONS (CAR (AFTER J N))
        (CDR (AFTER 1 (AFTER J N))))))))))

:hints ("Goal"
  :do-not-induct t
  :in-theory (enable after-len
    car-front
    isNeuron-duh
    len-1)
  :hands-off (isNeuron)
  :use ((:instance isNeuron-duh (n (after j n)) )
    (:instance isNeuron-duh (n (after k n)) )
    (:instance after-len (i j) (w n) )
    (:instance front-isNeuron (j (+ (- J) K)) (n (after j n)) )
    (:instance add-synapse-to-neuron (s (CAR (AFTER J N)) )
      (n (CDR (AFTER K N)) ) )
    (:instance after-isNeuron (k j) (n n) )
    (:instance after-after (a (+ (- j) k)) (b j) (x n) )) )
  "Subgoal 1"
  :use ((:instance app-isNeuron (a (FRONT J N) )
    (b (LIST* (CAR (AFTER K N))
      (CAR (AFTER J N))
      (CDR (AFTER K N)) ) ) )
    (:instance list*-isNeuron (a (CAR (AFTER K N))
      (b (CAR (AFTER J N))
      (c (CDR (AFTER K N)) ) ) ) ) )

(defthmd swap-isNeuron-sub2.1-1.1
  (IMPLIES
    (AND (NOT (EQUAL (+ (- J) K) 1))
      (ISNEURON (FRONT J N))
      (EQUAL (+ (- J) J K) K)
      (TRUE-LISTP (CDR N))
      (RATIONALP (CAR N))
      (ISLISTOFWEIGHTS (CDR N))
      (ISNEURON (AFTER K N))
      (NOT (EQUAL J 0))
      (< J K)
      (ISNEURON N)
      (INTEGERP J)
      (<= 0 J)
      (< J (+ 1 (LEN (CDR N))))
      (INTEGERP K)
      (<= 0 K)
      (<= 1 (LEN N))
      (<= 1 (+ 1 (LEN (CDR N))))
      (< K (+ 1 (LEN (CDR N))))
      (<= J K)
      (NOT (EQUAL J K)))
    (ISNEURON
      (APPEND (FRONT J N)
        (CONS (CAR (AFTER (+ (- J) K) (AFTER J N)))
          (APPEND (CDR (FRONT (+ (- J) K) (AFTER J N)))
            (CONS (CAR (AFTER J N))
              (CDR (AFTER 1 (AFTER J N))))))))))

```

```

(CDR (AFTER (+ (- J) K) (AFTER J N)))))))))

:hints ("Goal"
:do-not-induct t
:use ( (:instance isNeuron-duh (n (after j n)) )
      (:instance isNeuron-duh (n (after k n)) )
      (:instance after-len (i j) (w n) )
      (:instance cdr-isNeuron (n (FRONT (+ (- J) K) (AFTER J N))) )
      (:instance add-synapse-to-neuron (s (CAR (AFTER J N)) )
                                       (n (CDR (AFTER K N)) ) )
      (:instance add-synapse-to-neuron2 (s (CAR (AFTER K N)) )
                                       (n (APPEND (CDR (FRONT (+ (- J) K)
                                                         (AFTER J N)))
                                                (CONS (CAR (AFTER J N))
                                                    (CDR (AFTER K
                                                         N)))))) )

      (:instance front-len (i (+ (- J) K)) (w (AFTER J N)) )
      (:instance after-isNeuron (k j) (n n) )
      (:instance app-isNeuron (a (front j n) )
                              (b (CONS (CAR (AFTER K N))
                                       (APPEND (CDR (FRONT (+ (- J) K)
                                                         (AFTER J N)))
                                                (CONS (CAR (AFTER J N))
                                                    (CDR (AFTER K N)))))) ) )

      (:instance app-isNeuron (a (CDR (FRONT (+ (- J) K) (AFTER J N))) )
                              (b (CONS (CAR (AFTER J N))
                                       (CDR (AFTER K N)))) ) )

      (:instance front-isNeuron (j (+ (- J) K)) (n (after j n)) )
      (:instance after-after (a (+ (- j) k)) (b j) (x n)) )
:in-theory (enable after-len
                  car-front
                  isNeuron-duh
                  len-0
                  len-1)
:hands-off (isNeuron) )) )

(defthmd swap-isNeuron-sub2.1
  (IMPLIES (AND (< J K)
                (ISNEURON N)
                (INTEGERP J)
                (<= 0 J)
                (< J (LEN N))
                (INTEGERP K)
                (<= 0 K)
                (< K (LEN N))
                (<= J K)
                (NOT (EQUAL J K)))
           (ISNEURON (APPEND (FRONT J N)
                             (CONS (CAR (AFTER (+ (- J) K) (AFTER J N)))
                                   (APPEND (CDR (FRONT (+ (- J) K) (AFTER J N)))
                                           (CONS (CAR (FRONT (+ (- J) K) (AFTER J N)))
                                               (CDR (AFTER (+ (- J) K)
                                                         (AFTER J N)))))))))))

:hints ("Goal"
:cases ( (equal j 0)
        (not (equal j 0)) )
:in-theory (enable after-len
                  car-front
                  isNeuron-duh
                  len-1)
:hands-off (isNeuron)
:do-not-induct t
:use (front-isNeuron
      jk-trans-duh
      zero-j-duh
      isNeuron-true-listp
      isNeuron-duh
      after-isNeuron) )
("Subgoal 2"
:cases ( (equal k 1)

```



```

(AFTER J X)))
(CONS (CAR (FRONT (+ (- J) K)
(AFTER J X)))
(CDR (AFTER
(+ (- J) K)
(AFTER J X)))))) )
(:instance len-append (a (CDR (FRONT (+ (- J) K) (AFTER J X))) )
(b (CONS (CAR (FRONT (+ (- J) K) (AFTER J X)))
(CDR (AFTER (+ (- J) K) (AFTER J X))))))
zero-j-duh
(:instance len-step (a (car (after k x)))
(b (APPEND (CDR (FRONT (+ (- J) K) (AFTER J X)))
(CONS (CAR (FRONT (+ (- J) K)
(AFTER J X)))
(CDR (AFTER K X)))) ) )
(:instance len-step (a (CAR (FRONT (+ (- J) K) (AFTER J X))) )
(b (CDR (AFTER K X)) ) )
(:instance len-cdr (w (FRONT (+ (- J) K) (AFTER J X))) )
(:instance len-cdr (w (after k x)) )
(:instance after-len (i j) (w x) )
(:instance after-len (i k) (w x) )
(:instance after-after (a (+ (- j) k)) (b j) (x x) )
(:instance front-len (i (+ (- J) K)) (w (AFTER J X)) )
(:instance front-len (i j) (w x) )
:do-not-induct t)) )

(defthmd len-swap
  (implies (and (true-listp x)
    (natp j)
    (natp k)
    (< j (len x))
    (< k (len x)) )
    (equal (len x)
      (len (swap x j k)) ) )

  :hints (("Goal"
    :do-not-induct t
    :in-theory (enable front-len
      after-len)
    :use (len-swap-2.1
      (:instance len-swap-2.1 (j k) (k j) (x x) ) )
    :cases ( (equal j k)
      (< j k)
      (< k j) ) ) ) )

(defthmd same-neuron-len-after-swapNeuronSynapsesInLayer
  (implies (and (isLayer l)
    (natp j)
    (< j (len (car l)))
    (natp k)
    (< k (len (car l))) )
    (equal (len (car l))
      (len (car (swapNeuronSynapsesInLayer l j k))) ) )

  :hints (("Goal"
    :expand ( (swapNeuronSynapsesInLayer l j k) )
    :cases ( (equal j k)
      (not (equal j k)) ) )
    :use (isLayer-true-listp
      isLayer-isLayer2-duh
      (:instance isNeuron-true-listp (n (car l)) )
      (:instance len-swap (j j) (k k) (x (car l)) ) )
    :hands-off (swap
      isNeuron
      isLayer)
    :do-not-induct t )
  ("Subgoal 1"
    :use ((:instance swap-same (i j) (x (car l)) )
      (:instance swap-same (i k) (x (car l)) ) ) ) )

(defthmd swapNeuronSynapsesInLayer-isLayer

```

```

(implies (and (isLayer l)
              (natp j)
              (< j (len (car l)))
              (natp k)
              (< k (len (car l))) )
         (isLayer (swapNeuronSynapsesInLayer l j k)) )

:hints ("Goal"
       :induct (swapNeuronSynapsesInLayer l j k) )
("Subgoal *1/2"
 :do-not-induct t)
("Subgoal *1/1"
 :do-not '(generalize)
 :in-theory (enable len-1
                  len-0)
 :cases ( (equal (len l) 1)
          (<= 2 (len l)) )
 :hands-off (isLayer
             isNeuron
             swap)
 :use ((:instance swap-isNeuron (n (car l)) )
       isLayer-<-2
       (:instance isNeuron-true-listp (n (car l)) )
       (:instance isLayer-all-neurons-same (x l) )
       (:instance add-neurons-to-layer (n (SWAP (CAR L) J K))
                                         (l (SWAPNEURONSYNAPSESINLAYER
                                             (CDR L) J K)) )
       (:instance same-neuron-len-after-swapNeuronSynapsesInLayer
                   (l (cdr l)) )
       (:instance isLayer-isLayer2-duh (l (LIST (SWAP (CAR L) J K)) ) )
       (:instance len-swap (j j) (k k) (x (car l)) )
       isLayer-isLayer2-duh)
 :do-not-induct t)) )

(defthmd nth-in-network-isLayer
  (implies (and (isNetwork n)
                (< 1 (len n))
                (natp l) )
           (isLayer (nth l n)) ) )

(defthmd len-cdr-helper
  (implies (< 2 (+ 1 (LEN (CDR N))))
           (< 1 (LEN (CDR N)))) )

(defthmd len-cdr-helper2
  (implies (< L (- (len n) 1))
           (< (+ -1 L) (+ -1 (LEN (CDR N)))))) )

(defthmd argh!
  (equal (+ -1 1 L) (+ 1 -1 L)) )

(defthmd nth-helper
  (implies (and (consp n)
                (< 1 (- (len n) 1))
                (natp L) )
           (equal (NTH (+ 1 -1 L) (CDR N))
                  (NTH (+ 1 L) N)) ) )

:hints ("Goal"
       :use (argh!)
       :expand ( (NTH (+ 1 L) N) ) ) )

(defthmd nth-layer-in-network-consistent
  (implies (and (isNetwork n)
                (< 1 (len n))
                (natp l)
                (< 1 (- (len n) 1)) )
           (equal (len (nth l n))
                  (len (car (nth (+ 1 L) n)))) ) )

:hints ("Goal"

```

```

:hands-off (isNetwork
            isLayer)
:induct (nth 1 n) )
("Subgoal *1/3"
:cases ( (equal 2 (len n))
         (< 2 (len n)) )
:use (isNetwork-isNetwork2-duh
      len-cdr-helper
      nth-helper
      len-cdr-helper2
      (:instance len-cdr (w n) )
      isNetwork-cdr)
:do-not-induct t)
("Subgoal *1/2"
:use (isNetwork-isNetwork2-duh)
:do-not-induct t)
("Subgoal *1/1"
:use (isNetwork-isNetwork2-duh)
:do-not-induct t)) )

(defthmd cdr-len-help
  (implies (and (NOT (EQUAL (LEN N) 1))
               (NOT (ZP L))
               (natp 1)
               (< 1 (len n)) )
           (and (< (+ -1 L) (LEN (CDR N)))
                (< 0 (+ -1 (LEN N)))
                (< 0 (+ -1 1 (LEN (CDR N)))) ) ) ) )

(defun fn-helper (f i n l)
  (if (consp n)
      (fn-helper f (forwardLayer f i (car n)) (cdr n) (- 1 l))
      (+ 1 i)))

(defthmd forwardNetwork-over-front-after
  (implies (and (isNetwork n)
               (isListOfWeights i)
               (equal (len (caar n)) (len i))
               (natp 1)
               (< 1 (len n)) )
           (equal (forwardNetwork f i n)
                  (forwardNetwork f (forwardNetwork f i (front 1 n)) (after 1 n)) ) ) )

:hints ("Goal"
:hands-off (isNetwork
            isListOfWeights
            forwardLayer
            isLayer)
:induct (fn-helper f i n l) )
("Subgoal *1/2"
:do-not-induct t)
("Subgoal *1/1"
:expand ( (FORWARDNETWORK F I
           (CONS (CAR N) (FRONT (+ -1 L) (CDR N)))) )
:cases ( (ISLAYER (CDAR N))
         (NOT (ISLAYER (CDAR N))) )
:use (isNetwork-isNetwork2-duh
      (:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) )
      (:instance isLayer-isLayer2-duh (l (car n)) )
      (:instance isLayer-all-neurons-same (x (car n)) )
      (:instance forwardLayer-len (f f) (i i) (l (car n)) )
      cdr-len-help
      isNetwork-cdr)
:do-not-induct t)) )

(defthmd L-len-n-duh
  (implies (and (natp 1)
               (< L (+ -1 (LEN N))) )
           (<= (+ 2 L) (LEN N)) ) )

(defthmd one-layer-isNetwork

```

```

    (implies (isLayer l)
             (isNetwork (list l)) ) )

(defthmd come-on!
  (equal (+ 1 -1 L) (+ -1 1 L)) )

(defthmd come-on!2
  (equal (+ 1 2 -1 L) (+ 2 L)) )

(defthmd come-on!3
  (implies (natp L)
           (equal (+ 1 -1 L) L)) )

(defthmd first-two-of-after
  (implies (and (<= 2 (len n))
               (<= (+ 2 L) (len n))
               (natp L)
               (true-listp n))
           (equal (after L n)
                  (list* (nth L n)
                         (nth (+ 1 L) n)
                         (after (+ 2 1) n) ) ) )

:hints (("Goal"
        :induct (after 1 n) )
        ("Subgoal *1/3"
         :do-not-induct t)
        ("Subgoal *1/2"
         :use (come-on!
              come-on!2
              (:instance after-cdr (n n) (i (+ 2 -1 L)) )
              (:instance nth-1-cdr (n n) (l (+ 1 1)) ))
         :do-not-induct t)
        ("Subgoal *1/1"
         :expand ( (NTH 1 N)
                   (AFTER 2 N)
                   (AFTER 1 (CDR N)) )
         :do-not-induct t)) )

(defthmd forwardNetwork-over-two-swapped
  (implies (and (isListOfWeights i)
                (equal (len i) (len (car (nth L n))))
                (natp k)
                (< k (len (nth 1 n)))
                (natp j)
                (< j (len (nth 1 n)))
                (natp l)
                (<= 2 (len n))
                (<= (+ 2 L) (len n))
                (isNetwork n) )
           (equal (forwardNetwork f i (list* (nth L n)
                                             (nth (+ 1 L) n)
                                             (after (+ 2 1) n) ))
                  (forwardNetwork f i (list* (SWAP (NTH L N) J K)
                                             (SWAPNEURONSYNAPSESINLAYER
                                              (NTH (+ 1 L) N) J K)
                                             (after (+ 2 1) n) )) ) )

:hints (("Goal"
        :use (nth-in-network-isLayer
              first-two-of-after
              isNetwork-true-listp
              after-isNetwork
              (:instance isNetwork-isNetwork2-duh (n (after 1 n)) )
              (:instance nth-in-network-isLayer (l (+ 1 L)) (n n) )
              (:instance forwardLayer-len (f f) (i i) (l (nth 1 n)) )
              (:instance isListOfWeights-forwardLayer (f f) (i i) (l (nth 1 n)) )
              (:instance synapse-swap-in-neurons-of-layer (f f)
                   (i (FORWARDLAYER
                       F I (nth L n)))
                   (j j)

```

```

(k k)
(l (NTH (+ 1 L) N)) )
(:instance swap-over-forwardLayer (f f) (i i) (j j) (k k)
(l (nth 1 n)) )
:hands-off (forwardLayer
isLayer
swap)
:do-not-induct t)) )

(defthmd isListOfWeights-forwardNetwork
(implies (and (isNetwork n)
(isListOfWeights i)
(equal (len (caar n)) (len i)) )
(isListOfWeights (forwardNetwork f i n)) )

:hints ("Goal"
:in-theory (enable isLayer-true-listp
forwardLayer-len
isListOfWeights-forwardLayer
isNetwork-true-listp)
:induct (forwardNetwork f i n) )
("Subgoal *1/3"
:do-not-induct t)
("Subgoal *1/2"
:do-not-induct t)
("Subgoal *1/1"
:hands-off (forwardLayer
isLayer)
:use ((:instance forwardLayer-len (f f) (i i) (l (car n)) )
(:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) ))
:do-not-induct t)) )

(defthmd len-caar-swapLayerNeuronsInNetwork
(implies (and (isNetwork n)
(natp l)
(natp j)
(natp k)
(< l (len n))
(< j (len (nth 1 n)))
(< k (len (nth 1 n))) )
(equal (len (caar n))
(len (caar (swapLayerNeuronsInNetwork n l j k))) ) )

:hints ("Goal"
:hands-off (isNetwork
swap
isLayer) )
("Subgoal *1/1"
:use ((:instance same-len-neuron-in-swap (l (car n)) (j j) (k k) )
isNetwork-isNetwork2-duh)
:do-not-induct t)) )

(defthmd len-swapNeuronSynapsesInLayer
(implies (and (isLayer l)
(natp j)
(natp k) )
(equal (len (swapNeuronSynapsesInLayer l j k))
(len l) ) )

:hints ("Goal"
:induct (len l) )
("Subgoal *1/1"
:do-not-induct t)) )

(defthmd isNetwork-swapLayerNeuronsInNetwork
(implies (and (isNetwork n)
(natp l)
(natp j)
(natp k)
(< l (len n))
(< j (len (nth 1 n)))

```

```

      (< k (len (nth 1 n))) )
      (isNetwork (swapLayerNeuronsInNetwork n 1 j k)) )

:hints ("Goal"
       :do-not '(generalize)
       :hands-off (isLayer)
       :induct (swapLayerNeuronsInNetwork n 1 j k) )
("Subgoal *1/4"
 :do-not-induct t)
("Subgoal *1/3"
 :do-not-induct t)
("Subgoal *1/2"
 :use ((:instance len-caar-swapLayerNeuronsInNetwork (n (cdr n))
                (1 (+ -1 1)) (j j) (k k))
       len-caar-swapLayerNeuronsInNetwork)
 :do-not-induct t)
("Subgoal *1/1"
 :expand ( (SWAPLAYERNEURONSINNETWORK N 0 J K) )
 :use (isNetwork-isNetwork2-duh
      (:instance isLayer-true-listp (1 (car n)) )
      (:instance isNetwork-isNetwork2-duh (n (cdr n)) )
      (:instance swapNeuronSynapsesInLayer-isLayer (1 (cadr n)) (j j) (k k) )
      (:instance swap-isLayer (1 (car n)) (j j) (k k) )
      (:instance swap-len (1 (car n)) (j j) (k k) )
      (:instance len-swapNeuronSynapsesInLayer (1 (cadr n)) (j j) (k k) )
      (:instance same-neuron-len-after-swapNeuronSynapsesInLayer
                (1 (cadr n)) (j j) (k k) )
      (:instance isLayer-isLayer2-duh (1 (car n)) ))
 :hands-off (swap
             isLayer
             isNeuron)
 :do-not-induct t)) )

(defthmd not-cdr-isNetwork
  (implies (and (isNetwork n)
                (not (isNetwork (cdr n))) )
            (not (cdr n)) )

  :hints ("Goal"
         :use (isNetwork-true-listp) )) )

(defthmd isNetwork-cdr-duh
  (implies (isNetwork (cdr n))
            (cdr n)) )

(defthmd isNet-helper
  (implies (and (NOT (EQUAL 1 (LEN N)))
                (isNetwork n) )
            (and (isNetwork (cdr n))
                 (< 1 (len n)) ) ) )

(defthmd grrr!
  (equal (+ -1 -1 1 (LEN (CDR N))) (+ -1 (LEN (CDR N))) ) )

(defthmd len-forwardNetwork
  (implies (and (isNetwork n)
                (isListOfWeights i)
                (equal (len (caar n)) (len i)) )
            (equal (len (forwardNetwork f i n))
                   (len (nth (- (len n) 1) n)) ) )

  :hints ("Goal"
         :induct (forwardNetwork f i n) )
("Subgoal *1/2"
 :do-not-induct t)
("Subgoal *1/1"
 :in-theory (enable len-0
                    len-1)
 :cases ( (equal 1 (len n))
           (not (equal 1 (len n))) )
 :hands-off (isNetwork

```

```

        isLayer)
:use (isNetwork-cdr
      (:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) )
      not-cdr-isNetwork
      isNetwork-cdr-duh
      isNetwork-true-listp
      isNetwork-isNetwork2-duh)
:do-not-induct t)
("Subgoal *1/1.2.1'")
:use ((:instance forwardLayer-len (f f) (i i) (l (car n)) )) )
("Subgoal *1/1.1")
:use ((:instance forwardLayer-len (f f) (i i) (l (car n)) )
      grrr!
      (:instance nth-1-cdr (n n) (l (+ -1 1 (LEN (CDR N)))) )
      isNet-helper) )) )

(defthmd nth-front
  (implies (and (< a b)
                (<= b (len n))
                (natp a)
                (natp b) )
           (equal (nth a (front b n))
                  (nth a n) ) ) )

(defthmd nth-car-after
  (implies (and (< L (len n))
                (natp L) )
           (equal (car (after L n))
                  (nth L n) ) ) )

(defthmd grrr!2
  (equal (+ 1 1 L) (+ 2 L)) )

(defthmd obvious-helper-len-cdr
  (equal (+ -1 -1 1 (LEN (CDR N)))
         (+ -1 (LEN (CDR N))) ) )

(defthmd front-after-swapLayerNeuronsInNetwork-last-layer
  (implies (and (isNetwork n)
                (natp k)
                (< k (len (nth (- (len n) 1) n)))
                (natp j)
                (< j (len (nth (- (len n) 1) n))) )
                (equal (swapLayerNeuronsInNetwork n (- (len n) 1) j k)
                       (append (front (- (len n) 1) n)
                                (cons (swap (nth (- (len n) 1) n) j k)
                                       nil)) ) )
                )
           :hints (("Goal"
                    :induct (len n)
                    :hands-off (swap
                                swapNeuronSynapsesInLayer
                                isNetwork
                                isLayer) )
                    ("Subgoal *1/2"
                     :do-not-induct t)
                    ("Subgoal *1/1"
                     :cases ( (isNetwork (cdr n)) )
                     :use (isNetwork-cdr
                           (:instance cdr-nth (l (len (cdr n))) (a n) )
                           (:instance isNetwork-isNetwork2-duh (n (cdr n)) )
                           isNetwork-isNetwork2-duh)
                           :do-not-induct t)
                    ("Subgoal *1/1.1"
                     :use (obvious-helper-len-cdr)
                     :expand ( (NTH (+ -1 1 (LEN (CDR N))) N) ) ) ) )

(defthmd consp-len-natp-duh
  (implies (consp n)
           (NATP (+ -1 (LEN N))) ) )

```

```

(defthmd obvious-helper-len-cdr2
  (equal (+ 1 -1 -1 (LEN (CDR N)))
    (+ -1 (LEN (CDR N))) ) )

(defthmd consp-after-all-but-last
  (implies (consp n)
    (consp (AFTER (+ -1 (LEN N)) N) ) )

  :hints (("Goal"
    :in-theory (enable grrr!)
    :expand ( (AFTER (+ -1 1 (LEN (CDR N))) N) )
    :induct (LEN N) ) ) )

(defthmd one-step-forwardNetwork
  (implies (consp n)
    (equal (forwardNetwork f i n)
      (forwardNetwork f (forwardLayer f i (car n)) (cdr n)) ) ) )

(defthmd nothing-after-last-cdr
  (implies (and (true-listp n)
    (consp n) )
    (not (cdr (AFTER (+ -1 (LEN N)) N))) )

  :hints (("Goal"
    :in-theory (enable grrr!)
    :induct (LEN N) ) ) )

;-----
; 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.

(defthmd neuron-swap-in-input-or-hidden-layer-of-network
  (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)) ) )

  :hints (("Goal"
    :cases ( (equal l 0)
      (not (equal l 0)) )
    :hands-off ( swap
      forwardNetwork
      isNetwork
      isLayer
      forwardLayer )
    :do-not-induct t )
    ("Subgoal 2"
    :use (neuron-swap-in-first-layer-of-network) )
    ("Subgoal 1"
    :in-theory (enable car-front)
    :cases ( (equal (+ 2 L) (len n))
      (not (equal (+ 2 L) (len n))) )
    :use (L-len-n-duh
      first-two-of-after
      nth-layer-in-network-consistent
      front-after-swapLayerNeuronsInNetwork
      forwardNetwork-over-front-after
      front-isNetwork
      isNetwork-true-listp
      (:instance nth-in-network-isLayer (l (+ 1 L)) (n n) )
      after-all-nil

```

```

      (:instance swapNeuronSynapsesInLayer-isLayer
        (1 (NTH (+ 1 L) N)) (j j) (k k) )
      (:instance after-isNetwork (1 (+ 2 l)) (n n) )) )
("Subgoal 1.2.1"
:use ((:instance nth-front (a (+ -1 L)) (b L) (n n) )
      (:instance nth-layer-in-network-consistent (1 (- 1 l)) (n n) )
      come-on!3
      isNetwork-swapLayerNeuronsInNetwork
      (:instance forwardNetwork-over-two-swapped
        (i (FORWARDNETWORK F I (FRONT L N))) (j j) (k k) (l l) )
      (:instance isListOfWeights-forwardNetwork (f f) (i i) (n (front l n)) )
      (:instance isNetwork-app (a (front l n))
        (b (LIST (SWAP (NTH L N) J K)
                  (SWAPNEURONSYNAPSESINLAYER
                    (NTH (+ 1 L) N) J K)) ) )
      (:instance one-layer-isNetwork (1 (SWAPNEURONSYNAPSESINLAYER
        (NTH (+ 1 L) N) J K)) )
      (:instance isLayer-true-listp (1 (nth l n)) )
      (:instance front-len (i l) (w n) )
      (:instance swap-len (1 (nth l n)) (j j) (k k) )
      (:instance nth-in-network-isLayer (1 l) (n n) )
      (:instance len-forwardNetwork (f f) (i i) (n (front l n)) )
      (:instance same-neuron-len-after-swapNeuronSynapsesInLayer
        (1 (NTH (+ 1 L) N)) (j j) (k k) )
      (:instance swap-isLayer (1 (NTH L N)) (j j) (k k) )
      (:instance swapNeuronSynapsesInLayer-isLayer
        (1 (NTH (+ 1 L) N)) (j j) (k k) )
      (:instance add-layer-to-network (1 (SWAP (NTH L N) J K))
        (n (cons (SWAPNEURONSYNAPSESINLAYER
                  (NTH (+ 1 L) N) J K)
                  nil) ) )
      (:instance forwardNetwork-over-app (f f)
        (i i)
        (a (front l n))
        (b (LIST (SWAP (NTH L N) J K)
                  (SWAPNEURONSYNAPSESINLAYER
                    (NTH (+ 1 L) N)
                    J K)) ) ) ) )
("Subgoal 1.1.1"
:use ((:instance add-layer-to-network (1 (SWAPNEURONSYNAPSESINLAYER
        (NTH (+ 1 L) N) J K) )
      (n (AFTER (+ 2 L) N) ) )
      (:instance add-layer-to-network (1 (SWAP (NTH L N) J K) )
      (n (cons (SWAPNEURONSYNAPSESINLAYER
        (NTH (+ 1 L) N) J K)
        (AFTER (+ 2 L) N)) ) ) )
grrr!2
(:instance nth-front (a (+ -1 L)) (b L) (n n) )
(:instance nth-layer-in-network-consistent (1 (- 1 l)) (n n) )
come-on!3
(:instance front-len (i l) (w n) )
(:instance same-neuron-len-after-swapNeuronSynapsesInLayer
  (1 (NTH (+ 1 L) N)) (j j) (k k) )
nth-in-network-isLayer
(:instance len-forwardNetwork (f f) (i i) (n (front l n)) )
(:instance isListOfWeights-forwardNetwork (f f) (i i) (n (front l n)) )
(:instance forwardNetwork-over-two-swapped
  (i (FORWARDNETWORK F I (FRONT L N))) (j j) (k k) (l l) )
isNetwork-swapLayerNeuronsInNetwork
(:instance isLayer-true-listp (1 (NTH L N)) )
(:instance swap-len (1 (nth l n)) (j j) (k k) )
(:instance swap-isLayer (1 (nth L n)) (j j) (k k) )
(:instance nth-car-after (1 (+ 2 l)) (n n) )
(:instance nth-layer-in-network-consistent (1 (+ 1 L)) (n n) )
(:instance len-swapNeuronSynapsesInLayer
  (1 (NTH (+ 1 L) N)) (j j) (k k) )
(:instance forwardNetwork-over-app (f f)
  (i i)
  (a (front l n))
  (b (LIST* (SWAP (NTH L N) J K)
            (SWAPNEURONSYNAPSESINLAYER
              (NTH (+ 1 L) N)
              J K)) ) ) )

```

```
(NTH (+ 1 L) N) J K)
(AFTER (+ 2 L) N)))))) ) )
```

```
; 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.
```

```
(defthmd neuron-swap-in-output-layer-of-network
  (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) ) )

  :hints (("Goal"
          :cases ( (equal (len n) 1)
                  (not (equal (len n) 1)) )
          :expand ( (FORWARDNETWORK F I NIL)
                    (NTH (+ -1 (LEN (CDR N)))
                          (CONS (CAR N) (FRONT (+ -1 (LEN (CDR N))) (CDR N)))) )
          :use (front-after-swapLayerNeuronsInNetwork-last-layer
                (:instance neuron-swap-in-layer (l (nth (+ -1 (LEN N)) n)) (j j)
                                                  (k k) (i i) (f f) )
                (:instance neuron-swap-in-layer (l (nth (+ -1 (LEN N)) n))
                                                  (j j)
                                                  (k k)
                                                  (i (FORWARDNETWORK
                                                       F I (FRONT (+ -1 (LEN N)) N)))
                                                  (f f) )
                (:instance car-front (i (+ -1 (LEN N))) (x n) )
                (:instance nth-in-network-isLayer (l (+ -1 (LEN N))) (n n) )
                (:instance front-isNetwork (l (+ -1 (LEN N))) (n n) )
                (:instance swap-isLayer (l (NTH (+ -1 (LEN N)) n) ) (j j) (k k) )
                (:instance one-layer-isNetwork (l (SWAP (NTH (+ -1 (LEN N)) N) J K) ) )
                (:instance isNetwork-swapLayerNeuronsInNetwork (n n) (l (+ -1 (LEN N)))
                                                                (j j) (k k) )
                (:instance forwardNetwork-over-app (f f)
                                                    (i i)
                                                    (a (front (- (len n) 1) n))
                                                    (b (LIST
                                                         (SWAP (NTH (+ -1 (LEN N)) N)
                                                             J K)) ) )
                (:instance forwardNetwork-over-front-after (f f) (i i) (n n)
                                                            (l (- (len n) 1)) ) )

          :hands-off ( swap
                      isNetwork
                      isLayer
                      forwardLayer )
          :do-not-induct t )
  ("Subgoal 1"
   :use (isNetwork-cdr
         isNetwork-true-listp
         obvious-helper-len-cdr2
         isNetwork-isNetwork2-duh
         obvious-helper-len-cdr
         (:instance front-len (i (+ -1 1 (LEN (CDR N)))) (w n) )
         (:instance len-forwardNetwork (f f) (i i)
                                         (n (FRONT (+ -1 1 (LEN (CDR N))) N) ) )
         consp-len-natp-duh )
  ("Subgoal 1.6"
   :use ((:instance nth-layer-in-network-consistent
                  (l (+ -1 -1 (LEN (CDR N)))) (n (cdr n)) )
         (:instance nth-front (a (+ -1 -1 (LEN (CDR N))))
                              (b (+ -1 (LEN (CDR N)))) (n (cdr n)) ) ) )
  ("Subgoal 1.5"
   :use ((:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) )
```

```

(:instance car-front (i (+ -1 (LEN (cdr N)))) (x (cdr n)) )
(:instance forwardLayer-len (f f) (i i) (l (car n)) )
(:instance front-isNetwork (l (+ -1 (LEN (cdr N)))) (n (cdr n)) )
(:instance isListOfWeights-forwardNetwork (f f)
      (i (FORWARDLAYER
          F I (CAR N)))
      (n (FRONT
          (+ -1 (LEN (CDR N)))
          (CDR N) ) ) ) )

("Subgoal 1.4"
 :use ((:instance consp-after-all-but-last (n (cdr n)) )
       (:instance nothing-after-last-cdr (n (cdr n)) )
       (:instance one-step-forwardNetwork (f f)
      (i (FORWARDNETWORK
          F (FORWARDLAYER F I (CAR N))
          (FRONT (+ -1 (LEN (CDR N)))
                  (CDR N))) )
      (n (AFTER (+ -1 (LEN (CDR N)))
                  (CDR N)) ) )
       (:instance nth-car-after (l (+ -1 (LEN (CDR N)))) (n (cdr n)) ) ) )

("Subgoal 1.3"
 :use ((:instance nth-front (a (+ -1 -1 (LEN (CDR N))))
      (b (+ -1 (LEN (CDR N)))) (n (cdr n)) )
       (:instance nth-layer-in-network-consistent (l (+ -1 -1 (LEN (CDR N))))
      (n (cdr n)) ) ) )

("Subgoal 1.2"
 :use ((:instance isListOfWeights-forwardLayer (f f) (i i) (l (car n)) )
       (:instance car-front (i (+ -1 (LEN (cdr N)))) (x (cdr n)) )
       (:instance forwardLayer-len (f f) (i i) (l (car n)) )
       (:instance front-isNetwork (l (+ -1 (LEN (cdr N)))) (n (cdr n)) )
       (:instance isListOfWeights-forwardNetwork
      (f f) (i (FORWARDLAYER F I (CAR N)))
      (n (FRONT (+ -1 (LEN (CDR N))) (CDR N)) ) ) ) )

("Subgoal 1.1"
 :use ((:instance consp-after-all-but-last (n (cdr n)) )
       (:instance nothing-after-last-cdr (n (cdr n)) )
       (:instance one-step-forwardNetwork (f f)
      (i (FORWARDNETWORK
          F (FORWARDLAYER F I (CAR N))
          (FRONT (+ -1 (LEN (CDR N)))
                  (CDR N))) )
      (n (AFTER (+ -1 (LEN (CDR N)))
                  (CDR N)) ) )
       (:instance nth-car-after (l (+ -1 (LEN (CDR N)))) (n (cdr n)) ) ) ) )

```

```

;-----

```

```

****

```

```

dead.lisp

```

```

****

```

```

; Dead - Functions and theorems about dead neurons
(in-package "ACL2")

```

```

(include-book "form4")

```

```

;-----

```

```

; A dead synapse has a weight of 0.

```

```

(defun isDeadSynapse (s) (equal s 0))

```

```

; isDeadNeuron tests if a neuron makes no contribution to
; activation values. In other words, it checks if all of the
; neuron's synaptic weights are zero. The logic behind this
; definition of dead neuron only makes sense if the activation
; function maps 0 to 0.

```

```

(defun isDeadNeuron2 (n)
  (if (consp n)
      (if (isDeadSynapse (car n))
          (isDeadNeuron2 (cdr n))

```

```

        nil)
    t))

(defun isDeadNeuron (n)
  (if (isNeuron n)
      (isDeadNeuron2 n)
      nil))

; makeDeadNeuron creates a dead neuron of length i.

(defun makeDeadNeuron (i)
  (if (zp i)
      nil
      (cons 0 (makeDeadNeuron (- i 1)))))

; Inserting a neuron in a layer is accomplished by adding it to
; the front of the layer's list of neurons. No information about
; the position of the neuron in the layer is necessary, since it
; was proven earlier that the organization of neurons in a layer
; is irrelevant as long as the inputs in the next layer match up.
; After the neuron is added to the given layer, each neuron in the
; next layer needs to have an additional synapse added. The
; addNeuronToLayerOfNetwork function takes a neuron n and inserts
; it into the layer at index l of network net, and then adds one synapse
; from the list s to each neuron of the following layer.

(defun addSynapsesToNeuronsOfLayer (l s)
  (if (consp l)
      (cons (cons (car s) (car l))
            (addSynapsesToNeuronsOfLayer (cdr l) (cdr s)))
      nil) )

(defun addSynapsesToFirstLayerOfNetwork (s net)
  (if (consp net)
      (cons (addSynapsesToNeuronsOfLayer (car net) s)
            (cdr net))
      nil) )

(defun addNeuronToLayerOfNetwork (n s l net)
  (if (zp l)
      (cons (cons n (car net))
            (addSynapsesToFirstLayerOfNetwork s (cdr net)))
      (cons (car net)
            (addNeuronToLayerOfNetwork n s (- l 1) (cdr net)))) )

;-----

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

(defun dead-n-helper (i n)
  (if (consp n)
      (dead-n-helper (cdr i) (cdr n))
      (cons i n)))

(defthmd dead-neuron-forwardNeuron
  (implies (and (isDeadNeuron n)
                (isListOfWeights i)
                (equal (len n) (len i)))
           (equal (forwardNeuron i n) 0) )

  :hints (("Goal"
           :induct (dead-n-helper i n) )
          ("Subgoal *1/2"
           :do-not-induct t)
          ("Subgoal *1/1"
           :do-not-induct t)) )

(defthmd dead-neuron-output
  (implies (and (isDeadNeuron n)
                (isListOfWeights i)

```

```

      (equal (len n) (len i)) )
    (equal (activate f i n) 0) )

:hints (("Goal"
  :in-theory (enable dead-neuron-forwardNeuron)
  :induct (dead-n-helper i n) )
  ("Subgoal *1/2"
  :do-not-induct t)
  ("Subgoal *1/1"
  :do-not-induct t)) )

(defthmd makeDeadNeuron-isDeadNeuron
  (implies (not (zp i))
    (isDeadNeuron (makeDeadNeuron i)) ) )

(defthmd layer-len-helper
  (implies (and (NOT (EQUAL (LEN L) 1))
    (isLayer l) )
    (<= 2 (len l)) ) )

(defthmd car-addSynapsesToNeuronsOfLayer
  (implies (consp l)
    (equal (car (addSynapsesToNeuronsOfLayer l s))
      (cons (car s) (car l)) ) ) )

(defthmd len-sl-helper
  (equal (len (CONS (CADR S) (CADR L)))
    (+ 1 (len (CADR L)) ) ) )

(defthmd len-cdr2
  (implies (consp l)
    (equal (+ 1 (LEN (CDR L)))
      (LEN L) ) ) )

(defthmd len-car-addSynapsesToNeuronsOfLayer
  (implies (consp l)
    (equal (LEN (CAR (ADDSYNAPSESTONEURONSOF LAYER L S)))
      (+ 1 (len (car l)) ) ) )

:hints (("Goal"
  :use (car-addSynapsesToNeuronsOfLayer) ) ) )

(defthmd one-neuron-isLayer
  (implies (isNeuron n)
    (isLayer (list n)) ) )

(defthmd isLayer-addSynapsesToNeuronsOfLayer
  (implies (and (isLayer l)
    (isListOfWeights s)
    (<= (len l) (len s)) )
    (isLayer (addSynapsesToNeuronsOfLayer l s)) )

:hints (("Goal"
  :hands-off (isLayer
    isNeuron)
  :induct (addSynapsesToNeuronsOfLayer l s) )
  ("Subgoal *1/2"
  :use (isLayer-isLayer2-duh)
  :do-not-induct t )
  ("Subgoal *1/1"
  :cases ( (equal (len l) 1)
    (equal (len l) 2) )
  :do-not '(generalize)
  :use (isLayer-<-2
    layer-len-helper
    len-sl-helper
    (:instance len-cdr2 (l (car l)) )
    (:instance isLayer-all-neurons-same (x l) )
    (:instance car-addSynapsesToNeuronsOfLayer (s (cdr s)) (l (cdr l)) )
    (:instance isNeuron-duh (n (car l)) )
    (:instance one-neuron-isLayer (n (CONS (CAR S) (CAR L)) ) ) )

```

```

      (:instance len-car-addSynapsesToNeuronsOfLayer
        (l (cdr l)) (s (cdr s)) )
      (:instance add-neurons-to-layer (l (ADDSYNAPSESTONEURONSOF LAYER
        (CDR L) (CDR S)))
        (n (CONS (CAR S) (CAR L)) ) )
      (:instance add-synapse-to-neuron (s (car s)) (n (car l)) )
      (:instance isLayer-isLayer2-duh (l (cdr l)) )
      isLayer-isLayer2-duh)
    :do-not-induct t ) )

(defthmd add-layer-to-network2
  (implies (and (isLayer l)
    (isNetwork2 n (len l)) )
    (isNetwork (cons l n)) ) )

(defthmd len-addSynapsesToNeuronsOfLayer
  (equal (LEN (ADDSYNAPSESTONEURONSOF LAYER L S))
    (len l) ) )

(defthmd car-addNeuronToLayerOfNetwork
  (implies (and (consp n)
    (natp l) )
    (equal (car (addNeuronToLayerOfNetwork n s l net))
      (if (equal l 0)
        (cons n (car net))
        (car net)) ) ) )

(defthmd isNetwork-addNeuronToLayerOfNetwork
  (implies (and (isNeuron n)
    (isListOfWeights s)
    (isNetwork net)
    (equal (len n) (len (car (nth l net))))
    (<= (len (nth (+ l 1) net)) (len s))
    (natp l)
    (< l (len net)) )
    (isNetwork (addNeuronToLayerOfNetwork n s l net)) )

:hints ("Goal"
  :induct (addNeuronToLayerOfNetwork n s l net) )
  ("Subgoal *1/2"
  :use ((:instance car-addNeuronToLayerOfNetwork
    (n n) (s s) (l (+ -1 L)) (net (CDR NET)) ) )
  :do-not-induct t)
  ("Subgoal *1/1"
  :cases ( (equal (len net) 1)
    (equal (len net) 2) )
  :hands-off (isLayer
    isNetwork)
  :use ((:instance isNetwork-isNetwork2-duh (n net) )
    (:instance isNetwork-isNetwork2-duh (n (cdr net)) )
    (:instance isNetwork-cdr (n (cdr net)) )
    (:instance add-layer-to-network (l (CONS N (CAR NET)) )
      (n (CONS (ADDSYNAPSESTONEURONSOF LAYER
        (CADR NET) S)
        (CDDR NET)) ) )
    (:instance len-addSynapsesToNeuronsOfLayer (l (CADR NET)) (s s) )
    (:instance add-layer-to-network (l (ADDSYNAPSESTONEURONSOF LAYER
      (CADR NET) S))
      (n (caddr net)) )
    (:instance one-layer-isNetwork (l (CONS N (CAR NET)) ) )
    (:instance len-car-addSynapsesToNeuronsOfLayer (l (cadr net)) (s s) )
    (:instance isLayer-addSynapsesToNeuronsOfLayer (l (cadr net)) (s s) )
    (:instance add-neurons-to-layer (l (car net)) (n n) )
  :do-not-induct t)
  ("Subgoal *1/1.1"
  :use ((:instance add-layer-to-network2 (l (ADDSYNAPSESTONEURONSOF LAYER
    (CADR NET) S))
    (n (caddr net)) )
    (:instance one-layer-isNetwork (l (ADDSYNAPSESTONEURONSOF LAYER
      (CADR NET) S) ) ) )
  :do-not '(generalize) ) )

```

```

(defthmd simplify-helper-L
  (equal (+ 1 1 -1 L) (+ 1 L)) )

(defthmd addNeuronToLayerOfNetwork-app-front-after
  (implies (and (isNeuron n)
                (isListOfWeights s)
                (isNetwork net)
                (equal (len n) (len (car (nth 1 net))))
                (<= (len (nth (+ 1 1) net)) (len s))
                (natp 1)
                (< 1 (len net)) )
            (equal (addNeuronToLayerOfNetwork n s 1 net)
                  (append (front 1 net)
                          (cons (cons n (nth 1 net))
                                (addSynapsesToFirstLayerOfNetwork
                                 s (after (+ 1 1) net)))))) )

  :hints (("Goal"
          :hands-off (isNetwork
                     isLayer)
          :induct (addNeuronToLayerOfNetwork n s 1 net) )
         ("Subgoal *1/2"
          :use (simplify-helper-L
                (:instance after-cdr (n net) (i (+ 1 -1 L)) )
                (:instance isNetwork-cdr (n net) )
          :do-not-induct t)
         ("Subgoal *1/1"
          :do-not-induct t)) )

(defthmd makeDeadNeuron-len
  (implies (natp i)
            (equal (len (makeDeadNeuron i)) i) ) )

(defthmd isDeadNeuron-duh
  (iff (isDeadNeuron n)
       (and (isNeuron n)
            (isDeadNeuron2 n)) ) )

(defthmd mult-0
  (and (equal (* 0 y) 0)
        (equal (* x 0) 0)
        (equal (* a 0 b) 0) ) )

(defthmd zero-in-forwardLayer
  (implies (and (isLayer l)
                (isListOfWeights s)
                (equal (len s) (len l))
                (isListOfWeights i)
                (equal (len i) (len (car l)))) )
            (equal (forwardLayer f i l)
                  (forwardLayer f (cons 0 i)
                                (addSynapsesToNeuronsOfLayer l s)) ) )

  :hints (("Goal"
          :induct (addSynapsesToNeuronsOfLayer l s) )
         ("Subgoal *1/1"
          :in-theory (enable mult-0)
          :do-not '(generalize)
          :do-not-induct t)) )

(defthmd natp-iff
  (iff (natp x)
        (and (integerp x)
              (<= 0 x)) ) )

(defthmd strange-cdr-len-helper
  (implies (< L (+ -1 1 (LEN (CDR NET))))
            (< L (LEN NET)) ) )

(defthmd strange-ineq-to-eq-helper

```

```

(implies (EQUAL (LEN (nth (+ 1 1) NET)) (LEN S))
 (<= (LEN (nth (+ 1 1) NET)) (LEN S)) ) )

(defthmd stranger-helper-cdr-net
 (implies (and (isNetwork net)
 (natp 1)
 (< 1 (- (len net) 1)) )
 (< 0 (len (cdr net))) ) )

(defthmd isLayer-not-isLayer2-nil
 (implies (and (isLayer 1)
 (not (isLayer (cdr 1))) )
 (not (cdr 1)) ) )

(defthmd zero-in-forwardLayer2
 (implies (and (isLayer2 1 x)
 (natp x)
 (< 0 x)
 (isListOfWeights s)
 (equal (len s) (len 1))
 (isListOfWeights i)
 (equal (len i) (len (car 1)))) )
 (equal (forwardLayer f i 1)
 (forwardLayer f (cons 0 i)
 (addSynapsesToNeuronsOfLayer 1 s)) ) )

:hints ("Goal"
 :induct (addSynapsesToNeuronsOfLayer 1 s) )
("Subgoal *1/1"
 :in-theory (enable mult-0)
 :do-not '(generalize)
 :do-not-induct t)) )

(defthmd add-dead-neuron-to-first-layer
 (implies (and (isListOfWeights i)
 (< 0 (len i))
 (isNetwork net)
 (isListOfWeights s)
 (equal (len (nth (+ 1 1) net)) (len s))
 (equal (len i) (len (car (nth 1 net))))
 (< (+ 1 1) (len net))
 (natp 1) )
 (equal (forwardNetwork f i (LIST* (CONS (MAKEDEADNEURON
 (+ 1 (LEN (CDAR (NTH L NET))))))
 (NTH L NET))
 (ADDSYNAPSESTONEURONSOF LAYER
 (NTH (+ 1 L) NET) S)
 (CDR (AFTER (+ 1 L) NET))))
 (forwardNetwork f i (LIST* (NTH L NET)
 (NTH (+ 1 L) NET)
 (CDR (AFTER (+ 1 L) NET)))) ) )

:hints ("Goal"
 :induct (after 1 net)
 :hands-off (activate
 isNetwork
 isDeadNeuron
 isLayer
 isNeuron) )
("Subgoal *1/2"
 :use (come-on!
 (:instance isNetwork-cdr (n net) )
 (:instance nth-1-cdr (n net) (1 1) )
 (:instance nth-1-cdr (n net) (1 (+ 1 1)) ))
 :do-not-induct t)
("Subgoal *1/1"
 :cases ( (NOT (IS LAYER (CDR (nth 1 NET))))
 (IS LAYER (CDR (nth 1 NET))) )
 :use ((:instance isNetwork-cdr (n net) )
 (:instance isNeuron-duh (n (caar net)) )
 (:instance isNetwork-isNetwork2-duh (n net) )

```

```

      (:instance isLayer-not-isLayer2-nil (l (car net)) )
      (:instance isListOfWeights-forwardLayer (f f) (i i) (l (cadr net)) )
      (:instance isLayer-isLayer2-duh (l (car net)) )
      (:instance makeDeadNeuron-isDeadNeuron (i (+ 1 (LEN (CDAAR NET)))) ) )
      (:instance makeDeadNeuron-isDeadNeuron (i (LEN I) ) )
      (:instance makeDeadNeuron-len (i (LEN (CAR (NTH L NET)))) ) )
      (:instance makeDeadNeuron-len (i (LEN I) ) )
      (:instance zero-in-forwardLayer (f f)
                                       (l (CADR NET) )
                                       (i (list* (ACTIVATE F I (CAAR NET))
                                                (FORWARDLAYER
                                                  F I (CDAR NET)) ) )
                                       (s s) )
      (:instance isSynapse-activate (f f) (i i) (n (caar net)) )
      (:instance zero-in-forwardLayer (f f)
                                       (l (cadr NET) )
                                       (i (list (ACTIVATE F I (CAAR NET)) ) )
                                       (s s) )
      (:instance dead-neuron-output (f f) (i i) (n (MAKEDEADNEURON (LEN I))))
      (:instance forwardLayer-len (f f) (i i) (l (cadr net)) )
      :do-not-induct t)) )

(defthmd len-cdr-net-helper
  (implies (isNetwork (cdr net))
           (not (<= (+ 1 (LEN (CDR NET))) 1) ) ) )

(defthmd strange-nth-after-helper
  (implies (and (natp l)
                (< 1 (- (len net) 1))
                (equal (len (nth (+ 1 l) net)) (len s)) )
           (not (< (LEN S)
                   (LEN (CAR (AFTER (+ 1 L) NET)))) ) )

  :hints (("Goal"
           :do-not-induct t
           :use ((:instance nth-car-after (l (+ 1 l)) (n net) ) ) ) )

(defthmd very-obvious!
  (NATP (LEN (CAR (NTH L NET)))) )

(defthmd after-nth-helper
  (implies (and (natp l)
                (< 1 (- (len net) 1))
                (equal (len (nth (+ 1 l) net)) (len s)) )
           (equal (LEN (CAR (AFTER (+ 1 L) NET)))
                  (LEN S) ) )

  :hints (("Goal"
           :do-not-induct t
           :use ((:instance nth-car-after (l (+ 1 l)) (n net) ) ) ) )

(defthmd another-strange-helper
  (implies (isNetwork (cdr net))
           (not (< (+ 1 (LEN (CDR NET))) 2) ) ) )

(defthmd obvious-layer-consp-helper
  (implies (isLayer (NTH (+ 1 L) NET))
           (CONSP (NTH (+ 1 L) NET)) ) )

(defthmd obviously-2
  (implies (NOT (CDR (NTH L NET)))
           (EQUAL (+ 1 1 (LEN (CDR (NTH L NET)))) 2) ) )

(defthmd obvious-<
  (implies (< L (+ 1 (LEN (CDR NET))))
           (not (< (+ 1 (LEN (CDR NET))) L) ) )

(defthmd add-dead-neuron-to-hidden-layer-of-network-2.1.2
  (IMPLIES
   (AND (CONSP (CDR NET))
        (< 0 (+ 1 (LEN (CDDR NET))))

```

```

(< 1 (+ 1 1 (LEN (CDDR NET))))
(CONSP (CADR NET))
(<= 2 (+ 1 1 (LEN (CDDR NET))))
(ISNETWORK (CDR NET))
(ISLAYER (CADR NET))
(ISNETWORK2 (CDDR NET) (LEN S))
(ISDEADNEURON (MAKEDEADNEURON (LEN I)))
(ISNEURON (MAKEDEADNEURON (LEN I)))
(ISDEADNEURON2 (MAKEDEADNEURON (LEN I)))
(ISLAYER (ADDSYNAPSESTONEURONSOF LAYER (CADR NET)
S))
(EQUAL (LEN (ADDSYNAPSESTONEURONSOF LAYER (CADR NET)
S))
(LEN S))
(NOT (CDAR NET))
(ISNETWORK (LIST* (CONS (MAKEDEADNEURON (LEN I))
(CAR NET))
(ADDSYNAPSESTONEURONSOF LAYER (CADR NET)
S)
(CDDR NET)))
(ISLAYER (CONS (MAKEDEADNEURON (LEN I))
(CAR NET)))
(ISNETWORK (CONS (ADDSYNAPSESTONEURONSOF LAYER (CADR NET)
S)
(CDDR NET)))
(CONSP NET)
(< 0 (+ 1 1 (LEN (CDDR NET))))
(ISLAYER (CAR NET))
(CONSP (CAR NET))
(EQUAL (LEN (CAADR NET)) 1)
(CONSP (CAAR NET))
(ISNEURON (CAAR NET))
(ISSYNAPSE (CAAAR NET))
(ISLISTOFWEIGHTS (CDAAR NET))
(EQUAL (LEN (MAKEDEADNEURON (LEN I)))
(LEN I))
(TRUE-LISTP (CDDR NET))
(ISNETWORK NET)
(< 0 (LEN I))
(ISLISTOFWEIGHTS I)
(ISLISTOFWEIGHTS S)
(EQUAL (LEN (CADR NET)) (LEN S))
(EQUAL (LEN I) (+ 1 (LEN (CDAAR NET))))
(< 0 (+ -1 1 1 (LEN (CDDR NET))))
(EQUAL
(FORWARDNETWORK F (LIST (ACTIVATE F I (CAAR NET))
(CDR NET))
(FORWARDNETWORK F
(FORWARDLAYER F
(LIST (ACTIVATE F I (MAKEDEADNEURON (LEN I)))
(ACTIVATE F I (CAAR NET)))
(ADDSYNAPSESTONEURONSOF LAYER (CADR NET)
S))
(CDDR NET))))
:hints ("Goal"
:hands-off (isLayer
activate
isSynapse
isDeadNeuron
isNetwork
isNeuron)
:in-theory (enable len-0
car-front
natp-iff)
:do-not '(generalize)
:use ((:instance dead-neuron-output (f f)
(i i)
(n (MAKEDEADNEURON (LEN I))) )
(:instance zero-in-forwardLayer (f f)
(1 (CADR NET) )

```

```

                (i (list (ACTIVATE F I (CAAR NET)) ) )
                (s s) )
        (:instance isSynapse-activate (f f)
         (i i)
         (n (caar net)) ) )
:do-not-induct t)) )

(defthmd not-so-obvious-helper
  (implies (and (natp l)
                (NOT (EQUAL L 0))
                (< 0 (LEN (CDR NET)))
                (ISNETWORK (CDR NET))
                (EQUAL 1 (LEN (CAR (NTH (+ 1 L) NET))))
                (<= 2 (+ 1 (LEN (CDR NET))))
                (NOT (CDR (NTH L NET)))
                (CONSP (NTH (+ 1 L) NET))
                (CONSP (NTH L NET))
                (< L (+ 1 (LEN (CDR NET))))
                (<= 0 (+ 1 (LEN (CDAR (NTH L NET)))))
                (< 1 (+ 1 (LEN (CDR NET)))))
            (EQUAL (+ 1 (LEN (NTH (+ -1 L) (CDR NET)))) 2) )

  :hints (("Goal"
           :do-not-induct t
           :use ((:instance nth-1-cdr (n net) (l (+ 1 l)) )
                 (:instance nth-1-cdr (n net) (l l) ) ) ) )

(defthmd no-layer-forwardLayer-len
  (implies (not l)
            (equal (len (forwardLayer f i l)) 0) ) )

(defthmd not-net-helper
  (implies (<= (+ -1 l (LEN (CDR NET))) 0)
            (equal (len (cdr net)) 0) ) )

(defthmd forwardNetwork-makes-output
  (implies (and (isNetwork net)
                (isListOfWeights i)
                (equal (len i) (len (caar net))))
            (and (< 0 (len (forwardNetwork f i net)))
                 (forwardNetwork f i net) ) )

  :hints (("Goal"
           :induct (forwardNetwork f i net) )
           ("Subgoal *1/2"
            :do-not-induct t )
           ("Subgoal *1/1"
            :in-theory (enable len-0)
            :use (not-net-helper
                  (:instance isNetwork-isNetwork2-duh (n net) )
                  (:instance isLayer-isLayer2-duh (l (car net)) )
                  (:instance isNeuron-duh (n (caar net)) )
                  (:instance isListOfWeights-forwardLayer (f f) (i i) (l (car net)) )
                  (:instance forwardLayer-len (f f) (i i) (l (car net)) )
                  (:instance isNetwork-cdr (n net) ) )
            :hands-off (isNetwork
                       isLayer
                       forwardLayer)
            :do-not-induct t ) ) )

(defthmd apply-the-over-app-rule
  (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)
            (< 1 (- (len net) 1))
            (CONSP (CAR (NTH L NET)))
            (ISNETWORK (LIST* (CONS (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET)))))

```

```

(NTH L NET))
(ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S)
(CDR (AFTER (+ 1 L) NET))))
(ISNETWORK (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON
(+ 1 (LEN (CDAR (NTH L NET))))))
S L NET))
(EQUAL (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON
(+ 1 (LEN (CDAR (NTH L NET))))))
S L NET)
(APPEND (FRONT L NET)
(LIST* (CONS (MAKEDEADNEURON
(+ 1 (LEN (CDAR (NTH L NET))))))
(NTH L NET))
(ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S)
(CDR (AFTER (+ 1 L) NET))))))
(EQUAL (FORWARDNETWORK F I NET)
(FORWARDNETWORK F (FORWARDNETWORK F I (FRONT L NET))
(AFTER L NET)))
(EQUAL 1 (LEN (CAR (NTH (+ 1 L) NET))))
(NOT (CDR (NTH L NET)))
(NOT (EQUAL L 0)) )
(equal (FORWARDNETWORK F I
(ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON
(+ 1 (LEN (CDAR (NTH L NET))))))
S L NET))
(forwardNetwork f (forwardNetwork f i (front l net))
(LIST* (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
(NTH L NET))
(ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S)
(CDR (AFTER (+ 1 L) NET))) ) ) )

:hints ("Goal"
:hands-off (isLayer
activate
isSynapse
isDeadNeuron
front
after
nth
forwardLayer
forwardNetwork
isNetwork
isNeuron)
:use ((:instance car-front (i l) (x net) )
(:instance front-isNetwork (l l) (n net) )
(:instance forwardNetwork-over-app
(f f) (i i) (a (FRONT L NET) )
(b (LIST* (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
(NTH L NET))
(ADDSYNAPSESTONEURONSOF LAYER
(nth (+ 1 L) NET) S)
(CDR (AFTER (+ 1 L) NET))) ) ) )
:do-not '(generalize)
:do-not-induct t)) )

(defthmd forwardNetwork-step-over-list*
(implies (and (isNetwork (list* a b c))
(isListOfWeights i)
(equal (len i) (len (car a)))) )
(equal (forwardNetwork f i (list* a b c))
(forwardNetwork f
(forwardLayer f
(forwardLayer f i a) b) c) ) ) )

(defthmd obvious-ineq
(implies (<= (+ 1 L) (LEN NET))
(<= (+ 1 -1 L) (LEN (CDR NET))) ) )

(defthmd after-absorbs-nth
(implies (and (natp l)
(<= (+ 1 L) (len net)) )

```

```

(equal (CONS (NTH L NET) (AFTER (+ 1 L) NET))
      (after 1 net) ) )

:hints ("Goal"
       :induct (after 1 net) )
("Subgoal *1/2"
 :use (obvious-ineq
      (:instance after-cdr (i 1) (n net) )
      come-on!3)
 :do-not-induct t)
("Subgoal *1/1"
 :expand ( (AFTER 1 NET) )
 :do-not-induct t) )

(defthmd add-dead-neuron-to-hidden-layer-of-network-2.1.1
  (IMPLIES
   (AND
    (< 0 (LEN (CDR NET)))
    (< 1 (+ 1 (LEN (CDR NET))))
    (CONSP (NTH (+ 1 L) NET))
    (<= 2 (+ 1 (LEN (CDR NET))))
    (ISNETWORK (FRONT L NET))
    (ISNETWORK (AFTER (+ 1 L) NET))
    (ISNETWORK (CDR NET))
    (ISNETWORK2 (CDR (AFTER (+ 1 L) NET))
                 (LEN S))
    (ISNEURON (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))))
    (ISDEADNEURON2 (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))))
    (EQUAL
     (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
                                         S L NET)
     (APPEND (FRONT L NET)
              (LIST* (CONS (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
                               (NTH L NET))
                        (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET)
                                                       S)
                        (CDR (AFTER (+ 1 L) NET))))))
    (IS LAYER (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET)
                                             S))
    (EQUAL (LEN (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET)
                                             S))
            (LEN S))
    (CONSP (NTH L NET))
    (EQUAL 1 (LEN (CAR (NTH (+ 1 L) NET))))
    (EQUAL (FORWARDNETWORK F I NET)
            (FORWARDNETWORK F (FORWARDNETWORK F I (FRONT L NET))
                               (AFTER L NET)))
    (NOT (CDR (NTH L NET)))
    (ISNETWORK
     (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
                                         S L NET))
    (IS LAYER (CONS (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
                               (NTH L NET))
    (EQUAL (CAR (AFTER (+ 1 L) NET))
            (NTH (+ 1 L) NET))
    (ISNETWORK (CONS (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET)
                                                       S)
                    (CDR (AFTER (+ 1 L) NET))))
    (EQUAL (AFTER L NET)
            (LIST* (NTH L NET)
                   (NTH (+ 1 L) NET)
                   (AFTER (+ 2 L) NET)))
    (NOT (EQUAL L 0))
    (CONSP NET)
    (< L (+ 1 (LEN (CDR NET))))
    (<= 0 (+ 1 (LEN (CDAR (NTH L NET))))
    (IS LAYER (CAR NET))
    (CONSP (CAR NET))
    (ISNETWORK2 (CDR NET)
                 (+ 1 (LEN (CDAR NET))))
    (CONSP (CAAR NET))

```

```

(ISLAYER2 (CDAR NET) (LEN I))
(ISNEURON (CAAR NET))
(ISSYNAPSE (CAAAR NET))
(ISLISTOFWEIGHTS (CDAAR NET))
(ISLAYER (NTH L NET))
(ISLAYER (NTH (+ 1 L) NET))
(ISDEADNEURON (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))))
(CONSP (CAR (NTH L NET)))
(ISNEURON (CAR (NTH L NET)))
(EQUAL (LEN (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))))
      (+ 1 (LEN (CDAR (NTH L NET))))))
(TRUE-LISTP (CDAR NET))
(TRUE-LISTP (CDR NET))
(CONSP (AFTER (+ 1 L) NET))
(ISSYNAPSE (CAAR (NTH L NET)))
(ISLISTOFWEIGHTS (CDAR (NTH L NET)))
(ISNETWORK NET)
(< 0 (LEN I))
(ISLISTOFWEIGHTS I)
(ISLISTOFWEIGHTS S)
(EQUAL (LEN (NTH (+ 1 L) NET)) (LEN S))
(EQUAL (LEN I) (+ 1 (LEN (CDAAR NET))))
(INTEGERP L)
(<= 0 L)
(< L (+ -1 1 (LEN (CDR NET))))
(EQUAL
 (FORWARDNETWORK F I NET)
 (FORWARDNETWORK
  F I
  (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
                                S L NET))))

```

```

:hints ("Goal"
  :hands-off (isLayer
              activate
              isSynapse
              isDeadNeuron
              front
              after
              nth
              forwardLayer
              forwardNetwork
              isNetwork
              isNeuron)
  :in-theory (enable len-0
              natp-iff)
  :do-not '(generalize)
  :use ((:instance front-len (i l) (w net) )
        obvious-<
        (:instance car-front (i l) (x net) )
        (:instance nth-front (a (+ -1 l)) (b l) (n net) )
        (:instance len-car-addSynapsesToNeuronsOfLayer
          (l (NTH (+ 1 L) NET)) (s s) )
        (:instance add-layer-to-network (l (CONS (MAKEDEADNEURON
          (LEN (CAR (NTH L NET))))
          (NTH L NET)) )
          (n (CONS (ADDSYNAPSESTONEURONSOFPLAYER
          (NTH (+ 1 L) NET) S)
          (CDR (AFTER (+ 1 L) NET)))) ) )
        (:instance isNetwork-app (a (FRONT L NET) )
          (b (LIST* (CONS (MAKEDEADNEURON
          (LEN (CAR (NTH L NET))))
          (NTH L NET))
          (ADDSYNAPSESTONEURONSOFPLAYER
          (nth (+ 1 L) NET) S)
          (CDR (AFTER (+ 1 L) NET)))) ) )
        not-so-obvious-helper
        (:instance forwardNetwork-makes-output (net (front l net))
          (f f) (i i) )
        (:instance forwardNetwork-over-app
          (f f) (i i) (a (FRONT L NET) )

```

```

        (b (LIST* (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
                    (NTH L NET))
            (ADDSYNAPSESTONEURONSOF LAYER (nth (+ 1 L) NET) S)
            (CDR (AFTER (+ 1 L) NET))) ) )
    (:instance isListOfWeights-forwardNetwork (f f)
        (i i)
        (n (FRONT L NET) ) )
    (:instance add-dead-neuron-to-first-layer (f f)
        (l l)
        (i (FORWARDNETWORK
            F I (FRONT L NET)) )
        (net net) )
    (:instance len-forwardNetwork (f f) (i i) (n (front l net)) )
    (:instance nth-layer-in-network-consistent (l (+ -1 l)) (n net) )
    come-on!3)
:do-not-induct t)
("Subgoal 1"
:use (apply-the-over-app-rule
    (:instance dead-neuron-output (f f)
        (i (forwardNetwork f i (front l net)))
        (n (MAKEDEADNEURON
            (LEN (CAR (NTH L NET)))))) )

:hands-off (isLayer
    activate
    addNeuronToLayerOfNetwork
    forwardNetwork
    forwardLayer
    isSynapse
    makeDeadNeuron
    isDeadNeuron
    front
    after
    nth
    isNetwork
    isNeuron) )
("Subgoal 1.1"
:use (after-absorbs-nth
    (:instance forwardNetwork-step-over-list* (f f)
        (i (FORWARDNETWORK
            F I (FRONT L NET)))
        (a (NTH L NET) )
        (b (NTH (+ 1 L) NET) )
        (c (AFTER (+ 2 L) NET) ) )
    (:instance after-isNetwork (l l) (n net) )
    (:instance forwardNetwork-step-over-list*
        (f f) (i (FORWARDNETWORK F I (FRONT L NET)))
        (a (CONS (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))
            (NTH L NET)) )
        (b (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S) )
        (c (CDR (AFTER (+ 1 L) NET)) ) )
    (:instance isSynapse-activate (f f)
        (i (FORWARDNETWORK F I (FRONT L NET)) )
        (n (CAR (NTH L NET)) ) )
    (:instance zero-in-forwardLayer (f f)
        (l (nth (+ 1 l) NET) )
        (i (list (ACTIVATE
            F (FORWARDNETWORK
                F I (FRONT L NET))
            (CAR (nth l NET)) ) )
        (s s) ) ) )

(defthmd use-your-knowledge!
    (implies (EQUAL (NTH (+ -1 L) (FRONT L NET))
        (NTH (+ -1 L) NET))
        (EQUAL (LEN (NTH (+ -1 L) NET))
            (LEN (NTH (+ -1 L) (FRONT L NET)))))) )

(defthmd add-dead-neuron-to-hidden-layer-of-network-1.1
    (IMPLIES
        (AND (CONSP NET)
            (< L (+ 1 (LEN (CDR NET))))))

```

```

(=< 0 (+ 1 (LEN (CDAR (NTH L NET)))))
(ISLAYER (CAR NET))
(CONSP (CAR NET))
(ISNETWORK2 (CDR NET)
  (+ 1 (LEN (CDAR NET))))
(CONSP (CAAR NET))
(ISLAYER2 (CDAR NET) (LEN I))
(ISNEURON (CAAR NET))
(ISSYNAPSE (CAAAR NET))
(ISLISTOFWEIGHTS (CDAAR NET))
(ISLAYER (NTH L NET))
(ISLAYER (NTH (+ 1 L) NET))
(ISDEADNEURON (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET)))))
(CONSP (CAR (NTH L NET)))
(ISLAYER2 (CDR (NTH L NET))
  (+ 1 (LEN (CDAR (NTH L NET)))))
(ISNEURON (CAR (NTH L NET)))
(EQUAL (LEN (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET)))))
  (+ 1 (LEN (CDAR (NTH L NET)))))
(TRUE-LISTP (CDAR NET))
(TRUE-LISTP (CDR NET))
(CONSP (AFTER (+ 1 L) NET))
(ISSYNAPSE (CAAR (NTH L NET)))
(ISLISTOFWEIGHTS (CDAR (NTH L NET)))
(ISLAYER (CDR (NTH L NET)))
(ISNETWORK NET)
(< 0 (LEN I))
(ISLISTOFWEIGHTS I)
(ISLISTOFWEIGHTS S)
(EQUAL (LEN (NTH (+ 1 L) NET)) (LEN S))
(EQUAL (LEN I) (+ 1 (LEN (CDAAR NET))))
(INTEGERP L)
(<= 0 L)
(< L (+ -1 1 (LEN (CDR NET)))))
(EQUAL
(FORWARDNETWORK F I NET)
(FORWARDNETWORK
  F I
  (ADDNEURONTOLAYEROFNETWORK (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET)))))
    S L NET)))

:hints ("Goal"
:cases ( (equal 1 0)
  (not (equal 1 0)) )
:hands-off (isLayer
  activate
  isSynapse
  isDeadNeuron
  front
  after
  nth
  addNeuronToLayerOfNetwork
  forwardLayer
  forwardNetwork
  isNetwork
  isNeuron)
:use ( (:instance isDeadNeuron-duh
  (n (MAKEDEADNEURON (+ 1 (LEN (CDAR (NTH L NET))))) )
  (:instance forwardNetwork-over-front-after (1 1) (f f) (i i) (n net) )
  (:instance first-two-of-after (1 1) (n net) )
  after-absorbs-nth
  (:instance after-absorbs-nth (1 (+ 1 1)) (net net) )
  grrr!2
  (:instance after-isNetwork (1 1) (n net) )
  (:instance addNeuronToLayerOfNetwork-app-front-after
    (n (makeDeadNeuron (len (car (nth 1 net))))
      (s s) (1 1) (net net) ) )
:in-theory (enable len-0
  natp-iff)
:do-not '(generalize)
:do-not-induct t)

```

```

("Subgoal 2.1"
 :hands-off (isLayer
             makeDeadNeuron
             activate
             isSynapse
             isDeadNeuron
             addNeuronToLayerOfNetwork
             isNetwork
             isNeuron)
 :use ((:instance forwardNetwork-step-over-list* (f f)
                (i i)
                (a (car NET) )
                (b (cadr NET) )
                (c (caddr NET) ) )
      (:instance forwardNetwork-step-over-list* (f f)
                (i i)
                (a (car NET) )
                (b (cadr NET) )
                (c (caddr NET) ) )
      (:instance isListOfWeights-forwardLayer (f f) (i i) (l (cdar net)))
      (:instance forwardLayer-len (f f) (i i) (l (cdar net)))
      (:instance dead-neuron-output (f f)
                (i i)
                (n (MAKEDEADNEURON (LEN I))))
      (:instance isSynapse-activate (f f) (i i) (n (caar net)))
      (:instance zero-in-forwardLayer (f f)
                (l (CADR NET) )
                (i (CONS (ACTIVATE F I (CAAR NET))
                        (FORWARDLAYER
                         F I (CDAR NET))))
                (s s) ) ) )

("Subgoal 1.1"
 :use (after-absorbs-nth
       come-on!3
       (:instance car-front (i l) (x net) )
       (:instance after-absorbs-nth (l (+ 1 l)) (net net) )
       (:instance front-isNetwork (l l) (n net) )
       (:instance isListOfWeights-forwardNetwork (f f)
                (i i)
                (n (FRONT L NET) ) )
       (:instance after-isNetwork (l l) (n net) )
       (:instance isSynapse-activate (f f)
                (i (FORWARDNETWORK F I (FRONT L NET)))
                (n (CAR (NTH L NET) ) ) )
       (:instance len-forwardNetwork (f f) (i i) (n (front l net) ) )
       (:instance front-len (i l) (w net) )
       (:instance nth-front (a (+ -1 l)) (b l) (n net) )
       (:instance nth-layer-in-network-consistent (l (+ -1 l)) (n net) ) ) )

("Subgoal 1.1.1"
 :use ((:instance cdr-after2 (i (+ 1 l)) (x net) )
      (:instance after-isNetwork (l (+ 1 l)) (n net) )
      (:instance isNetwork-isNetwork2-duh (n (after (+ 1 l) net)) )
      (:instance len-addSynapsesToNeuronsOfLayer
                (l (NTH (+ 1 L) NET)) (s s) )
      (:instance forwardNetwork-step-over-list* (f f)
                (i (FORWARDNETWORK
                   F I (FRONT L NET)))
                (a (NTH L NET) )
                (b (NTH (+ 1 L) NET) )
                (c (AFTER (+ 2 L) NET) ) )
      (:instance isLayer-addSynapsesToNeuronsOfLayer
                (l (nth (+ 1 L) NET)) (s s) )
      (:instance add-layer-to-network2 (l (ADDSYNAPSESTONEURONSOF LAYER
                (nth (+ 1 L) NET) S) )
                (n (AFTER (+ 2 L) NET) ) ) ) )

("Subgoal 1.1.1.1"
 :hands-off (isLayer
             activate
             isSynapse
             isDeadNeuron
             front)

```

```

        after
        nth
        makeDeadNeuron
        addNeuronToLayerOfNetwork
        forwardNetwork
        isNetwork
        isNeuron)
:use (use-your-knowledge!
      (:instance add-neurons-to-layer
        (l (NTH L NET) )
        (n (MAKEDEADNEURON (LEN (CAR (NTH L NET)))) ) )
      (:instance add-layer-to-network
        (l (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
                (NTH L NET)) )
        (n (CONS (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S)
                (AFTER (+ 2 L) NET)) ) )
      (:instance dead-neuron-output (f f)
        (i (forwardNetwork f i (front l net)) )
        (n (MAKEDEADNEURON
            (LEN (CAR (NTH L NET)))) ) )
      (:instance forwardNetwork-step-over-list*
        (f f) (i (FORWARDNETWORK F I (FRONT L NET)))
        (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
                (NTH L NET)) )
        (b (ADDSYNAPSESTONEURONSOF LAYER (NTH (+ 1 L) NET) S) )
        (c (AFTER (+ 2 L) NET) ) )
      (:instance isListOfWeights-forwardLayer (f f)
        (i (FORWARDNETWORK
            F I (FRONT L NET)) )
        (l (CDR (NTH L NET)) ) )
      (:instance forwardLayer-len (f f)
        (i (FORWARDNETWORK F I (FRONT L NET)) )
        (l (CDR (NTH L NET)) ) )

      (:instance zero-in-forwardLayer
        (f f) (l (nth (+ 1 l) NET) )
        (i (LIST* (ACTIVATE F (FORWARDNETWORK F I (FRONT L NET))
                (CAR (NTH L NET)))
                (FORWARDLAYER F (FORWARDNETWORK F I (FRONT L NET))
                (CDR (NTH L NET)))) )
        (s s) )
      (:instance nth-layer-in-network-consistent (l l) (n net) )
      (:instance len-car-addSynapsesToNeuronsOfLayer
        (l (NTH (+ 1 L) NET)) (s s) )
      (:instance isNetwork-app (a (FRONT L NET) )
        (b (LIST* (CONS (MAKEDEADNEURON
            (LEN (CAR (NTH L NET))))
            (NTH L NET))
            (ADDSYNAPSESTONEURONSOF LAYER
            (nth (+ 1 L) NET) S)
            (AFTER (+ 2 L) NET)) ) )
      (:instance forwardNetwork-over-app
        (f f) (i i) (a (FRONT L NET) )
        (b (LIST* (CONS (MAKEDEADNEURON (LEN (CAR (NTH L NET))))
            (NTH L NET))
            (ADDSYNAPSESTONEURONSOF LAYER (nth (+ 1 L) NET) S)
            (AFTER (+ 2 L) NET)) ) ) )

(defthmd isNetwork-not-isNetwork2-nil
  (implies (and (isNetwork net)
                (not (isNetwork (cdr net)))) )
  (not (cdr net)) ) )

(defthmd same-nth-helper
  (implies (< 1 (len NET))
    (equal (NTH (+ -1 (LEN (CDR NET)))
              (CDR NET))
           (NTH (+ -1 (LEN NET))
                NET) ) )

:hints ("Goal'")

```

```

:do-not-induct t
:use (:instance grrr! (n net) )
:expand ( (NTH (+ -1 1 (LEN (CDR NET))) NET) ) ) )

(defthmd not-zero-dead-neuron-helper
  (implies (isNetwork net)
    (< 0 (len (car (nth (- (len net) 1) net))) ) )

  :hints (("Goal"
    :induct (len net)
    :in-theory (enable len-0) )
    ("Subgoal *1/1"
    :do-not-induct t
    :use (same-nth-helper
      isNetwork-not-isNetwork2-nil)
    :cases ( (isNetwork (cdr net)) ) ) ) )

(defthmd elementary-arithmetic
  (equal (+ -1 -1 1 1 1 (LEN (CDDR NET))) (+ -1 1 1 (LEN (CDDR NET))) ) )

(defthmd nth-nil-helper
  (implies (and (consp net)
    (true-listp net))
    (equal (NTH (+ -1 1 1 (LEN (CDR NET))) NET) nil) )

  :hints (("Goal"
    :induct (len net) )
    ("Subgoal *1/2"
    :expand ( (NTH (+ -1 1 1 1 (LEN (CDDR NET))) NET) )
    :use (elementary-arithmetic)
    :do-not-induct t)
    ("Subgoal *1/3"
    :do-not-induct t)) )

(defthmd isNetwork-<-1
  (implies (and (isNetwork net)
    (not (equal (len net) 1)) )
    (< 1 (len net)) ) )

(defthmd len-cdr-<-1
  (implies (< 1 (len net))
    (< 0 (len (cdr net))) ) )

(defthmd mensch!
  (equal (+ -1 1 (LEN (CDR NET))) (LEN (CDR NET)) ) )

(defthmd mensch!2
  (equal (+ 1 -1 (LEN (CDR NET))) (LEN (CDR NET)) ) )

(defthmd stupidity
  (equal (+ 1 -1 (LEN (CDR NET))) (LEN (CDR NET)) ) )

(defthmd isNetwork-app-specific-helper
  (implies (and (isNetwork net)
    (ISNETWORK (FRONT (LEN (CDR NET)) NET))
    (ISNETWORK (LIST (CONS (MAKEDEADNEURON
      (LEN (CAR (NTH (LEN (CDR NET)) NET))))
      (NTH (LEN (CDR NET)) NET)))) )
    (ISNETWORK (APPEND (FRONT (LEN (CDR NET)) NET)
      (LIST (CONS (MAKEDEADNEURON
        (LEN (CAR (NTH (LEN (CDR NET)) NET))))
        (NTH (LEN (CDR NET)) NET)))))) )

  :hints (("Goal"
    :hands-off ( front
      after
      nth )
    :use (mensch!
      stupidity
      (:instance nth-layer-in-network-consistent
        (1 (+ -1 (len (cdr net)))) (n net) )

```

```

      (:instance nth-front (a (+ -1 (LEN (CDR NET))))
        (b (LEN (CDR NET))) (n net) )
      (:instance front-len (i (+ -1 (LEN NET)) ) (w net) )
      (:instance makeDeadNeuron-len
        (i (len (car (nth (- (len net) 1) net))) ) )
      (:instance isNetwork-app (a (FRONT (LEN (CDR NET)) NET) )
        (b (LIST (CONS (MAKEDEADNEURON
          (LEN (CAR (NTH (LEN (CDR NET))
            NET))))
          (NTH (LEN (CDR NET)) NET)))))) )

:do-not-induct t)) )

(defthmd forwardNetwork-step-over-singleton-list
  (implies (and (isNetwork (list a))
    (isListOfWeights i)
    (equal (len i) (len (car a))) )
    (equal (forwardNetwork f i (list a))
      (forwardLayer f i a) ) ) )

(defthmd len-nth-front-helper
  (implies (and (isNetwork net)
    (consp (cdr net)) )
    (EQUAL (LEN (NTH (+ -1 (LEN (CDR NET))) (FRONT (LEN (CDR NET)) NET)))
      (LEN (CAR (NTH (LEN (CDR NET)) (FRONT (+ 1 (LEN (CDR NET)) NET)))))) )

:hints ("Goal"
  :hands-off (isNetwork
    isLayer
    front
    nth
    isNeuron)
  :in-theory (enable len-1)
  :use (mensch!
    mensch!2
    (:instance isNetwork-true-listp (n net) )
    (:instance len-cdr2 (l net) )
    (:instance isNetwork-cdr (n net) )
    (:instance isNetwork-isNetwork2-duh (n net) )
    (:instance isLayer-isLayer2-duh (l (car net)) )
    (:instance nth-in-network-isLayer (l (LEN (cdr NET))) (n net) )
    (:instance isLayer-isLayer2-duh (l (NTH (LEN (CDR NET)) NET)) )
    (:instance isNeuron-duh (n (car (NTH (LEN (CDR NET)) NET))) )
    (:instance isNeuron-duh (n (caar net)) )
    (:instance nth-layer-in-network-consistent
      (l (+ -1 (len (cdr net)))) (n net) )
    (:instance nth-front (a (LEN (cdr NET))) (b (+ 1 (LEN (cdr NET))))
      (n net) )
    (:instance nth-front (a (+ -1 (LEN (cdr NET)))) (b (LEN (cdr NET)))
      (n net) ))

:do-not-induct t)) )

(defthmd obvious-math
  (equal (+ -1 1 1 (LEN (CDR NET))) (+ 1 (LEN (CDR NET))) ) )

(defthmd one-step-of-forwardLayer
  (implies (and (isLayer (cons a b))
    (isListOfWeights i)
    (equal (len i) (len a)) )
    (equal (forwardLayer f i (cons a b))
      (cons (activate f i a)
        (forwardLayer f i b)) ) ) )

(defthmd after-at-end
  (implies (and (true-listp net)
    (consp net) )
    (equal (AFTER (+ -1 (LEN NET)) NET)
      (cons (nth (+ -1 (LEN NET)) net) nil) ) )

:hints ("Goal"
  :induct (len net) )
"Subgoal *1/2"

```

```

:do-not-induct t)
("Subgoal *1/1"
:do-not '(generalize)
:use ( (:instance grrr! (n net) ) )
:expand ( (AFTER (+ -1 1 (LEN (CDR NET))) NET)
          (NTH (+ -1 1 (LEN (CDR NET))) NET) )
:do-not-induct t) )

;-----

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

(defthmd add-dead-neuron-to-input-or-hidden-layer-of-network
  (implies (and (isNetwork net)
                (< 0 (len i))
                (isListOfWeights i)
                (isListOfWeights s)
                (equal (len (nth (+ 1 1) net)) (len s))
                (equal (len i) (len (caar net)))
                (natp 1)
                (< 1 (- (len net) 1)) )
            (equal (forwardNetwork f i net)
                  (forwardNetwork f i
                                (addNeuronToLayerOfNetwork (makeDeadNeuron
                                                             (len (car (nth 1 net))))
                                                             s 1 net)) ) )

:hints ("Goal"
:hands-off (isLayer
            activate
            isSynapse
            isDeadNeuron
            isNetwork
            isNeuron)
:in-theory (enable len-0
              car-front
              natp-iff)
:do-not '(generalize)
:cases ( (NOT (ISLAYER (CDR (nth 1 NET))))
         (ISLAYER (CDR (nth 1 NET))) )
:use ( (:instance natp-iff (x 1) )
      strange-cdr-len-helper
      very-obvious!
      (:instance isNetwork-isNetwork2-duh (n net) )
      (:instance isLayer-isLayer2-duh (l (car net)) )
      (:instance isNeuron-duh (n (caar net)) )
      (:instance nth-in-network-isLayer (l 1) (n net) )
      (:instance nth-in-network-isLayer (l (+ 1 1)) (n net) )
      (:instance makeDeadNeuron-isDeadNeuron (i (LEN (CAR (NTH L NET)))) )
      (:instance isLayer-isLayer2-duh (l (nth 1 net)) )
      (:instance makeDeadNeuron-len (i (LEN (CAR (NTH L NET))) ) )
      (:instance isLayer-true-listp (l (car net)) )
      (:instance isNetwork-true-listp (n net) )
      (:instance consp-after-helper (k (+ 1 L)) (w net) )
      (:instance isNeuron-duh (n (car (nth 1 net))) ) )
:do-not-induct t )
("Subgoal 2.1"
:cases ( (equal 1 0)
        (not (equal 1 0)) )
:use (strange-ineq-to-eq-helper
      stranger-helper-cdr-net
      len-cdr-net-helper
      obvious-layer-consp-helper
      after-nth-helper
      strange-nth-after-helper
      another-strange-helper
      obviously-2
      (:instance front-isNetwork (l 1) (n net) )
      (:instance after-isNetwork (l (+ 1 1)) (n net) )
      (:instance isNetwork-cdr (n net) )

```

```

(:instance isNetwork-isNetwork2-duh (n (AFTER (+ 1 L) NET)) )
(:instance isDeadNeuron-duh (n (makeDeadNeuron
                               (len (car (nth 1 net)))))) )
(:instance addNeuronToLayerOfNetwork-app-front-after
  (n (makeDeadNeuron (len (car (nth 1 net))))
   (s s) (1 1) (net net) )
)
(:instance isLayer-addSynapsesToNeuronsOfLayer
  (1 (CAR (AFTER (+ 1 L) NET))) (s s) )
)
(:instance isLayer-addSynapsesToNeuronsOfLayer (1 (nth (+ 1 1) net))
  (s s) )
)
(:instance len-addSynapsesToNeuronsOfLayer (1 (NTH (+ 1 L) NET))
  (s s) )
)
(:instance nth-layer-in-network-consistent (1 1) (n net) )
(:instance forwardNetwork-over-front-after (1 1) (f f) (i i) (n net) )
(:instance isLayer-not-isLayer2-nil (1 (nth 1 net) ) )
(:instance isNetwork-addNeuronToLayerOfNetwork
  (n (MAKEDEADNEURON (LEN (CAR (NTH L NET)))) ) )
  (s s) (1 1) (net net) )
)
(:instance add-neurons-to-layer (1 (NTH L NET) )
  (n (MAKEDEADNEURON
      (LEN (CAR (NTH L NET)))))) ) )
)
(:instance nth-car-after (1 (+ 1 1)) (n net) )
(:instance add-layer-to-network2 (1 (ADDSYNAPSESTONEURONSOF LAYER
  (CAR (AFTER (+ 1 L) NET)) S) )
  (n (CDR (AFTER (+ 1 L) NET)) ) ) )
)
(:instance car-front (i 1) (x net) )
(:instance first-two-of-after (1 1) (n net) ) )
)
("Subgoal 2.1.2''')
:use (add-dead-neuron-to-hidden-layer-of-network-2.1.2) )
("Subgoal 2.1.1'4")
:use (add-dead-neuron-to-hidden-layer-of-network-2.1.1) )
("Subgoal 1.1")
:use (add-dead-neuron-to-hidden-layer-of-network-1.1) ) )

```

; 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.

```

(defthmd add-dead-neuron-to-output-layer-of-network
  (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)) ) )
    )
)
:hints ("Goal"
  :do-not-induct t
  :cases ( (equal (len net) 1)
    (not (equal (len net) 1)) )
  :use (not-zero-dead-neuron-helper
    (:instance isNetwork-true-listp (n net) )
    (:instance front-isNetwork (1 (- (len net) 1)) (n net) )
    (:instance after-isNetwork (1 (- (len net) 1)) (n net) )
    (:instance dead-neuron-output
      (f f) (i i)
      (n (MAKEDEADNEURON (len (car (nth (- (len net) 1) net)))))) ) )
    (:instance isNetwork-isNetwork2-duh (n net) )
    (:instance isLayer-isLayer2-duh (1 (car net)) )
    (:instance isNeuron-duh (n (caar net)) )
    (:instance nth-layer-in-network-consistent (1 (+ -1 (len net)))
      (n net) )
    (:instance isDeadNeuron-duh
      (n (makeDeadNeuron (len (car (nth (- (len net) 1) net)))))) ) )
  )

```

```

(:instance makeDeadNeuron-isDeadNeuron
  (i (len (car (nth (- (len net) 1) net))) ) )
(:instance makeDeadNeuron-len
  (i (len (car (nth (- (len net) 1) net))) ) )
(:instance addNeuronToLayerOfNetwork-app-front-after
  (n (makeDeadNeuron (len (car (nth (- (len net) 1) net)))))
  (s s) (l (+ -1 (len net)) ) (net net) )
(:instance forwardNetwork-over-front-after (l (- (len net) 1)) (f f)
  (i i) (n net) )

:hands-off (isLayer
  activate
  isSynapse
  isDeadNeuron
  forwardLayer
  forwardNetwork
  makeDeadNeuron
  isNetwork
  isNeuron)
:in-theory (enable len-0
  car-front
  natp-iff )

("Subgoal 2"
:hands-off (isLayer
  activate
  isSynapse
  isDeadNeuron
  isNetwork
  isNeuron))

("Subgoal 1"
:do-not '(generalize)
:hands-off (isLayer
  front
  nth
  activate
  isSynapse
  isDeadNeuron
  forwardLayer
  forwardNetwork
  makeDeadNeuron
  isNetwork
  isNeuron)

:use (isNetwork-<-1
  len-cdr-<-1
  (:instance front-len (i (+ -1 (LEN NET)) ) (w net) )
  isNetwork-app-specific-helper
  (:instance isNetwork-true-listp (n net) )
  (:instance after-true-listp (j (- (len net) 1) ) (i net) )
  (:instance after-all-nil (n (cdr net)) )
  (:instance nth-in-network-isLayer (l (+ -1 (LEN NET))) (n net) )
  (:instance add-neurons-to-layer (l (NTH (+ -1 (LEN NET)) NET) )
    (n (MAKEDEADNEURON
      (LEN (CAR
        (NTH (+ -1 (LEN NET))
          NET)))) ) )

  (:instance one-layer-isNetwork
    (l (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
      (NTH (+ -1 (LEN NET)) NET)) ) )
  (:instance isNetwork-addNeuronToLayerOfNetwork
    (n (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET)))))
    (s s) (l (+ -1 (len net)) ) (net net) )
  (:instance forwardNetwork-over-app
    (f f) (i i) (a (FRONT (- (len net) 1) NET) )
    (b (LIST (CONS (MAKEDEADNEURON
      (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
      (NTH (+ -1 (LEN NET)) NET)) ) )
  (:instance len-forwardNetwork (f f) (i i)
    (n (front (+ -1 (LEN NET)) net)) )
  (:instance isListOfWeights-forwardNetwork
    (f f) (i i)
    (n (FRONT (+ -1 (LEN NET)) NET) ) )

stupidity

```

```

    mensch!
    nth-nil-helper )
("Subgoal 1.8"
 :use (:instance nth-layer-in-network-consistent
        (l (+ -1 (len net))) (n net) )
      (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
      len-nth-front-helper
      (:instance one-step-of-forwardLayer
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
        (b (NTH (LEN (CDR NET)) NET) ) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
                (NTH (+ -1 (LEN NET)) NET) ) ) )
      (:instance dead-neuron-output (f f)
        (i (FORWARDNETWORK
            F I (FRONT (LEN (CDR NET)) NET)) )
        (n (MAKEDEADNEURON
            (LEN (CAR (NTH (LEN (CDR NET))
                          NET)))) ) )

      (:instance after-at-end (net (cdr net)) )
      (:instance nth-1-cdr (n net) (l (len (cdr net))) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.7"
 :use (:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
        (n net))
      (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
      len-nth-front-helper
      (:instance one-step-of-forwardLayer
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
        (b (NTH (LEN (CDR NET)) NET) ) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
                (NTH (+ -1 (LEN NET)) NET) ) ) )
      (:instance dead-neuron-output
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
      (:instance after-at-end (net (cdr net)) )
      (:instance nth-1-cdr (n net) (l (len (cdr net))) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.6"
 :use (:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
        (n net) )
      (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
      len-nth-front-helper
      (:instance one-step-of-forwardLayer
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
        (b (NTH (LEN (CDR NET)) NET) ) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
                (NTH (+ -1 (LEN NET)) NET) ) ) )
      (:instance dead-neuron-output
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
      (:instance after-at-end (net (cdr net)) )
      (:instance nth-1-cdr (n net) (l (len (cdr net))) )
      (:instance forwardNetwork-step-over-singleton-list
        (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
        (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.5"
 :use (:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
        (n net) )

```

```

(:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
len-nth-front-helper
(:instance one-step-of-forwardLayer
  (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
  (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
  (b (NTH (LEN (CDR NET)) NET) ) )
(:instance forwardNetwork-step-over-singleton-list
  (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
  (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
    (NTH (+ -1 (LEN NET)) NET)) ) )
(:instance dead-neuron-output
  (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
  (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
(:instance after-at-end (net (cdr net)) )
(:instance nth-1-cdr (n net) (l (len (cdr net))) )
(:instance forwardNetwork-step-over-singleton-list
  (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
  (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.4"
 :use ((:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
  (n net))

  (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
  len-nth-front-helper
  (:instance one-step-of-forwardLayer
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
    (b (NTH (LEN (CDR NET)) NET) ) )
  (:instance forwardNetwork-step-over-singleton-list
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
      (NTH (+ -1 (LEN NET)) NET)) ) )
  (:instance dead-neuron-output (f f)
    (i (FORWARDNETWORK
      F I (FRONT (LEN (CDR NET)) NET)) )
    (n (MAKEDEADNEURON
      (LEN (CAR (NTH (LEN (CDR NET))
        NET)))) ) )

  (:instance after-at-end (net (cdr net)) )
  (:instance nth-1-cdr (n net) (l (len (cdr net))) )
  (:instance forwardNetwork-step-over-singleton-list
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.3"
 :use ((:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
  (n net))

  (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
  len-nth-front-helper
  (:instance one-step-of-forwardLayer
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
    (b (NTH (LEN (CDR NET)) NET) ) )
  (:instance forwardNetwork-step-over-singleton-list
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
      (NTH (+ -1 (LEN NET)) NET)) ) )
  (:instance dead-neuron-output
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
  (:instance after-at-end (net (cdr net)) )
  (:instance nth-1-cdr (n net) (l (len (cdr net))) )
  (:instance forwardNetwork-step-over-singleton-list
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.2"
 :use ((:instance nth-layer-in-network-consistent (l (+ -1 (len net)))
  (n net))

  (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
  len-nth-front-helper
  (:instance one-step-of-forwardLayer
    (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
    (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )

```

```

      (b (NTH (LEN (CDR NET)) NET) ) )
    (:instance forwardNetwork-step-over-singleton-list
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
        (NTH (+ -1 (LEN NET)) NET)) ) )
    (:instance dead-neuron-output
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
    (:instance after-at-end (net (cdr net)) )
    (:instance nth-1-cdr (n net) (l (len (cdr net)))) )
    (:instance forwardNetwork-step-over-singleton-list
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )
("Subgoal 1.1"
 :use ((:instance nth-layer-in-network-consistent (l (+ -1 (len net))
      (n net) )
    (:instance nth-front (a (+ -1 (LEN NET))) (b (LEN NET)) (n net) )
    len-nth-front-helper
    (:instance one-step-of-forwardLayer
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (a (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) )
      (b (NTH (LEN (CDR NET)) NET) ) )
    (:instance forwardNetwork-step-over-singleton-list
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (a (CONS (MAKEDEADNEURON (LEN (CAR (NTH (+ -1 (LEN NET)) NET))))
        (NTH (+ -1 (LEN NET)) NET)) ) )
    (:instance dead-neuron-output
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (n (MAKEDEADNEURON (LEN (CAR (NTH (LEN (CDR NET)) NET)))) ) )
    (:instance after-at-end (net (cdr net)) )
    (:instance nth-1-cdr (n net) (l (len (cdr net)))) )
    (:instance forwardNetwork-step-over-singleton-list
      (f f) (i (FORWARDNETWORK F I (FRONT (LEN (CDR NET)) NET)) )
      (a (NTH (+ -1 (LEN (CDR NET))) (CDR NET)) ) ) )

```

```

;-----
****

```

replace.lisp

```
****
```

```
; Replace - functions for replacing neurons in a network
(in-package "ACL2")
```

```
(include-book "dead")
```

```
;-----
```

```
; replaceValInList replaces the value in list l at index
; i with the value n.
```

```
(defun replaceValInList (j n l)
  (if (zp j)
      (cons n (cdr l))
      (cons (car l)
            (replaceValInList (- j 1) n (cdr l)))))
```

```
; replaceNeuronInLayerOfNetwork replaces the neuron at index i
; of the layer at index l in the network net with neuron n.
```

```
(defun replaceNeuronInLayerOfNetwork (j l n net)
  (if (zp l)
      (cons (replaceValInList j n (car net))
            (cdr net))
      (cons (car net)
            (replaceNeuronInLayerOfNetwork j (- l 1) n (cdr net)))))
```

```
;-----
```

```
; The following are several uninteresting lemmas. No interesting
; theorems are contained in this book.
```

```

(defthmd replaceValInList-front-after
  (implies (and (natp j)
                (consp l)
                (< j (len l)) )
            (equal (replaceValInList j n l)
                  (append (front j l)
                          (cons n (after (+ j 1) l))) ) )

  :hints (("Goal"
           :induct (replaceValInList j n l) ) ) )

(defthmd len-neuron-in-after
  (implies (and (isLayer l)
                (natp j)
                (< j (len l)) )
            (equal (len (car l))
                  (len (car (after j l))) ) ) )

(defthmd isLayer-replaceValInList
  (implies (and (isLayer l)
                (isNeuron n)
                (equal (len (car l)) (len n))
                (natp j)
                (< j (len l)) )
            (isLayer (replaceValInList j n l)) )

  :hints (("Goal"
           :use ((:instance car-front (i j) (x l) )
                 isLayer-true-listp
                 replaceValInList-front-after
                 isLayer-isLayer2-duh
                 (:instance isLayer-isLayer2-duh (l (after j l)) )
                 len-neuron-in-after
                 (:instance cdr-after2 (i j) (x l) )
                 (:instance app-isLayer (a (front j l)
                                           (b (cons n (after (+ j 1) l)) ) )
                 (:instance add-neurons-to-layer2 (i (len (car (after j l))) )
                                                    (l (after (+ j 1) l))
                                                    (n n) )
                 (:instance isLayer-after2 (j j) (l l) )
                 (:instance isLayer-front (i j) (l l) )
           :do-not-induct t
           :hands-off (isLayer
                       isNeuron) ) ) )

(defthmd replaceNeuronInLayerOfNetwork-front-after
  (implies (and (isNetwork net)
                (isNeuron n)
                (natp l)
                (natp j)
                (< l (len net))
                (< j (len (nth l net)))
                (equal (len n) (len (car (nth l net)))) )
            (equal (replaceNeuronInLayerOfNetwork j l n net)
                  (append (front l net)
                          (cons (replaceValInList j n (nth l net))
                                  (after (+ l 1) net))) ) )

  :hints (("Goal"
           :induct (replaceNeuronInLayerOfNetwork J L N NET) )
           ("Subgoal *1/2"
            :do-not-induct t)
           ("Subgoal *1/1"
            :do-not-induct t)) )

(defthmd add-layer-to-network2
  (implies (and (isLayer l)
                (isNetwork2 n (len l)) )
            (isNetwork (cons l n)) ) )

(defthmd len-replaceValInList

```

```

(implies (and (consp l)
              (< j (len l)) )
         (equal (len (replaceValInList j n l))
                (len l)) )

:hints ("Goal"
       :induct (replaceValInList j n l) ) )

(defthmd len-car-replaceValInList
  (implies (and (isLayer l)
                (isNeuron n)
                (natp j)
                (< j (len l))
                (equal (len n) (len (car l))) )
           (equal (len (car (replaceValInList j n l)))
                  (len (car l)) ) ) )

(defthmd isNetwork-replaceNeuronInLayerOfNetwork
  (implies (and (isNetwork net)
                (isNeuron n)
                (natp l)
                (natp j)
                (< l (len net))
                (< j (len (nth l net)))
                (equal (len n) (len (car (nth l net)))) )
           (isNetwork (replaceNeuronInLayerOfNetwork j l n net)) )

:hints ("Goal"
       :do-not '(generalize)
       :use ((:instance nth-layer-in-network-consistent (l (+ -1 L)) (n net) )
             come-on!3
             (:instance cdr-after2 (i l) (x net) )
             (:instance isNetwork-true-listp (n net) )
             (:instance nth-front (a (+ -1 l)) (b l) (n net) )
             (:instance front-len (i l) (w net) )
             (:instance len-car-replaceValInList (j j)
                                                  (n n)
                                                  (l (nth l net)) )
             (:instance front-isNetwork (l l) (n net) )
             (:instance nth-in-network-isLayer (l l) (n net) )
             (:instance isNetwork-app (a (front l net))
                                       (b (CONS (replaceValInList J N (NTH L NET))
                                               (after (+ l l) net)) ) )
             (:instance len-replaceValInList (j j) (n n) (l (nth l net)))
             (:instance add-layer-to-network2 (l (replaceValInList J N (nth l NET)))
                                               (n (after (+ l l) net)) )
             (:instance isLayer-replaceValInList (j j) (n n) (l (nth l net)) )
             (:instance after-isNetwork (l l) (n net) )
             (:instance nth-car-after (l l) (n net) )
             (:instance isNetwork-isNetwork2-duh (n (after l net)) )
             replaceNeuronInLayerOfNetwork-front-after
             (:instance isNetwork-isNetwork2-duh (n net) )
       :hands-off (isNetwork
                  isLayer
                  nth
                  front
                  isNeuron)
       :do-not-induct t)) )

(defthmd forwardLayer-replaceValInList
  (implies (and (isLayer l)
                (isListOfWeights i)
                (equal (len i) (len (car l)))
                (isNeuron n)
                (equal (len (car l)) (len n))
                (natp j)
                (< j (len l)) )
           (equal (forwardLayer f i (replaceValInList j n l))
                  (replaceValInList j (activate f i n) (forwardLayer f i l)) ) ) )

(defthmd nth-in-layer-is-neuron

```



```

                                (1 1) ))
:hands-off (isLayer
            isDeadNeuron2
            makeDeadNeuron
            replaceValInList
            isNeuron
            forwardLayer)
:do-not-induct t)
("Subgoal 2"
:hands-off (isLayer
            isDeadNeuron2
            makeDeadNeuron
            isNeuron
            forwardLayer) )
("Subgoal 1"
:use ( (:instance cdr-after2 (i j) (x 1) )
      split-after-helper
      (:instance isLayer-isLayer2-duh (1 (after j 1)) )
      (:instance nth-car-after (1 j) (n 1) )
      (:instance add-neurons-to-layer2 (i (len (car (after j 1))) )
                                         (1 (after (+ j 1) 1))
                                         (n (nth j 1)) ) ) ) )

(defthmd dead-with-dead-neuron-in-layer
  (implies (and (isLayer 1)
                (natp j)
                (< j (len 1))
                (isDeadNeuron (nth j 1)) )
            (equal (replaceValInList j (makeDeadNeuron (len (nth j 1))) 1)
                    L) )

  :hints (("Goal"
          :induct (nth j 1) )
          ("Subgoal *1/2"
          :use ( (:instance all-dead-neurons-the-same (i (len (car 1))) (n (nth j 1)) ) )
          :do-not-induct t)
          ("Subgoal *1/1"
          :do-not-induct t)) )

(defthmd isLayer-nth-neuron-same
  (implies (and (isLayer 1)
                (< j (len 1)) )
            (equal (len (car 1)) (len (nth j 1)) ) )

(defthmd general-len
  (implies (and (consp net)
                (consp (cdr net)) )
            (and (equal (len net) (+ 1 (len (cdr net)))) )
                 (equal (+ 1 (len (cdr net))) (+ 1 1 (len (caddr net)))) )
                 (equal (len net) (+ 1 1 (len (caddr net))) ) ) ) )

(defthmd bizarre
  (implies (and (natp 1)
                (true-listp net)
                (< 1 (len net))
                (after (+ 1 L) net) )
            (< 1 (- (len net) 1)) )

  :hints (("Goal"
          :use ( (:instance after-all-nil (n net) ) )
          :do-not-induct t)) )

(defthmd forwardNetwork-over-replaceNeuronInLayerOfNetwork
  (implies (and (isNetwork net)
                (isNeuron n)
                (isListOfWeights i)
                (equal (len i) (len (caar net)))
                (natp 1)
                (natp j)
                (< 1 (len net))
                (< j (len (nth 1 net)))

```

```

(equal (len n) (len (car (nth 1 net)))) )
(equal (forwardNetwork f i (replaceNeuronInLayerOfNetwork j 1 n net))
      (forwardNetwork f i (front 1 net))
      (cons (replaceValInList j n (nth 1 net))
            (after (+ 1 1) net))) )

:hints ("Goal"
       :cases ( (not (zp 1)) )
       :hands-off (isNetwork
                  isLayer
                  isNeuron
                  replaceNeuronInLayerOfNetwork
                  forwardLayer
                  forwardNetwork
                  activate)
       :use (replaceNeuronInLayerOfNetwork-front-after
            (:instance isNetwork-isNetwork2-duh (n net) )
            (:instance isLayer-isLayer2-duh (1 (car net)) ) )
       :do-not-induct t)
("Subgoal 2"
 :hands-off (isNetwork
            isLayer
            isNeuron
            replaceNeuronInLayerOfNetwork
            forwardLayer
            activate) )
("Subgoal 1"
 :cases ( (not (equal (after (+ 1 L) net) nil)) )
 :use (isNetwork-replaceNeuronInLayerOfNetwork
      (:instance car-front (i 1) (x net) )
      (:instance front-isNetwork (n net) )
      (:instance nth-in-network-isLayer (1 1) (n net) )
      (:instance isLayer-replaceValInList (1 (nth 1 net)) )
      (:instance forwardNetwork-over-front-after (n net) ) ) )
("Subgoal 1.2"
 :use ((:instance forwardNetwork-over-app (f f)
                                             (i i)
                                             (a (FRONT L Net))
                                             (b (CONS
                                                  (REPLACEVALINLIST
                                                    J N (NTH L NET))
                                                  (AFTER (+ 1 L) NET)) ) ) )
      (:instance one-layer-isNetwork (1 (REPLACEVALINLIST
                                          J N (NTH L NET)) ) ) ) )
("Subgoal 1.1"
 :cases ( (equal (len net) 1)
          (< 1 (len net)) )
 :do-not '(generalize fertilize)
 :hands-off (isNetwork
            isLayer
            nth
            front
            after
            isNeuron
            replaceNeuronInLayerOfNetwork
            forwardLayer
            forwardNetwork
            activate)
 :use ((:instance after-isNetwork (1 (+ 1 L)) (n net) )
      (:instance isNetwork-cdr (n net) )
      (:instance nth-car-after (1 (+ 1 L)) (n net) )
      (:instance isNetwork-isNetwork2-duh (n (cdr net)) )
      (:instance just-len-cdr (a net) )
      (:instance nth-car-after (1 (+ 1 L)) (n net) )
      (:instance isLayer-isLayer2-duh (1 (nth 1 net)) )
      (:instance len-replaceValInList (1 (nth 1 net)) )
      (:instance add-layer-to-network (1 (REPLACEVALINLIST J N (NTH L NET)) )
                                       (n (AFTER (+ 1 L) NET)) ) ) ) )
("Subgoal 1.1.1"
 :use (bizarre
      (:instance isNetwork-true-listp (n net) )

```

```

      (:instance nth-layer-in-network-consistent (l l) (n net) )) )
("Subgoal 1.1.1.1"
 :use (
      (:instance forwardNetwork-over-app (f f)
      (i i)
      (a (FRONT L Net))
      (b (CONS
          (REPLACEVALINLIST
            J N (NTH L NET))
          (AFTER (+ 1 L) NET)) ) ) ) ) )
(defthmd nth-replaceValInList
  (implies (and (natp j)
                (< j (len l)) )
            (equal (nth j (replaceValInList j n l)) n) ) )

```

prune.lisp

```

; Prune - removing neurons from networks
(in-package "ACL2")

```

```

(include-book "replace")

```

```

; removeFromList removes the neuron at index j from
; layer l.

```

```

(defun removeFromList (j l)
  (append (front j l)
          (after (+ j 1) l)))

```

```

; removeSynapsesFromLayer removes the synapse from
; position i of each neuron in layer l.

```

```

(defun removeSynapsesFromLayer (i l)
  (if (consp l)
      (cons (removeFromList i (car l))
            (removeSynapsesFromLayer i (cdr l)))
      nil))

```

```

; removeFromNetwork removes a neuron from index j of the
; the l-th layer of a network n.

```

```

(defun repairRemainder (i n)
  (if (consp n)
      (cons (removeSynapsesFromLayer i (car n))
            (cdr n))
      nil))

```

```

(defun removeFromNetwork (l j n)
  (append (front l n)
          (cons (removeFromList j (nth l n))
                (repairRemainder j (after (+ 1 l) n)))))

```

```

; The following are several uninteresting lemmas. Skip ahead
; to find interesting theorems.

```

```

(defthmd forwardNetwork-nil
  (equal (FORWARDNETWORK F I NIL) I) )

```

```

(defthmd zero-input-to-zero-output
  (implies (and (isDeadNeuron i)
                (isNeuron n)
                (equal (len i) (len n)) )
            (equal (forwardNeuron i n) 0) )

```

```

: hints ("Goal"
        : induct (forwardNeuron i n) ) )

(defthmd isListOfWeights-app
  (implies (and (isListOfWeights a)
                (isListOfWeights b))
            (isListOfWeights (append a b)) ) )

(defthmd forwardNeuron-extra-0
  (implies (and (consp n)
                (equal (len n) (+ 1 (len i))))
            (equal (forwardNeuron (cons 0 i) n)
                    (forwardNeuron i (cdr n)) ) ) )

(defthmd activate-extra-0
  (implies (and (consp n)
                (equal (len n) (+ 1 (len i))))
            (equal (activate f (cons 0 i) n)
                    (activate f i (cdr n)) ) )

: hints ("Goal"
        : use (forwardNeuron-extra-0) ) )

(defthmd add-remove-0
  (implies (and (isListOfWeights i)
                (isLayer l)
                (< 0 (len i))
                (equal (+ 1 (len i)) (len (car l))))
            (equal (FORWARDLAYER F (CONS 0 i) l)
                    (FORWARDLAYER F i (removeSynapsesFromLayer 0 l)) ) )

: hints ("Goal"
        : hands-off (forwardLayer
                     isLayer)
        : induct (removeSynapsesFromLayer 0 l) )
("Subgoal *1/2"
: use (isLayer-isLayer2-duh)
: do-not-induct t)
("Subgoal *1/1"
: in-theory (enable after-cdr)
: hands-off (isLayer
             activate)
: use (isLayer-<-2
      isLayer-isLayer2-duh
      (:instance activate-extra-0 (n (car l)) )
      (:instance isLayer-all-neurons-same (x l) )
      not-isLayer-cdr-nil)
: do-not-induct t) )

(defthmd front-append-front
  (implies (<= i (len x))
            (equal (front i (append (front i x) y))
                    (front i x) ) ) )

(defthmd after-0
  (equal (after 0 x) x))

(defthmd front-0
  (equal (front 0 x) nil))

(defthmd swap-replaceValInList
  (implies (and (natp j)
                (not (zp j))
                (consp i)
                (< j (len i)) )
            (equal (SWAP (REPLACEVALINLIST J 0 I) J 0)
                    (cons 0 (REPLACEVALINLIST (- J 1) (car i) (cdr I)) ) ) )

: hints ("Goal"
        : use (:instance after-append-front (i (+ -1 J))

```

```

(x (cdr i))
(y (CONS 0 (AFTER (+ -1 1 J) (CDR I)))) )
(:instance front-append-front (i (+ -1 J))
(x (cdr i))
(y (CONS 0 (AFTER (+ -1 1 J) (CDR I)))) )
(:instance replaceValInList-front-after (j (- j 1))
(n (car i))
(l (cdr i)) )
(:instance replaceValInList-front-after (j j)
(n 0)
(l i)) )

:do-not-induct t)) )

(defthmd isNeuron-cdr
  (implies (and (isNeuron n)
                (<= 2 (len n)) )
            (isNeuron (cdr n)) ) )

(defthmd obvious-ineq2
  (implies (and (< j (len i))
                (natp j))
            (< (- j 1) (len (cdr i))) ) )

(defthmd obvious-ineq3
  (implies (and (< j (len i))
                (natp j))
            (< j (+ 1 (len (cdr i)))) ) )

(defthmd little-after-cdr-helper
  (implies (and (natp j)
                (consp n)
                (not (equal j (len (cdr n))))
                (< j (len n)) )
            (CONSP (AFTER (+ -1 J) (CDR N))) )

  :hints (("Goal"
           :induct (after j n)) ) )

(defthmd not-both-ineq
  (implies (< a b)
            (not (< b a)) ) )

(defthmd not-because-zp
  (implies (and (not (zp j))
                (consp i)
                (< j (len i))
                (NOT (EQUAL J 1)) )
            (not (< (+ 1 (LEN (CDR i))) 2)) ) )

(defthmd not-isNeuron-not-cdr
  (implies (and (isNeuron n)
                (not (isNeuron (cdr n)))) )
            (not (cdr n)) ) )

(defthmd isListOfWeights-to-isNeuron
  (implies (and (isListOfWeights i)
                (consp i) )
            (isNeuron i) ) )

(defthmd equal-len-cdr
  (implies (equal (len i) (len n))
            (equal (len (cdr i)) (len (cdr n))) ) )

(defthmd app-nil
  (implies (true-listp x)
            (equal (append x nil) x) ) )

(defthmd one-synapse-isNeuron
  (implies (isSynapse s)
            (isNeuron (list s)) ) )

```

```

(defthmd forwardNeuron-rearrange
  (implies (and (isListOfWeights i)
                (<= 2 (len i))
                (not (zp j))
                (equal (len i) (len n))
                (isNeuron n)
                (< j (len i))
                (natp j) )
    (EQUAL (forwardNeuron
            (REPLACEVALINLIST (+ -1 J)
                              (CAR I)
                              (CDR I))
            (APPEND (FRONT (+ -1 J) (CDR n))
                    (CONS (CAR n)
                          (CDR (AFTER (+ -1 J) (CDR n))))))
          (forwardNeuron
            (CONS (CAR I)
                  (APPEND (FRONT (+ -1 J) (CDR I))
                          (AFTER (+ 1 J) I)))
            (CONS (CAR n)
                  (APPEND (FRONT (+ -1 J) (CDR n))
                          (AFTER (+ 1 J) n)))))) )

:hints ("Goal"
  :cases ( (equal j 1)
            (equal j (len (cdr i))) )
  :hands-off (isNeuron
              isSynapse)
  :use (isNeuron-true-listp
        isNeuron-cdr
        isNeuron-duh
        (:instance cdr-after2 (i (+ -1 J)) (x (cdr n)) )
        (:instance replaceValInList-front-after (j (+ -1 J))
          (n (car i))
          (l (cdr i)) ))
  :do-not-induct t )
("Subgoal 3"
  :use ((:instance not-both-ineq (a (+ -1 J)) (b (LEN (CDR n)))) )
        (:instance not-both-ineq (a (+ -1 J)) (b (LEN (CDR I)))) )
        (:instance front-len (i (+ -1 J)) (w (cdr i)) )
        (:instance front-len (i (+ -1 J)) (w (cdr n)) )
        obvious-ineq2
        obvious-ineq3
        not-because-zp
        (:instance front-subset (i (+ -1 J)) (w (cdr i)) )
        (:instance front-isNeuron (j (+ -1 J)) (n (cdr n)) )
        (:instance front-subset (i (+ -1 J)) (w (cdr n)) )
        (:instance front-isNeuron (j (+ -1 J)) (n (cdr n)) )) )
("Subgoal 3.1"
  :use ((:instance obvious-ineq2 (i n) )
        (:instance obvious-ineq3 (i n) )
        (:instance after-len (i (+ -1 j)) (w (cdr n)) )
        (:instance after-len (i (+ -1 1 j)) (w (cdr i)) )
        (:instance after-isNeuron (k (+ -1 J)) (n (cdr n)) )
        (:instance after-subset (i (+ -1 J)) (w (cdr n)) )
        (:instance after-subset (i (+ -1 1 J)) (w (cdr i)) )
        (:instance add-synapse-to-neuron (s (car i))
          (n (AFTER (+ -1 1 J) (CDR I)) ) )
        (:instance isNeuron-duh (n (AFTER (+ -1 J) (CDR N))) )
        (:instance add-synapse-to-neuron (s (car n))
          (n (CDR (AFTER (+ -1 J) (CDR N))) ) )
        (:instance forwardNeuron-over-app (a (FRONT (+ -1 J) (CDR I)) )
          (b (CONS (CAR I)
                  (AFTER (+ -1 1 J) (CDR I))))
          (c (FRONT (+ -1 J) (CDR N)) )
          (d (CONS (CAR N)
                  (CDR (AFTER (+ -1 J)
                              (CDR N)))))) ) )
("Subgoal 3.1.1"
  :use ((:instance after-isNeuron (k (+ -1 1 J)) (n (cdr i)) )

```

```

(:instance after-isNeuron (k (+ -1 J)) (n (cdr i)) )
(:instance isNeuron-duh (n i) )
isListOfWeights-to-isNeuron
isListOfWeights-true-listp
not-isNeuron-not-cdr
(:instance isNeuron-duh (n i) )
(:instance not-isNeuron-not-cdr (n i) )
(:instance not-isNeuron-not-cdr (n (AFTER (+ -1 J) (CDR N)) ) )
(:instance cdr-after2 (i (+ -1 j)) (x (cdr i)) )
(:instance forwardNeuron-over-app (a (FRONT (+ -1 J) (CDR I)) )
                                   (b (AFTER (+ -1 1 J) (CDR I)) )
                                   (c (FRONT (+ -1 J) (CDR N)) )
                                   (d (CDR (AFTER (+ -1 J)
                                                (CDR N)))) ) ) )

("Subgoal 1.1"
 :use ( (:instance cdr-after2 (i (+ -1 (len (cdr i)))) (x (cdr i)) )
        isListOfWeights-true-listp
        equal-len-cdr
        isNeuron-duh
        (:instance app-nil (x (FRONT (+ -1 (LEN (CDR I))) (CDR I))) )
        (:instance app-nil (x (FRONT (+ -1 (LEN (CDR I))) (CDR N))) )
        (:instance forwardNeuron-over-app (a (FRONT (+ -1 (LEN (CDR I)))
                                                    (CDR I)) )
                                           (b (LIST (CAR I)) )
                                           (c (FRONT (+ -1 (LEN (CDR I)))
                                                    (CDR N)) )
                                           (d (LIST (CAR N)) ) )
        (:instance one-synapse-isNeuron (s (car i)) )
        (:instance one-synapse-isNeuron (s (car n)) )
        (:instance front-len (i (+ -1 (len (cdr i)))) (w (cdr i)) )
        (:instance front-len (i (+ -1 (len (cdr i)))) (w (cdr n)) )
        (:instance front-subset (i (+ -1 (len (cdr i)))) (w (cdr n)) )
        (:instance front-subset (i (+ -1 (len (cdr i)))) (w (cdr i)) )
        (:instance after-all-nil (n (cdr i)) )
        (:instance after-all-nil (n (cdr n)) )
        (:instance just-len-cdr (a i) ) ) ) )

(defthmd activate-rearrange
  (implies (and (isListOfWeights i)
                (<= 2 (len i))
                (not (zp j))
                (equal (len i) (len n))
                (isNeuron n)
                (< j (len i))
                (natp j) )
            (EQUAL (ACTIVATE F
                    (REPLACEVALINLIST (+ -1 J)
                                       (CAR I)
                                       (CDR I))
                    (APPEND (FRONT (+ -1 J) (CDR n))
                             (CONS (CAR n)
                                   (CDR (AFTER (+ -1 J) (CDR n))))))
                (ACTIVATE F
                    (CONS (CAR I)
                          (APPEND (FRONT (+ -1 J) (CDR I))
                                   (AFTER (+ 1 J) I)))
                    (CONS (CAR n)
                          (APPEND (FRONT (+ -1 J) (CDR n))
                                   (AFTER (+ 1 J) n)))))) )

:hints ("Goal"
        :use (forwardNeuron-rearrange) ) )

(defthmd forwardLayer-over-removeSynapsesFromLayer-sub-3.1-helper
  (implies (and (isListOfWeights i)
                (<= 2 (len i))
                (not (zp j))
                (isLayer l)
                (equal (len i) (len (car l)))
                (< j (len i))
                (natp j) )
            )

```

```

(equal (FORWARDLAYER F
        (REPLACEVALINLIST (+ -1 J) (CAR I) (CDR I))
        (REMOVESYNAPSESFROMLAYER
         0 (SWAPNEURONSYNAPSESINLAYER L J 0)))
 (FORWARDLAYER F
  (APPEND (FRONT J I) (AFTER (+ 1 J) I))
  (REMOVESYNAPSESFROMLAYER J L) ) )

:hints ("Goal"
:induct (SWAPNEURONSYNAPSESINLAYER L J 0) )
("Subgoal *1/2"
:do-not-induct t)
("Subgoal *1/1"
:do-not '(generalize)
:hands-off (isLayer
            isListOfWeights
            activate)
:use (isLayer-isLayer2-duh
      not-isLayer-cdr-nil
      (:instance isLayer-all-neurons-same (x 1) )
      (:instance activate-rearrange (n (car 1)) )
      (:instance isNeuron-duh (n (car 1)) )
      (:instance car-front (i j) (x (car 1)) ))
:do-not-induct t))

(defthmd forwardNeuron-rearrange2
  (implies (and (isListOfWeights i)
                (<= 2 (len i))
                (not (zp j))
                (equal (len i) (len (car 1)))
                (isNeuron (car 1))
                (< j (len i))
                (natp j) )
            (equal (forwardNeuron
                    (CONS (CAR I)
                          (APPEND (FRONT (+ -1 J) (CDR I))
                                   (AFTER (+ -1 1 J) (CDR I))))
                    (CONS (CAAR L)
                          (APPEND (FRONT (+ -1 J) (CDAR L))
                                   (CDR (AFTER (+ -1 J) (CDAR L))))))
            (forwardNeuron
             (CONS (CAR I)
                   (REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
             (CAR L) ) ) )

:hints ("Goal"
:cases ( (equal j 1)
         (equal j (len (cdr i))) )
:do-not '(generalize)
:use (isListOfWeights-true-listp
      (:instance isNeuron-true-listp (n (car 1)) )
      (:instance front-len (i (+ -1 J)) (w (cdar 1)) )
      (:instance front-len (i (+ -1 J)) (w (cdr i)) )
      (:instance replaceValInList-front-after (j j)
                                               (n 0)
                                               (l i) ))
:do-not-induct t)
("Subgoal 3.1"
:use ((:instance len-replaceValInList (j (+ -1 J)) (n 0) (l (cdr i)) )
      (:instance after-len (i (+ -1 j)) (w (cdar 1)) )
      (:instance after-len (i (+ -1 1 j)) (w (cdr i)) )
      (:instance cdr-after2 (i (+ -1 J)) (x (cdr i)) ) )
("Subgoal 3.1.1"
:use ((:instance front-subset (i (+ -1 J)) (w (cdr i)) )
      (:instance after-subset (i (+ -1 J)) (w (cdr i)) )
      (:instance front-isNeuron (j (+ -1 J)) (n (cdar 1)) )
      (:instance after-isNeuron (k (+ -1 J)) (n (cdar 1)) )
      (:instance forwardNeuron-over-app (a (FRONT (+ -1 J) (CDR I)) )
                                         (b (CONS
                                              0
                                              (CDR (AFTER (+ -1 J) (CDR I)))) ) ) )

```

```

(c (FRONT (+ -1 J) (CDaR l)) )
(d (AFTER (+ -1 J) (CDaR l)) ) )
(:instance forwardNeuron-over-app
(a (FRONT (+ -1 J) (CDR I)) )
(b (cdr (AFTER (+ -1 J) (CDR I)))) )
(c (FRONT (+ -1 J) (CDaR l)) )
(d (CDR (AFTER (+ -1 J)
(CDaR l))) ) ) )
("Subgoal 3.1.1.1"
:use ( (:instance front-after-components (j (+ -1 j)) (n (cdar l)) ) )
"Subgoal 1.1"
:use ( (:instance len-replaceValInList (j (+ -1 (LEN (CDR I)))) (n 0)
(l (cdr i)) )
(:instance after-len (i (+ -1 (LEN (CDR I)))) (w (cdar l)) )
(:instance after-len (i (+ -1 1 (LEN (CDR I)))) (w (cdr i)) )
(:instance cdr-after2 (i (+ -1 (LEN (CDR I)))) (x (cdr i)) ) ) )
("Subgoal 1.1.1"
:in-theory (enable app-nil)
:use ( (:instance after-all-nil (n (cdr i)) )
(:instance after-all-nil (n (cdar l)) )
(:instance equal-len-cdr (n (car l)) )
(:instance just-len-cdr (a i) )
(:instance front-isNeuron (j (+ -1 (len (cdr i)))) (n (cdar l)) )
(:instance front-subset (i (+ -1 (len (cdr i)))) (w (cdr i)) )
(:instance after-isNeuron (k (+ -1 (len (cdr i)))) (n (cdar l)) )
(:instance after-subset (i (+ -1 (len (cdr i)))) (w (cdr i)) )
(:instance forwardNeuron-over-app (a (FRONT (+ -1 (LEN (CDR I)))
(CDR I)) )
(b (cons 0 nil) )
(c (FRONT (+ -1 (LEN (CDR I)))
(CDaR l)) )
(d (AFTER (+ -1 (LEN (CDR I)))
(CDaR l)) ) ) )
(:instance front-after-components (j (+ -1 (LEN (CDR I)))
(n (cdar l)) )
(:instance cdr-after2 (i (+ -1 (LEN (CDR I)))) (x (cdr i)) )
(:instance cdr-after2 (i (+ -1 (LEN (CDR I)))) (x (cdar l)) ) ) ) )
(defthmd activate-rearrange2
(implies (and (isListOfWeights i)
(<= 2 (len i))
(not (zp j))
(equal (len i) (len (car l)))
(isNeuron (car l))
(< j (len i))
(natp j) )
(equal (ACTIVATE F
(CONS (CAR I)
(APPEND (FRONT (+ -1 J) (CDR I))
(AFTER (+ -1 1 J) (CDR I))))
(CONS (CAAR L)
(APPEND (FRONT (+ -1 J) (CDAR L))
(CDR (AFTER (+ -1 J) (CDAR L)))))
(ACTIVATE F
(CONS (CAR I)
(REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
(CAR L)) ) )
:hints ( ("Goal"
:use (forwardNeuron-rearrange2)
:do-not-induct t) ) )
(defthmd forwardLayer-over-removeSynapsesFromLayer-sub-1.2
(IMPLIES
(AND (EQUAL (SWAP (CONS (CAR I)
(REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
J 0)
(CONS 0
(REPLACEVALINLIST (+ -1 J)
(CAR I)
(CDR I))))
(EQUAL (REPLACEVALINLIST (+ -1 J) 0 (CDR I))

```

```

      (APPEND (FRONT (+ -1 J) (CDR I))
              (CONS 0 (AFTER (+ -1 1 J) (CDR I))))))
(EQUAL (REPLACEVALINLIST (+ -1 J)
                          (CAR I)
                          (CDR I))
      (APPEND (FRONT (+ -1 J) (CDR I))
              (CONS (CAR I)
                    (AFTER (+ -1 1 J) (CDR I))))))
(CONSP (CAR L))
(RATIONALP (CAAR L))
(ISLISTOFWEIGHTS (CDAR L))
(TRUE-LISTP (CDAR L))
(EQUAL (CDR (AFTER (+ -1 J) (CDAR L)))
      (AFTER (+ -1 1 J) (CDAR L)))
(EQUAL (ACTIVATE F
          (REPLACEVALINLIST (+ -1 J)
                            (CAR I)
                            (CDR I))
        (APPEND (FRONT (+ -1 J) (CDAR L))
                (CONS (CAAR L)
                      (CDR (AFTER (+ -1 J) (CDAR L)))))))
      (ACTIVATE F
        (CONS (CAR I)
              (APPEND (FRONT (+ -1 J) (CDR I))
                      (AFTER (+ -1 1 J) (CDR I))))
        (CONS (CAAR L)
              (APPEND (FRONT (+ -1 J) (CDAR L))
                      (CDR (AFTER (+ -1 J) (CDAR L)))))))
(ISNEURON (CAR L))
(RATIONALP (CAR I))
(ISLISTOFWEIGHTS (FRONT (+ -1 J) (CDR I)))
(ISLISTOFWEIGHTS (AFTER (+ -1 1 J) (CDR I)))
(EQUAL (+ 1 (LEN (FRONT (+ -1 J) (CDR I)))
        J)
      (CONSP I))
(EQUAL (LEN (AFTER (+ -1 1 J) (CDR I)))
      (+ (- (+ 1 J)) 1 (LEN (CDR I))))
(ISLISTOFWEIGHTS (REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
(CONSP L)
(EQUAL (ACTIVATE F
        (CONS (CAR I)
              (REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
        (CAR L))
      (ACTIVATE F
        (CONS 0
              (REPLACEVALINLIST (+ -1 J)
                                (CAR I)
                                (CDR I)))
              (SWAP (CAR L) J 0)))
      (ISLAYER L)
      (<= 2 (+ 1 (LEN (CDR I))))
      (ISLISTOFWEIGHTS (CDR I))
      (<= 0 J)
      (< J (+ 1 (LEN (CDR I))))
      (EQUAL (+ 1 (LEN (CDR I)))
            (+ 1 (LEN (CDAR L)))))
      (EQUAL (ACTIVATE F
              (REPLACEVALINLIST (+ -1 J)
                                (CAR I)
                                (CDR I))
            (APPEND (FRONT (+ -1 J) (CDAR L))
                    (CONS (CAAR L)
                          (CDR (AFTER (+ -1 J) (CDAR L)))))))
            (ACTIVATE F
              (CONS (CAR I)
                    (REPLACEVALINLIST (+ -1 J) 0 (CDR I)))
              (CAR L))))))
:hints (("Goal"
        :hands-off (activate)
        :use (activate-rearrange2

```

```

      (:instance cdr-after2 (i (+ -1 J)) (x (cdr i)) )
      :do-not-induct t)) )

(defthmd forwardLayer-over-removeSynapsesFromLayer
  (implies (and (isLayer l)
                (<= 2 (len i))
                (isListOfWeights i)
                (natp j)
                (< j (len i))
                (equal (len i) (len (car l)))) )
    (equal (forwardLayer f (removeFromList j i)
                    (removeSynapsesFromLayer j l))
      (forwardLayer f (replaceValInList j 0 i) l)) )

:hints ("Goal"
  :hands-off (forwardLayer
    removeSynapsesFromLayer
    swapNeuronSynapsesInLayer
    activate
    swap
    isLayer
    isNeuron)
  :do-not '(generalize)
  :use (not-isLayer-cdr-nil
    isLayer-isLayer2-duh
    (:instance front-subset (i j) (w i) )
    (:instance after-subset (i (+ 1 j)) (w i) )
    (:instance front-len (i j) (w i) )
    (:instance after-len (i (+ 1 j)) (w i) )
    (:instance len-append (a (front j i))
      (b (cons 0 (after (+ j 1) i)))) )
    (:instance isListOfWeights-app (a (front j i))
      (b (cons 0 (after (+ j 1) i)))) )
    (:instance synapse-swap-in-neurons-of-layer (f f)
      (i (APPEND
        (FRONT J I)
        (CONS
          0
          (AFTER (+ 1 J) I)))) )
      (l l)
      (j j)
      (k 0) )
    (:instance replaceValInList-front-after (j j)
      (n 0)
      (l i) ))

  :do-not-induct t)
("Subgoal 3"
  :hands-off (forwardLayer
    removeSynapsesFromLayer
    swapNeuronSynapsesInLayer
    activate
    swap
    front
    isLayer
    isNeuron)
  :use ((:instance front-0 (x i) )
    (:instance swap-same (i 0) (x (CONS 0 (CDR I)))) )
    (:instance add-remove-0 (i (cdr i)) )
    (:instance swapNeuronSynapsesInLayer-same (j 0) (l l) )
    swap-replaceValInList) )
("Subgoal 3.1"
  :hands-off (forwardLayer
    removeSynapsesFromLayer
    swapNeuronSynapsesInLayer
    activate
    swap
    front
    after
    isLayer
    isNeuron)
  :use ((:instance len-replaceValInList (j (+ -1 J)) (n (car i)) (l (cdr i)) )

```

```

      (:instance add-remove-0 (i (REPLACEVALINLIST (+ -1 J) (CAR I) (CDR I)))
        (l (SWAPNEURONSYNAPSESINLAYER L J 0)))
      (:instance front-subset (i (+ -1 j)) (w (cdr i)))
      (:instance after-subset (i (+ -1 J 1)) (w (cdr i)))
      (:instance replaceValInList-front-after (j (+ -1 J))
        (n (car i))
        (l (cdr i)))
      (:instance same-neuron-len-after-swapNeuronSynapsesInLayer
        (l 1) (j j) (k 0))
      (:instance swapNeuronSynapsesInLayer-isLayer (l 1) (j j) (k 0))
      (:instance isListOfWeights-app (a (front (+ -1 J) (cdr i)))
        (b (cons (car i) (after (+ -1 J 1)
          (cdr i)))))
      forwardLayer-over-removeSynapsesFromLayer-sub-3.1-helper)
("Subgoal 1"
 :expand ( (REMOVESYNAPSESFROMLAYER J L) )
 :in-theory (enable len-append)
 :use (swap-replaceValInList
      (:instance cdr-after (x (car l)))
      (:instance replaceValInList-front-after (j j)
        (n 0)
        (l i))
      (:instance replaceValInList-front-after (j (+ -1 J))
        (n (car i))
        (l (cdr i)))
      (:instance activate-extra-0 (i (cdr i)) (n (car l)))
      (:instance isNeuron-duh (n (car l)))
      (:instance isNeuron-true-listp (n (car l)))
      (:instance swap-same (i 0) (x (car l)))
      (:instance cdr-after2 (i (+ -1 J)) (x (cdar l)))
      (:instance activate-rearrange (n (car l)))
      (:instance swap-same (i 0) (x (CONS 0 (CDR I))))
 :hands-off (activate
      swap
      isLayer
      isNeuron) )
("Subgoal 1.2'"
 :use (forwardLayer-over-removeSynapsesFromLayer-sub-1.2) ))

(defthmd forwardNetwork-over-repairRemainder
  (implies (and (isNetwork n)
    (<= 2 (len i))
    (isListOfWeights i)
    (equal (len i) (len (caar n)))
    (natp j)
    (< j (len i)))
    (equal (forwardNetwork f (removeFromList j i)
      (repairRemainder j n))
      (forwardNetwork f (replaceValInList j 0 i) n) ))

 :hints (("Goal"
 :hands-off (activate
      removeSynapsesFromLayer
      removeFromList
      replaceValInList
      isLayer
      forwardLayer)
 :in-theory (enable forwardLayer-over-removeSynapsesFromLayer)
 :expand ( (FORWARDNETWORK F (REPLACEVALINLIST J 0 I) N) )
 :use ((:instance forwardLayer-over-removeSynapsesFromLayer (l (car n)) )
      (:instance isLayer-isLayer2-duh (l (car n)) ))
 :do-not-induct t)) )

(defthmd consp-front
  (iff (consp (front l n))
    (front l n) ))

(defthmd front-0-arg
  (implies (consp n)
    (iff (front i n)
      (not (zp i)))))

```

```

(defthmd little-after-len-help
  (implies (and (NOT (EQUAL J (+ -1 (LEN L))))
                (natp j)
                (< j (len l)) )
            (< (+ 1 j) (len l)) ) )

(defthmd isLayer-removeFromList
  (implies (and (isLayer l)
                (<= 2 (len l))
                (natp j)
                (< j (len l)) )
            (isLayer (removeFromList j l)) )

  :hints (("Goal"
           :hands-off (isLayer)
           :cases ( (equal j 0)
                    (equal j (- (len l) 1)) )
           :use ( (:instance front-len (i j) (w l) )
                  (:instance isLayer-front (i j) (l l) )
                  (:instance isLayer-after2 (j (+ 1 j)) (l l) )
                  (:instance after-len (i (+ 1 j)) (w l) )
                  same-neuron-len-in-front
                  (:instance same-neuron-len-in-after (j (+ 1 j)) )
                  (:instance app-isLayer (a (front j l)) (b (after (+ 1 j) l)) ) )
           :do-not-induct t)
          ("Subgoal 3"
           :use (little-after-len-help) )
          ("Subgoal 1"
           :use (isLayer-true-listp
                  (:instance app-nil (x (FRONT (LEN (CDR L)) L)) )
                  (:instance after-all-nil (n (cdr l)) )
                  (:instance just-len-cdr (a l) ) ) ) ) )

(defthmd specific-nth-len
  (implies (< 0 (len (cdr n)))
            (equal (NTH (+ -1 (LEN (CDR N))) (CDR N))
                   (NTH (LEN (CDR N)) N) ) ) )

(defthmd do-math-nth
  (equal (+ -1 -1 1 -1 (LEN (CDR N)))
          (+ -1 -1 (LEN (CDR N)) ) ) )

(defthmd obvious-nth-cdr
  (implies (and (not (zp l))
                (<= 2 (len (nth l n)) )
                (not (< (LEN (NTH (+ -1 L) (CDR N))) 2) ) ) )

(defthmd len-car-removeSynapsesFromLayer
  (implies (and (natp j)
                (consp l)
                (< j (len (car l)))
                (<= 2 (len (car l)) )
                (equal (len (car (removeSynapsesFromLayer j l)))
                       (- (len (car l)) 1) ) )

  :hints (("Goal"
           :in-theory (enable front-len
                               after-len
                               len-append)
           :expand ( (REMOVESYNAPSESFROMLAYER J L) )
           :do-not-induct t)) )

(defthmd not-isNeuron-after
  (implies (and (isNeuron n)
                (natp i)
                (<= i (len n))
                (not (isNeuron (after i n))) )
            (and (not (after i n))
                 (equal (len n) i)) ) )

```

```

:hints ("Goal"
       :use (isNeuron-true-listp) )) )

(defthmd not-removeSynapsesFromLayer
  (implies (not l)
    (not (removeSynapsesFromLayer j l)) ))

(defthmd isLayer-removeSynapsesFromLayer
  (implies (and (isLayer l)
    (natp j)
    (< j (len (car l)))
    (<= 2 (len (car l))))
    (isLayer (removeSynapsesFromLayer j l)) ))

:hints ("Goal"
       :induct (removeSynapsesFromLayer j l) )
("Subgoal *1/2"
 :do-not-induct t)
("Subgoal *1/1"
 :cases ( (not (cdr l)) )
 :in-theory (enable app-nil
  len-append
  front-len
  after-len
  len-0)
 :hands-off (isNeuron
  isLayer)
 :expand ( (REMOVESYNAPSESFROMLAYER J L)
  (REMOVESYNAPSESFROMLAYER j (CDR L)) )
 :do-not '(generalize)
 :use (isLayer-<-2
  isLayer-isLayer2-duh
  isLayer-true-listp
  (:instance add-neurons-to-layer (n (APPEND (FRONT J (CAR L))
    (AFTER (+ 1 J) (CAR L)))) )
    (l (REMOVESYNAPSESFROMLAYER
      j (CDR L)) ) )
  (:instance not-isNeuron-after (i (+ 1 j)) (n (car l)) )
  (:instance app-isNeuron (a (FRONT J (CAR L)) )
    (b (AFTER (+ 1 J) (CAR L)) ) )
  (:instance len-car-removeSynapsesFromLayer (l (cdr l)) )
  (:instance isLayer-all-neurons-same (x l) )
  (:instance one-neuron-isLayer (n (cdar l)) )
  (:instance front-isNeuron (j j) (n (car l)) )
  (:instance after-isNeuron (k (+ 1 j)) (n (car l)) )
  (:instance isNeuron-duh (n (car l)) ))
 :do-not-induct t)
("Subgoal *1/1.1"
 :use ((:instance one-neuron-isLayer (n (APPEND (FRONT J (CAR L))
  (AFTER (+ 1 J) (CAR L)))) )
  (:instance cdr-isNeuron (n (car l)) )) )) )

(defthmd len-removeSynapsesFromLayer
  (equal (len (removeSynapsesFromLayer j l))
    (len l) ))

(defthmd isNetwork-repairRemainder
  (implies (and (isNetwork n)
    (<= 2 (len (caar n)))
    (< j (len (caar n)))
    (natp j) )
    (isNetwork (repairRemainder j n)) ))

:hints ("Goal"
       :use (isNetwork-isNetwork2-duh
  isNetwork-cdr
  not-cdr-isNetwork
  (:instance isLayer-isLayer2-duh (l (car n)) )
  (:instance add-layer-to-network (l (REMOVESYNAPSESFROMLAYER J (CAR N)))
    (n (cdr n)) )
  (:instance one-layer-isNetwork (l (REMOVESYNAPSESFROMLAYER J (CAR N))))))

```



```

      (:instance front-after-components (j l) (n n) ))
:cases ( (equal 1 (- (len n) 1))
        (not (equal 1 (- (len n) 1)))) )
:do-not-induct t )
"Subgoal 2"
:use (consp-front
      (:instance front-isNetwork (l (len (cdr n))) (n n) )
      (:instance front-0-arg (i l) )
      (:instance one-layer-isNetwork (l (APPEND (FRONT J (NTH l N))
                                                (AFTER (+ 1 J)
                                                    (NTH l N)))) ) )

      (:instance after-all-nil (n (cdr n)) )
      (:instance isNetwork-app (a (FRONT (LEN (CDR N)) N) )
                                (b (LIST (APPEND (FRONT J
                                                  (NTH (LEN (CDR N)) N))
                                                  (AFTER (+ 1 J)
                                                      (NTH (LEN (CDR N))
                                                            N)))) ) )

      (:instance car-front (i (LEN (CDR N))) (x n) ))
:hands-off (activate
            isNetwork
            isLayer
            replaceValInList) )
"Subgoal 2.2"
:use ((:instance front-len (i (len (cdr n))) (w n) )
      specific-nth-len
      (:instance nth-front (a (+ -1 (LEN (CDR N))) ) (b (LEN (CDR N))) (n n))
      (:instance car-front (i j) (x (NTH (LEN (CDR N)) N)) ))
:in-theory (enable car-append
                  car-front) )
"Subgoal 2.2.1"
:use ((:instance car-append (a (FRONT J (NTH (+ -1 (LEN (CDR N))) (CDR N))) )
                             (b (AFTER (+ 1 J)
                                         (NTH (+ -1 (LEN (CDR N))) (CDR N)))) ) )
      (:instance car-front (i j) (x (NTH (- (LEN (CDR N)) 1) (cdr N))) )) )
"Subgoal 2.1"
:expand ( (NTH (+ -1 1 -1 (LEN (CDR N)))
              (CONS (CAR N)
                    (FRONT (+ -1 (LEN (CDR N))) (CDR N))))
        (AFTER 1
              (NTH (+ -1 (LEN (CDR N))) (CDR N))) )
:use ((:instance car-front (i j) (x (NTH (- (LEN (CDR N)) 1) (cdr N))) )
      do-math-nth
      (:instance nth-front (a (+ -1 -1 1 -1 (LEN (CDR N))) )
                            (b (+ -1 (LEN (CDR N))) ) (n (cdr n)) )
      (:instance car-append (a (FRONT J (NTH (+ -1 (LEN (CDR N))) (CDR N))) )
                             (b (AFTER (+ 1 J)
                                         (NTH (+ -1 (LEN (CDR N))
                                                (CDR N)))) ) ) ) )
"Subgoal 1"
:cases ( (zp l) )
:hands-off (repairRemainder
            isLayer
            isNetwork
            replaceValInList)
:do-not '(generalize fertilize)
:use (bleh!
      only-1
      front-isNetwork
      (:instance after-isNetwork (l (+ 1 l)) (n n) )) )
"Subgoal 1.2.12"
:use ((:instance nth-car-after (l (+ 1 L)) (n n) )
      (:instance len-car-removeSynapsesFromLayer (j j) (l (NTH (+ 1 L) N)) )
      (:instance consp-repairRemainder (n (after (+ 1 L) n)) )
      (:instance len-append (a (front j (nth l n)) )
                             (b (after (+ 1 j) (nth l n)) ) )
      (:instance nth-layer-in-network-consistent (l l) (n n) )
      (:instance isNetwork-repairRemainder (n (after (+ 1 L) n)) )
      (:instance add-layer-to-network
        (l (APPEND (FRONT J (NTH (+ -1 L) (CDR N)))
                  (AFTER (+ 1 J) (NTH (+ -1 L) (CDR N)))) )

```



```

                                (REPAIRREMAINDER
                                  J (AFTER (+ 1 L) N))) ) )
      (:instance isNetwork-repairRemainder (n (after (+ 1 L) n)) )
      (:instance add-layer-to-network (l (APPEND
        (FRONT J (NTH (+ -1 L) (CDR N)))
        (AFTER (+ 1 J)
          (NTH (+ -1 L) (CDR N))))))
      (n (REPAIRREMAINDER
        J (AFTER (+ 1 L) N)) ) ) ) ) )

(defthmd more-bleh!
  (implies (< 1 (- (len n) 1))
    (< 1 (len n)) ) )

(defthmd peel-from-after
  (implies (and (natp l)
    (< 1 (len n)) )
    (equal (AFTER L N)
      (cons (nth l n)
        (after (+ 1 L) n)) ) ) )

:hints ("Goal"
  :induct (after 1 n) )
("Subgoal *1/2"
  :use (argh!)
  :expand ( (AFTER (+ 1 L) N) )
  :do-not-induct t)
("Subgoal *1/1"
  :expand ( (AFTER 1 N) )
  :do-not-induct t)) )

(defthmd forwardLayer-over-removeFromList
  (implies (and (isLayer l)
    (isListOfWeights i)
    (equal (len i) (len (car l)) )
    (<= 2 (len l))
    (< j (len l))
    (natp j) )
    (equal (forwardLayer f i (removeFromList j l))
      (removeFromList j (forwardLayer f i l)) ) ) )

:hints ("Goal"
  :hands-off (activate
    forwardLayer
    isLayer)
  :use ((:instance isLayer-front (i j) (l l) )
    (:instance isLayer-after2 (j (+ 1 j)) (l l) )
    isLayer-isLayer2-duh)
  :in-theory (enable forwardLayer-over-app)
  :do-not-induct t)
("Subgoal 5"
  :hands-off (activate
    isLayer) )
("Subgoal 4"
  :cases ( (equal (+ 1 J) (LEN L)) )
  :in-theory (enable app-nil)
  :use (same-neuron-len-in-front
    (:instance same-neuron-len-in-after (j (+ 1 j)) )
    isLayer-true-listp
    forwardLayer-len
    forwardLayer-true-listp
    (:instance after-all-nil (n (forwardLayer f i l)) )
    (:instance after-all-nil (n L) )
    (:instance forwardLayer-over-front (f f) (i i) (k j) (l l) )
    (:instance forwardLayer-over-after (l l) (i i) (f f) (j (+ 1 j)) )
    (:instance forwardLayer-over-app (a (front j l) )
      (b (after (+ 1 j) l) )
      (i i)
      (f f) ))
  :hands-off (activate
    forwardLayer

```

```

        isLayer) )
("Subgoal 3"
 :cases ( (equal (+ 1 J) (LEN L)) )
 :in-theory (enable app-nil)
 :use (same-neuron-len-in-front
       (:instance same-neuron-len-in-after (j (+ 1 j)) )
       isLayer-true-listp
       forwardLayer-len
       forwardLayer-true-listp
       (:instance after-all-nil (n (forwardLayer f i l)) )
       (:instance after-all-nil (n L) )
       (:instance forwardLayer-over-front (f f) (i i) (k j) (l l) )
       (:instance forwardLayer-over-after (l l) (i i) (f f) (j (+ 1 j)) )
       (:instance forwardLayer-over-app (a (front j l)
                                           (b (after (+ 1 j) l)
                                               (i i)
                                               (f f) ))
                                           )
       :hands-off (activate
                   forwardLayer
                   isLayer) )) )

(defthmd replaceValInList-zero-identity
  (implies (equal (nth j i) 0)
           (equal (replaceValInList j 0 i) i) ) )

(defthmd nth-forwardLayer-dead-neuron-is-zero
  (implies (and (isDeadNeuron (nth j l))
                (natp j)
                (< j (len l))
                (isLayer l)
                (equal (len i) (len (car l)))
                (isListOfWeights i) )
           (equal (nth j (forwardLayer f i l)) 0) )

:hints (("Goal"
 :do-not '(generalize)
 :hands-off (activate
             isDeadNeuron)
 :induct (nth j l) )
 "Subgoal *1/2"
 :use ((:instance dead-neuron-output (n (nth j l)) )
       (:instance isDeadNeuron-duh (n (nth j l)) )
 :do-not-induct t)
 "Subgoal *1/1"
 :do-not-induct t)) )

(defthmd consp-front2
  (implies (and (consp l)
                (not (zp j)) )
           (consp (front j l)) ) )

(defthmd len-car-removeFromList
  (implies (and (isLayer l)
                (<= 2 (len l))
                (< j (len l))
                (natp j) )
           (equal (len (car (removeFromList j l)))
                 (len (car l)) ) )

:hints (("Goal"
 :in-theory (enable car-append
                   car-front)
 :do-not-induct t
 :cases ( (zp j) ) )
 "Subgoal 2"
 :use ((:instance car-front (i j) (x l) )
       consp-front2
       (:instance car-append (a (front j l)
                                 (b (after (+ 1 j) l)) )) ) ) )

(defthmd mensch!!!
  (equal (+ -1 1 -1 L)

```

```

(+ -1 L) )

(defthmd specific-nth-bleh
  (implies (and (consp n)
                (< 0 l)
                (natp l)
                (< l (len n)) )
            (equal (NTH (+ 1 -1 L) N)
                  (NTH (+ -1 L) (cdr N)) ) )

  :hints (("Goal"
           :use (mensch!!!)
           :do-not-induct t
           :expand ( (NTH (+ 1 -1 L) N) ) ) ) )

(defthmd len-removeFromList
  (implies (and (natp j)
                (< j (len l)) )
            (equal (len (removeFromList j l))
                  (- (len l) 1)) )

  :hints (("Goal"
           :in-theory (enable front-len
                               after-len
                               len-append)
           :do-not-induct t)) )

(defthmd very-bleh!
  (implies (and (consp n)
                (< L (+ -1 1 (LEN (CDR N)))) )
            (< L (- (LEN N) 1)) ) )

(defthmd impossible!
  (implies (< L (+ 1 (LEN (CDR N))))
            (not (< (+ 1 (LEN (CDR N))) L)) ) )

(defthmd isNeuron-not-zero
  (implies (isNeuron n)
            (< 0 (len n)) ) )

(defthmd isDead-in-nth-spec
  (implies (and
            (isNeuron (NTH J (nth l N)) )
            (EQUAL
             (NTH J
              (REPLACEVALINLIST J
               (MAKEDEADNEURON (LEN (NTH J (nth l N))))
               (CAR N)))
             (MAKEDEADNEURON (LEN (NTH J (nth l N)))) ) )
            (isDeadNeuron
             (NTH J
              (REPLACEVALINLIST J
               (MAKEDEADNEURON (LEN (NTH J (nth l N))))
               (CAR N))) ) )

  :hints (("Goal"
           :use ( (:instance isNeuron-not-zero (n (nth j (nth l n)) ) )
                  (:instance makeDeadNeuron-isDeadNeuron (i (len (nth j (nth l n))) ) ) )
           :do-not-induct t)) )

(defthmd len-aaaaaaaaaaaaaaaaaargh!
  (implies (< 0 (LEN I))
            (not (zp (len i)) ) )

(defthmd double-aaaaaaaaaaaaaaaaaargh!
  (implies (< 0 (len i))
            (ISNEURON (MAKEDEADNEURON (LEN I)) )

  :hints (("Goal"
           :do-not-induct t
           :use ( (:instance makeDeadNeuron-isDeadNeuron (i (len i)) ) )

```

```

len-aaaaaaaaaaaaaaaaaargh!)) )

(defthmd after-isLayer-or-nil
  (implies (and (isLayer l)
                (natp j)
                (<= j (len l))
                (not (isLayer (after j l)))) )
    (not (after j l)) )

  :hints (("Goal"
    :use (isLayer-true-listp) )) )

(defthmd consp-forwardLayer
  (implies (consp l)
    (consp (forwardLayer f i l)) ) )

(defthmd one-step-forwardLayer
  (implies (consp l)
    (equal (forwardLayer f i l)
      (cons (activate f i (car l))
        (forwardLayer f i (cdr l)))) ) ) )

(defthmd forwardLayer-nil
  (equal (forwardLayer f i nil) nil))

(defthmd do-it!
  (equal (CONS 0 (FORWARDLAYER F I NIL))
    (cons 0 nil) ) )

(defthmd zero-forwardLayer-from-dead-neuron
  (implies (and (isLayer l)
                (natp j)
                (< j (len l))
                (equal (len i) (len (car l)))
                (isListOfWeights i) )
    (equal (REPLACEVALINLIST J 0 (FORWARDLAYER F I L))
      (FORWARDLAYER F I
        (REPLACEVALINLIST J
          (MAKEDEADNEURON (len (nth j l))
            L)) ) )

  :hints (("Goal"
    :in-theory (enable forwardLayer-over-app
      makeDeadNeuron-len
      makeDeadNeuron-isDeadNeuron
      replaceValInList-front-after)
    :use ((:instance isLayer-front (i j) )
      isLayer-nth-neuron-same
      nth-in-layer-is-neuron
      isLayer-true-listp
      (:instance isNeuron-duh (n (nth j l) ) )
      (:instance isNeuron-not-zero (n (nth j l) ) )
      (:instance dead-neuron-output (n (MAKEDEADNEURON (len (nth j l)))) ) )
      (:instance isLayer-after2 (j (+ 1 j)) ) )
    :hands-off (activate
      isLayer
      forwardLayer
      isNeuron)
    :do-not-induct t)
  ("Subgoal 4"
    :hands-off (activate
      isNeuron) )
  ("Subgoal 3"
    :hands-off (activate
      isNeuron) )
  ("Subgoal 2"
    :cases ( (< (+ 1 J) (LEN L))
      (equal (+ 1 J) (LEN L)) )
    :use ((:instance same-neuron-len-in-after (j (+ 1 j)) )
      (:instance after-isLayer-or-nil (j (+ 1 j)) )
      len-aaaaaaaaaaaaaaaaaargh!

```

```

double-aaaaaaaaaargh!
(:instance add-neurons-to-layer (n (MAKEDEADNEURON (LEN I)) )
      (l (after (+ 1 j) l)) )
(:instance isLayer-front (i (len (cdr l))) )
(:instance makeDeadNeuron-isDeadNeuron (i (len i)) )
(:instance forwardLayer-over-app (a (FRONT (LEN (CDR L)) L) )
      (b (CONS
            (MAKEDEADNEURON (LEN I))
            (AFTER (+ 1 (LEN (CDR L))) L))) ))

:hands-off (activate
            isLayer
            forwardLayer
            binary-append
            isNeuron) )
("Subgoal 2.1"
:hands-off (activate
            isLayer
            binary-append
            isNeuron)
:use (consp-forwardLayer
      do-it!
      (:instance car-front (i (len (cdr l))) (x l) )
      (:instance forwardLayer-over-app (a (FRONT (LEN (CDR L)) L) )
            (b (list (MAKEDEADNEURON (LEN I)) ) ) )

      forwardLayer-len
      forwardLayer-nil
      (:instance after-all-nil (n l) )
      (:instance one-step-forwardLayer (l (LIST (MAKEDEADNEURON (LEN I)) ) ) )
      (:instance forwardLayer-over-front (k (len (cdr l))) )
      (:instance after-all-nil (n (forwardLayer f i l)) )
      (:instance replaceValInList-front-after (j j)
            (n 0)
            (l (forwardLayer f i l) ) )
      (:instance one-neuron-isLayer (n (MAKEDEADNEURON (LEN I)) ) ) ) )
("Subgoal 1"
:hands-off (activate
            isLayer
            makeDeadNeuron
            forwardLayer
            isNeuron)
:use (consp-forwardLayer
      forwardLayer-len
      double-aaaaaaaaaargh!
      (:instance makeDeadNeuron-isDeadNeuron (i (len i)) )
      (:instance car-front (i j) (x l) )
      (:instance forwardLayer-over-front (k j) )
      (:instance forwardLayer-over-after (j (+ 1 j)) )
      (:instance add-neurons-to-layer (n (MAKEDEADNEURON (LEN I)) )
            (l (after (+ 1 j) l)) )
      (:instance forwardLayer-over-app (a (FRONT j L) )
            (b (CONS (MAKEDEADNEURON (LEN I))
                    (AFTER (+ 1 j) L)) ) )
      (:instance one-step-forwardLayer (l (CONS (MAKEDEADNEURON (LEN I))
            (AFTER (+ 1 J) L)) ) )
      (:instance after-all-nil (n (forwardLayer f i l)) )
      (:instance after-all-nil (n l) )
      (:instance same-neuron-len-in-after (j (+ 1 j)) )
      (:instance replaceValInList-front-after (j j)
            (n (MAKEDEADNEURON (LEN I)))
            (l l) )
      (:instance replaceValInList-front-after (j j)
            (n 0)
            (l (forwardLayer f i l) ) ) ) ) )

(defthmd small-duh
  (implies (and (isNetwork n)
                (not (equal l 0))
                (natp l)
                (< l (len n)) )
            (and (not (<= (LEN N) 1))
                 (CONSP (NTH L N)) ) ) )

```

```

:hints ("Goal"
       :hands-off (isNetwork)
       :use (nth-in-network-isLayer)
       :do-not-induct t)) )

(defthmd open-your-eyes!
  (implies (and (EQUAL (LEN (NTH (+ -1 L) N))
                      (LEN (CAR (NTH L N))))
              (EQUAL (LEN (FRONT L N)) L)
              (EQUAL (NTH (+ -1 L) (FRONT L N))
                      (NTH (+ -1 L) N)) )
            (EQUAL (LEN (NTH (+ -1 L) (FRONT L N)))
                  (LEN (CAR (NTH L N)))))) )

(defthmd meh
  (implies (and (< 1 (- (len n) 1))
                (natp 1)
                (consp n) )
            (not (<= (+ -1 1 (LEN (CDR N))) (+ -1 L))) ) )

(defthmd meh2
  (implies (and (natp 1)
                (isNetwork n)
                (natp j)
                (< 1 (- (len n) 1))
                (< j (len (nth 1 n)))
                (<= 2 (len (nth 1 n))) )
            (not (<= (+ 1 (LEN (CDR (NTH L N)))) J)) )

:hints ("Goal"
       :do-not-induct t)) )

(defthmd yargh
  (implies (and (EQUAL (LEN (CAR (NTH L N)))
                    (+ 1 (LEN (CDR (NTH J (NTH L N))))))
              (EQUAL (LEN (MAKEDEADNEURON (+ 1 (LEN (CDR (NTH J (NTH L N))))))
                    (+ 1 (LEN (CDR (NTH J (NTH L N)))))) )
              (EQUAL (LEN (MAKEDEADNEURON (LEN (CAR (NTH L N))))
                    (LEN (CAR (NTH L N)))))) )

;-----
; 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.

(defthmd prune-input-or-hidden-layer-dead-no-change
  (implies (and (isNetwork n)
                (isDeadNeuron (nth j (nth 1 n)))
                (isListOfWeights i)
                (equal (len i) (len (caar n)))
                (<= 2 (len (car (nth (+ 1 1) n))))
                (natp 1)
                (natp j)
                (< 1 (- (len n) 1))
                (<= 2 (len (nth 1 n)))
                (< j (len (nth 1 n))) )
            (equal (forwardNetwork f i n)
                  (forwardNetwork f i (removeFromNetwork 1 j n))) )

:hints ("Goal"
       :do-not '(generalize)
       :cases ( (equal 1 0) )
       :expand ( (removeFromNetwork 1 j n) )
       :use (more-bleh!
            isNetwork-cdr
            peel-from-after
            nth-in-network-isLayer
            forwardNetwork-over-removeFromNetwork
            isNetwork-isNetwork2-duh

```

```

forwardNetwork-over-front-after
isNetwork-true-listp
(:instance isDeadNeuron-duh (n (nth j (nth l n))) )
(:instance isLayer-true-listp (l (nth l n)) )
(:instance isLayer-isLayer2-duh (l (car n)) )
(:instance isNeuron-duh (n (caar n)) )
:hands-off (isNetwork
            isDeadNeuron
            isLayer
            isNeuron
            removeFromList
            removeFromNetwork
            repairRemainder
            front
            after
            forwardNetwork
            forwardLayer
            activate)
:do-not-induct t)
("Subgoal 2"
:use (impossible!
      (:instance car-front (i l) (x n) )
      front-isNetwork
      (:instance isListOfWeights-forwardNetwork (n (front l n)) )
      very-bleh!
      more-bleh!
      (:instance front-len (i l) (w n) )
      (:instance nth-forwardLayer-dead-neuron-is-zero
        (i (FORWARDNETWORK F I (FRONT L N)))
        (l (nth l n)) )
      (:instance one-step-forwardNetwork
        (i (FORWARDNETWORK F I (FRONT L N)))
        (n (CONS (REMOVEFROMLIST J (nth l n))
                 (REPAIRREMAINDER J (after (+ 1 L) N))) ) ) )
("Subgoal 2.2"
:do-not '(generalize fertilize)
:use ((:instance one-step-forwardNetwork
        (i (FORWARDNETWORK F I (FRONT L N)) )
        (n (after l n)) )
      (:instance after-isNetwork (l (+ 1 L)) )
      (:instance forwardLayer-over-removeFromList
        (i (FORWARDNETWORK F I (FRONT L N)) )
        (l (nth l n)) )
      (:instance cdr-after2 (i L) (x n) )
      (:instance nth-car-after (l L) (n n) )
      (:instance nth-car-after (l (+ 1 L)) (n n) )
      (:instance replaceValInList-zero-identity
        (i (FORWARDLAYER F (FORWARDNETWORK F I (FRONT L N)) (nth l n)) ) )
      (:instance nth-front (a (+ -1 L)) (b L) (n n) )
      (:instance len-forwardNetwork (n (front l n)) ) )
("Subgoal 2.2.4"
:use (come-on!3
      (:instance isListOfWeights-forwardLayer
        (i (FORWARDNETWORK F I (FRONT L N)) )
        (l (nth l n)) )
      (:instance forwardLayer-len (i (FORWARDNETWORK F I (FRONT L N)) )
        (l (NTH L N)) )
      (:instance nth-layer-in-network-consistent (l L) (n n) )
      (:instance nth-layer-in-network-consistent (l (+ -1 L)) (n n) )
      (:instance forwardNetwork-over-repairRemainder
        (i (FORWARDLAYER F (FORWARDNETWORK F I (FRONT L N))
        (NTH L N)) )
        (n (after (+ 1 L) n)) ) ) )
("Subgoal 2.2.4.1"
:do-not '(generalize) )
("Subgoal 2.2.1"
:use (come-on!3
      (:instance cdr-after2 (i L) (x n) )
      (:instance nth-car-after (l L) (n n) )
      (:instance nth-car-after (l (+ 1 L)) (n n) )
      (:instance nth-layer-in-network-consistent (l L) (n n) )

```

```

      (:instance nth-layer-in-network-consistent (l (+ -1 L)) (n n) )) )
("Subgoal 2.1"
 :use (specific-nth-bleh
      (:instance nth-layer-in-network-consistent (l L) (n n) )
      (:instance nth-layer-in-network-consistent (l (+ -1 L)) (n n) )
      (:instance nth-front (a (+ -1 L) ) (b L) (n n) )
      (:instance len-forwardNetwork (n (front l n) )) )
("Subgoal 1"
 :use ((:instance forwardLayer-over-removeFromList (i i)
      (l (car n) ))
      (:instance forwardLayer-len (l (car n)) )
      (:instance nth-forwardLayer-dead-neuron-is-zero (l (car n)) )
      (:instance replaceValInList-zero-identity
      (i (FORWARDLAYER F I (CAR N)) ))
      (:instance forwardNetwork-over-repairRemainder
      (i (FORWARDLAYER F I (CAR N))) (n (cdr n)) )
      (:instance isListOfWeights-forwardLayer (l (car n)) )
      (:instance one-step-forwardNetwork (i i)
      (n (CONS
      (REMOVEFROMLIST J (car N))
      (REPAIRREMAINDER J (cdr N)))) ) )
      forwardNetwork-nil)
:hands-off (isNetwork
            isDeadNeuron
            isLayer
            isNeuron
            removeFromList
            repairRemainder
            forwardNetwork
            forwardLayer
            activate) )
("Subgoal 1.2"
 :use (one-step-forwardNetwork) )
("Subgoal 1.1"
 :use ((:instance nth-layer-in-network-consistent (l 0) (n n) )) )) )

; 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.

(defthmd remove-input-or-hidden-equivalent-to-replace-with-dead
  (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) ) ) )

:hints ("Goal"
 :in-theory (enable len-0)
 :use ((:instance nth-in-network-isLayer (l l) (n n) )
      isNetwork-cdr
      (:instance isNeuron-not-zero (n (nth j (nth l n)) ))
      (:instance double-aaaaaaaaaaaaaargh! (i (nth j (nth l n)))) )
      isNetwork-isNetwork2-duh
      (:instance isLayer-isLayer2-duh (l (car n)) )
      (:instance len-replaceValInList (j j)
      (n (makeDeadNeuron
      (len (nth j (nth l n)))) )
      (l (nth l n) ))
      (:instance nth-in-layer-is-neuron (j j) (l (nth l n)) )
      (:instance isNeuron-duh (n (nth j (nth l n)) ))
      (:instance makeDeadNeuron-isDeadNeuron (i (len (nth j (nth l n)))) )) )
:hands-off (isNetwork
            isDeadNeuron

```

```

        isLayer
        isNeuron
        removeFromList
        removeFromNetwork
        repairRemainder
        front
        after
        makeDeadNeuron
        forwardNetwork
        forwardLayer
        activate)
:do-not '(generalize)
:cases ( (equal 1 0) )
:do-not-induct t)
("Subgoal 2"
:do-not '(generalize fertilize)
:hands-off (isNetwork
            isDeadNeuron
            isLayer
            isNeuron
            removeFromList
            removeFromNetwork
            repairRemainder
            front
            after
            nth
            makeDeadNeuron
            forwardNetwork
            forwardLayer
            activate)
:use (small-duh
      meh
      meh2
      yargh
      come-on!3
      (:instance nth-layer-in-network-consistent (1 (- 1 1)) (n n) )
      (:instance isLayer-<-2 (1 (nth 1 n)) )
      (:instance makeDeadNeuron-len (i (len (nth j (nth 1 n)))) ) )
      (:instance car-front (i 1) (x n) ) )
("Subgoal 2.8"
:do-not '(generalize)
:use (front-isNetwork
      (:instance isLayer-all-neurons-same (x (nth 1 n) ) )
      (:instance len-forwardNetwork (n (front 1 n)) )
      (:instance isLayer-nth-neuron-same (j j) (1 (nth 1 n)) )
      (:instance nth-front (a (+ -1 L) ) (b L) (n n) )
      (:instance isListOfWeights-forwardNetwork (n (front 1 n)) )
      (:instance isListOfWeights-forwardLayer (i (FORWARDNETWORK
                                                    F I (FRONT L N)) )
                                                    (1 (nth 1 n) ) )
      (:instance front-len (i 1) (w n) ) ) )
("Subgoal 2.8.1"
:do-not '(generalize)
:use (forwardNetwork-over-removeFromNetwork
      (:instance after-isNetwork (1 (+ 1 1)) )
      (:instance one-step-forwardNetwork (i (FORWARDNETWORK F I (FRONT L N)))
                                           (n (CONS
                                                (REMOVEFROMLIST J (nth 1 n))
                                                (REPAIRREMAINDER
                                                 J (after (+ 1 L) N))) ) )
      (:instance forwardLayer-over-removeFromList
        (i (FORWARDNETWORK F I (FRONT L N)) ) (1 (nth 1 n)) )
      (:instance forwardNetwork-over-replaceNeuronInLayerOfNetwork
        (net n) (n (makeDeadNeuron (len (nth j (nth 1 n)))) ) )
      (:instance nth-replaceValInList
        (n (makeDeadNeuron (len (nth j (nth 1 n)))) ) (1 (nth 1 n)) )
      (:instance forwardLayer-len (i (FORWARDNETWORK F I (FRONT L N)))
                                   (1 (NTH L N) ) )
      (:instance nth-car-after (1 (+ 1 L)) (n n) )
      (:instance nth-layer-in-network-consistent (1 L) (n n) )
      (:instance zero-forwardLayer-from-dead-neuron

```

```

        (i (FORWARDNETWORK F I (FRONT L N)) ) (l (nth l n)) )
(:instance one-step-forwardNetwork
  (i (FORWARDNETWORK F I (FRONT L N)) )
  (n (CONS (REPLACEVALINLIST
            J (NTH J (REPLACEVALINLIST
                    J (MAKEDEADNEURON (LEN (CAR (NTH L N))))
                    (NTH L N)))
            (NTH L N))
        (AFTER (+ 1 L) N)) ) )
(:instance forwardNetwork-over-repairRemainder
  (i (FORWARDLAYER F (FORWARDNETWORK F I (FRONT L N))
      (NTH L N)) )
  (n (AFTER (+ 1 L) N) ) ) )
("Subgoal 1"
 :do-not '(generalize)
 :use ((:instance isListOfWeights-forwardLayer (i i )
                (l (car n) ) )
        (:instance forwardNetwork-over-replaceNeuronInLayerOfNetwork
          (net n) (n (makeDeadNeuron (len (nth j (car n)))) ) )
        (:instance nth-replaceValInList
          (n (makeDeadNeuron (len (nth j (car n)))) ) (l (car n)) )
        (:instance forwardLayer-len (i i)
                (l (car N) ) )
        (:instance isLayer-all-neurons-same (x (car n) ) )
        (:instance forwardNetwork-over-repairRemainder
          (i (FORWARDLAYER F I (CAR N)) ) (n (cdr n) ) )
        (:instance zero-forwardLayer-from-dead-neuron (l (car n)) )
        (:instance forwardLayer-over-removeFromList (l (car n)) ) )
:hands-off (isNetwork
  isDeadNeuron
  isLayer
  isNeuron
  removeFromList
  repairRemainder
  makeDeadNeuron
  forwardLayer
  activate) )
("Subgoal 1.4"
 :use ((:instance nth-forwardLayer-dead-neuron-is-zero
  (i i ) (l (REPLACEVALINLIST
            J (MAKEDEADNEURON (LEN (NTH J (CAR N))))
            (CAR N)) ) ) )
("Subgoal 1.3"
 :use ((:instance nth-layer-in-network-consistent (l 0) (n n) )
        (:instance nth-forwardLayer-dead-neuron-is-zero
  (i i ) (l (REPLACEVALINLIST
            J (MAKEDEADNEURON (LEN (NTH J (CAR N))))
            (CAR N)) ) ) ) )
("Subgoal 1.1"
 :use ((:instance isLayer-<-2 (l (car n)) )
        (:instance isLayer-true-listp (l (car n)) )
        (:instance nth-forwardLayer-dead-neuron-is-zero
  (i i ) (l (REPLACEVALINLIST
            J (MAKEDEADNEURON (LEN (NTH J (CAR N))))
            (CAR N)) ) ) ) ) )

```

;-----