projet
Modélisation et simulation à base d'agents des systèmes Complexes

Programmation sous NetLogo
Exemple choisi: Artificial Neural Network - Multilayer
Modification: Construction d'un AutoEncoder

EL HADJ MUSTAPHA Mohamed Kais
LUSSIEZ Corentin

2021/2022
Université de Lyon 1 Claude Bernard

The example we chose is “Artificial Neural Net - Multilayer”, also known as an MLP or a multilayer perceptron.

1) *Explaining the chosen example*

Perceptrons

A perceptron is the simplest form of a neural network, a multilayer perceptron or a simple neural network is a set of connected perceptrons.

A perceptron is mainly composed of:

- A vector of input values, which allows us to enter our data to the perceptron, noted X.
- An output value, indicates whether the perceptron fires or not, after entering the data, noted y.
- A vector of weights, the important part of the perceptron, has the same size as the input vector; they go along in the perceptron structure, noted W.
- A bias value, can be considered as an extra weight, noted b.
- A sum function, performs the following:

$$Y = \sum(weight * input) + bias$$

- An activation function, a function that uses the returned value of the previous function as input, and returns the output value of the perceptron.
  We have multiple activation functions that could be used based on what we’re using the neural network for, examples:

  - Non linear activation, $f(x) = x$
  - Sigmoid activation, $f(x) = 1 / 1 + e^{-x}$
  - Tanh activation, $f(x) = e^{x} - e^{-x} / e^{x} + e^{-x}$
  - ReLU activation, $f(x) = max(0, x)$

- A learning rate, a small number that is used when the perceptron is trained.

How they work

Since we'll be mainly focusing on MLPs in this report, there is no need to speak in so much detail about perceptrons, but mainly, as any other ML model, perceptrons need to be trained and adapted to the training data. Perceptrons are a supervised learning type of model, this means our data will be labeled and might take the shape of: ***X1, X2, ..., Xn.*** for features and ***Y*** for the label.
Through the training phase, each line of the input data will be entered into the perceptron, plugged into the first function (sum function), the returned value will then plugged into the activation function, the returned value will be the perceptron's prediction of the label for the entered features, which can be noted ***Y'.***
Now the error rate will be measured between ***Y*** and ***Y',*** and used to slightly change the values of the perceptron's weights to adapt it more to the data, through something like,

$$Wn = Wn + (Y - Y') * Xn * learningRate$$

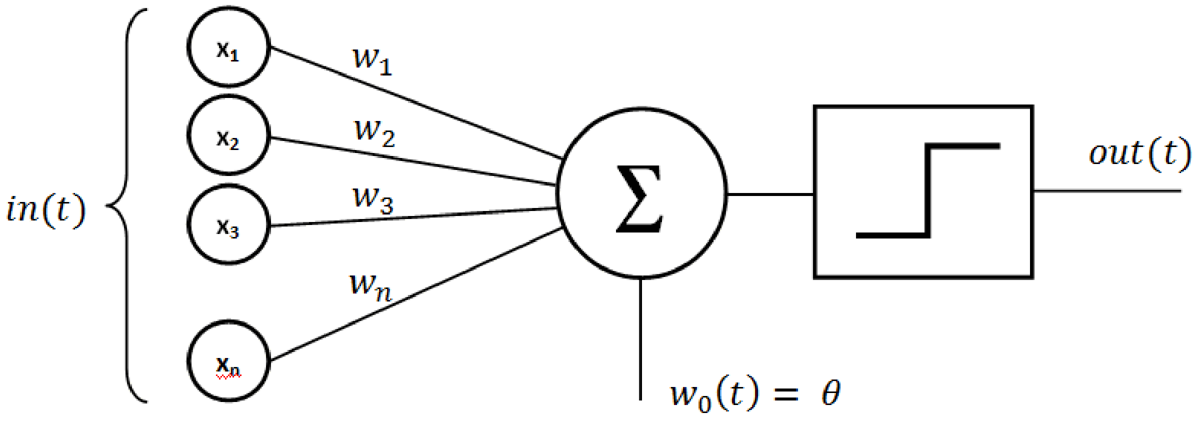


Figure 0: representation of a perceptron

MultiLayer Perceptrons

As already mentioned, an MLP is a set of perceptrons connected through activation functions.
MLPs come in use when the data gets too complex to be handled by a perceptron, they can get extremely complex.

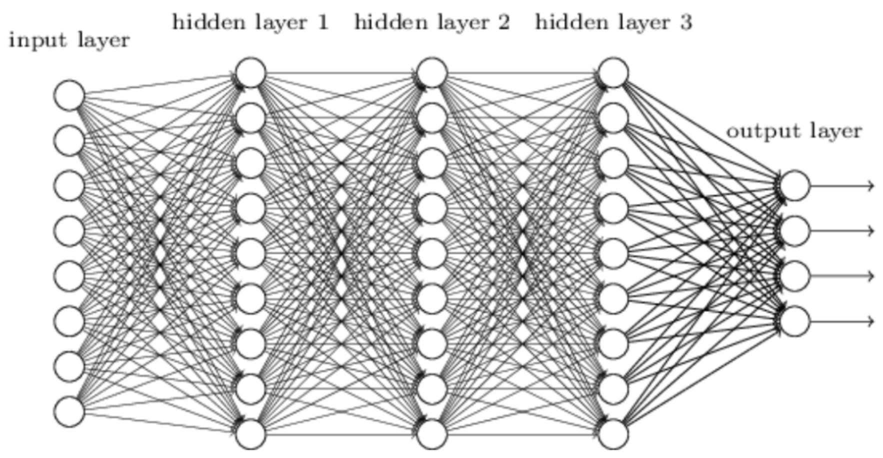


Figure 1: representation of a complex neural network

However, the example we chose here is fairly simple in comparison, it is composed of two input nodes, 2 hidden nodes in a one hidden layer and a one output node.

Now we will be talking about the essential parts of a neural network, its training, testing and core items throughout the NetLogo code of the model we have chosen.

FIrst of all, here is how our MLP looks.

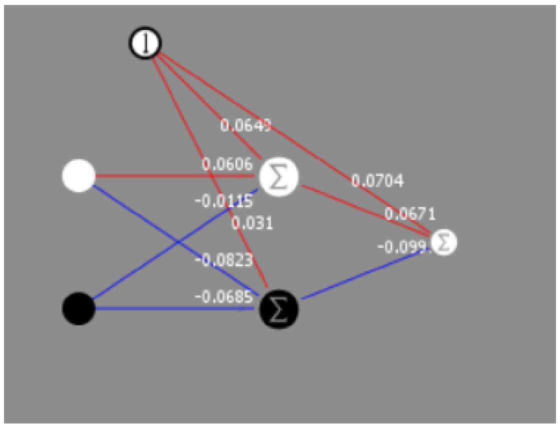
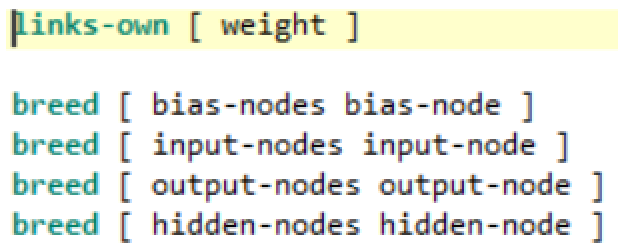


Figure 2: an image of the MLP model in NetLogo

Starting with the components of an MLP, they're pretty the same as the ones for a perceptron,

- Input nodes
- Output nodes
- Hidden nodes
- Bias nodes
- Weights

That is defined by the following line of code:



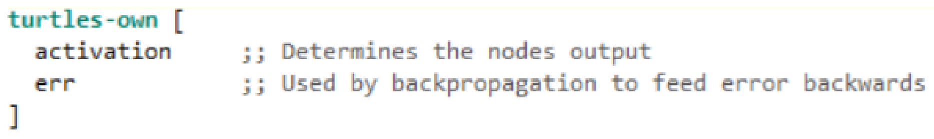
```
links-own [ weight ]

breed [ bias-nodes bias-node ]
breed [ input-nodes input-node ]
breed [ output-nodes output-node ]
breed [ hidden-nodes hidden-node ]
```

Figure 3: Code screenshot, MLP components

From the previous image, the keyword ***links-own*** allows adding a special property to the pre-defined agentset ***links,*** this agentset represents - as the name indicates - a set of links that can be used to connect agents to each other (aka ***turtles*** in NetLogo).

The keyword ***breed*** allows the creation of a new agentset, whereas the first input is the name of the agentset, and the second one the name of each member that belongs to that agentset.

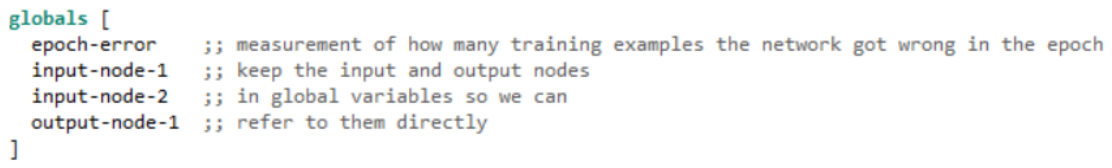
In MLPs, each node has a value that represents it, and an error value which is used by backpropagation later on.



```
turtles-own [
  activation     ;; Determines the nodes output
  err            ;; Used by backpropagation to feed error backwards
]
```

Figure 4: Code screenshot, node attributes

When working on on agent-based models in NetLogo, we could define global variables, in this case we define the following:



```
globals [
  epoch-error    ;; measurement of how many training examples the network got wrong in the epoch
  input-node-1   ;; keep the input and output nodes
  input-node-2   ;; in global variables so we can
  output-node-1  ;; refer to them directly
]
```

Figure 5: Code screenshot, global variables

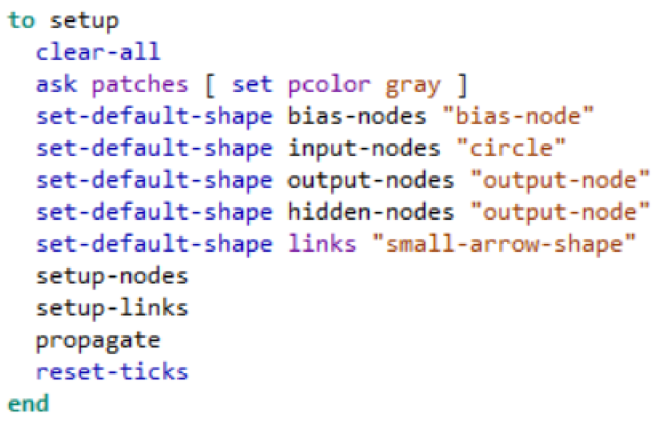
These variable names (and the comments along) explain their roles.

Now that we're done with definitions, time to move to the setup procedures.

```
;;;
;;; SETUP PROCEDURES
;;;
```

Figure 6: Code screenshot, setup procs

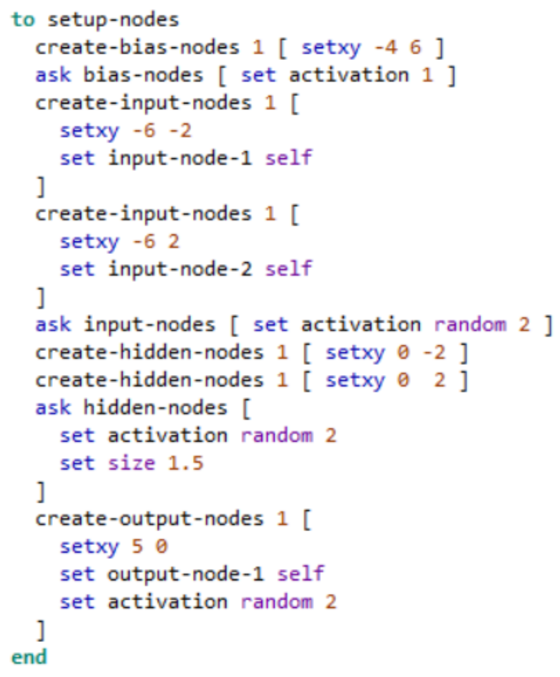
The first one is the ***setup*** procedure (procedures in NetLogo are created using the keywords ***to*** to being a procedure bloc and ***end*** and close the bloc).
This procedure is an essential one to all NetLogo models, originally when choosing a model, clicking the ***setup*** button (procedures can be attached to buttons to execute them freely) sets up the model in the environment to be used.



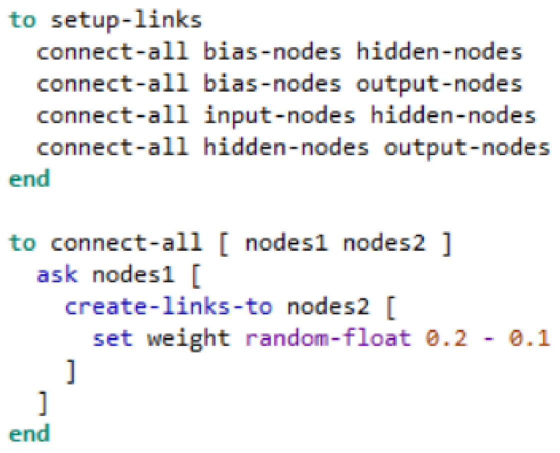
```
to setup
  clear-all
  ask patches [ set pcolor gray ]
  set-default-shape bias-nodes "bias-node"
  set-default-shape input-nodes "circle"
  set-default-shape output-nodes "output-node"
  set-default-shape hidden-nodes "output-node"
  set-default-shape links "small-arrow-shape"
  setup-nodes
  setup-links
  propagate
  reset-ticks
end
```

Figure 7: Code screenshot, setup procedure

We start by clearing up everything and basically doing a reset on the environment. Then color the ***patches*** of the environment (***patches*** is the unit used to define the measurements of the simulation environment, similar to pixels) using the gray color.
We proceed to define the default shapes of the MLP nodes, and lastly call three other procedures, ***setup-nodes***, ***setup-links*** and ***propagate,*** which will be discussed later.
Reset-ticks are used as a way to update the environment.



```
to setup-nodes
  create-bias-nodes 1 [ setxy -4 6 ]
  ask bias-nodes [ set activation 1 ]
  create-input-nodes 1 [
    setxy -6 -2
    set input-node-1 self
  ]
  create-input-nodes 1 [
    setxy -6 2
    set input-node-2 self
  ]
  ask input-nodes [ set activation random 2 ]
  create-hidden-nodes 1 [ setxy 0 -2 ]
  create-hidden-nodes 1 [ setxy 0  2 ]
  ask hidden-nodes [
    set activation random 2
    set size 1.5
  ]
  create-output-nodes 1 [
    setxy 5 0
    set output-node-1 self
    set activation random 2
  ]
end
```

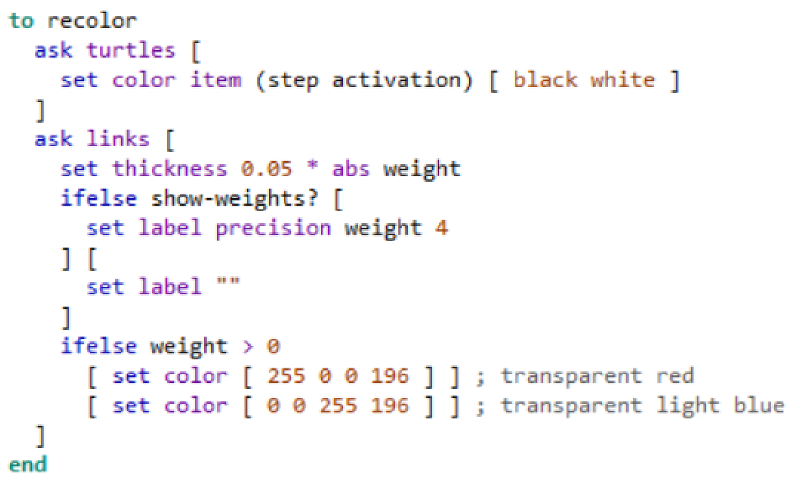


```
to setup-links
  connect-all bias-nodes hidden-nodes
  connect-all bias-nodes output-nodes
  connect-all input-nodes hidden-nodes
  connect-all hidden-nodes output-nodes
end

to connect-all [ nodes1 nodes2 ]
  ask nodes1 [
    create-links-to nodes2 [
      set weight random-float 0.2 - 0.1
    ]
  ]
end
```

Figure 7: Code screenshots, setting up the MLP

The ***setup-nodes*** procedure goes through the creation of the nodes using a syntax that is similar to create-turtles (which is used to create a set of agents. Turtles by default in NetLogo), more specifically through the commands setxy and set VARIABLE_NAME self to respectfully set up the ***x,y*** coordinates for the name of the node.
We set up the values of other properties as well, such as ***size*** and ***activation***.

Next we have the ***setup-links*** procedure, in which we used the ***connect-all*** procedure to basically connect the nodes to each other following the scheme displayed in the ***setup-link*** procedure.



```
to recolor
  ask turtles [
    set color item (step activation) [ black white ]
  ]
  ask links [
    set thickness 0.05 * abs weight
    ifelse show-weights? [
      set label precision weight 4
    ] [
      set label ""
    ]
    ifelse weight > 0
      [ set color [ 255 0 0 196 ] ] ; transparent red
      [ set color [ 0 0 255 196 ] ] ; transparent light blue
  ]
end
```

Figure 8: Code screenshot, node coloring

Last step of the setup is coloring the nodes, and configuring their colors to change during the training phase when activated.

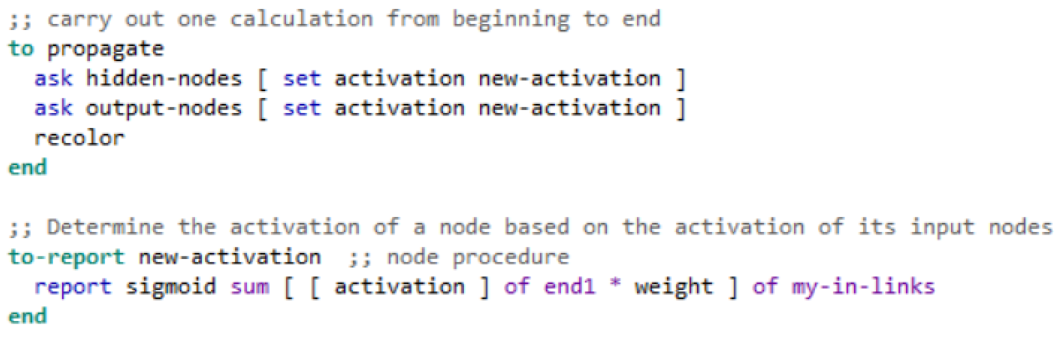
Now we get to the core parts of the model, propagation procedures.

```
;;;
;;; PROPAGATION PROCEDURES
;;;
```

Figure 9: Code screenshot, propagation procs

In deep learning, we have 2 very crucial concepts, forward-propagation and backward-propagation. Understanding these two is quite important for machine learning and deep learning practices.

Forward-propagation is a term that describes the process of simply running the neural network against some data, while training or testing. It is simply to use the network we built, which is done through applying the sum and the activation functions for each perceptron in the MLP until we obtain the final output value(s), which is done in the NetLogo model through the following code:



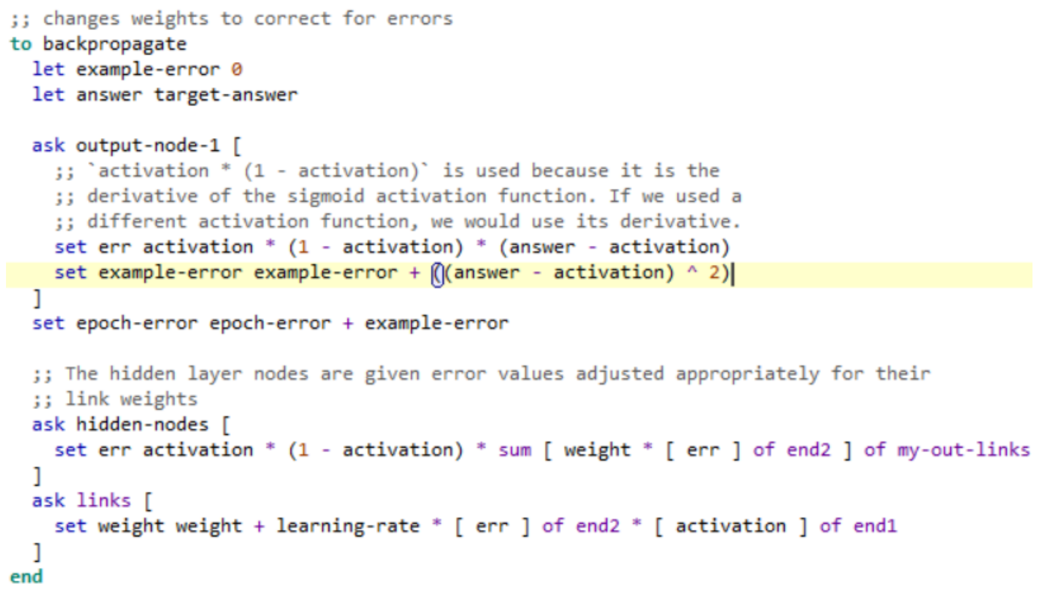
```
;; carry out one calculation from beginning to end
to propagate
  ask hidden-nodes [ set activation new-activation ]
  ask output-nodes [ set activation new-activation ]
  recolor
end

;; Determine the activation of a node based on the activation of its input nodes
to-report new-activation  ;; node procedure
  report sigmoid sum [ [ activation ] of end1 * weight ] of my-in-links
end
```

Figure 10: Code screenshot, forward-propagation

We notice that this activation is applied only to the ***hidden-nodes*** and the ***output-nodes***, which makes sense considering that ***input-nodes*** are simply input values.

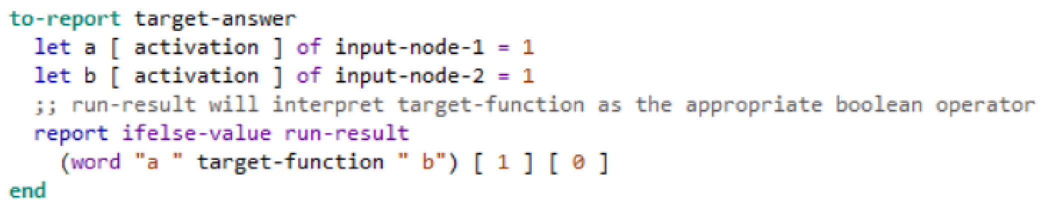
The real activation happens in the ***new-activation*** procedure, it's worth mentioning as well that the activation function used in this case is the ***sigmoid activation***, as indicated by the use of a procedure called ***sigmoid***.

Back-propagation is the phase that comes right after ***forward-propagation*** when training an MLP, after running the neural network, output values can be obtained, and based on that we calculate the total loss and say whether our model is doing good or not.If not, we make use of loss value to recalculate weights again for forward pass. And, this weight re-calculation process is made simple and efficient using ***back-propagation***.



```
;; changes weights to correct for errors
to backpropagate
  let example-error 0
  let answer target-answer

  ask output-node-1 [
    ;; `activation * (1 - activation)` is used because it is the
    ;; derivative of the sigmoid activation function. If we used a
    ;; different activation function, we would use its derivative.
    set err activation * (1 - activation) * (answer - activation)
    set example-error example-error + ((answer - activation) ^ 2)
  ]
  set epoch-error epoch-error + example-error

  ;; The hidden layer nodes are given error values adjusted appropriately for their
  ;; link weights
  ask hidden-nodes [
    set err activation * (1 - activation) * sum [ weight * [ err ] of end2 ] of my-out-links
  ]
  ask links [
    set weight weight + learning-rate * [ err ] of end2 * [ activation ] of end1
  ]
end
```

Figure 11: Code screenshot, back-propagation



```
to-report target-answer
  let a [ activation ] of input-node-1 = 1
  let b [ activation ] of input-node-2 = 1
  ;; run-result will interpret target-function as the appropriate boolean operator
  report ifelse-value run-result
    (word "a " target-function " b") [ 1 ] [ 0 ]
end
```

Figure 12: Code screenshot, target-answer

It's worth mentioning at first that procedures that begin with ***to-report*** are similar to functions in other programming languages, and the keyword ***report*** is used the same way as ***return***.

In the ***back-prop*** procedure, we start by setting ***example-error*** to zero, and set our answer (correct outcome/label) using the ***target-answer*** function, this latter will return either 1 or 0 based on the target-function we're trying to simulate.
If we're doing ***XOR*** and the inputs are 0 and 0, the returned value will be 0, this is the value that we will use to compare our model prediction to and use to measure the error rate (the loss).

Next, we visit the output-node, since we've already gone through forward-propagation, an output value already exists, that is in the ***[activation] of output-node-1*** in the NetLogo code.
We proceed then to measure the error, replacing ***activation*** by ***Y*** and ***answer*** by ***Y'***, we obtain this formula:
$Err = Y * (1 - Y) * (Y' - Y)$

We then increment the ***example-error*** by $(Y - Y')^2$, this ***example-error*** is added to the ***epoch-error***, which is already defined above.

This value will keep getting updated and will be used in a graphical representation in the interface of the NetLogo to portray the error rate per epoch.

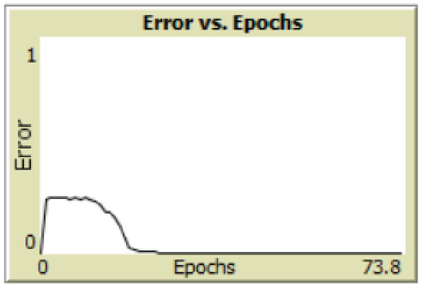
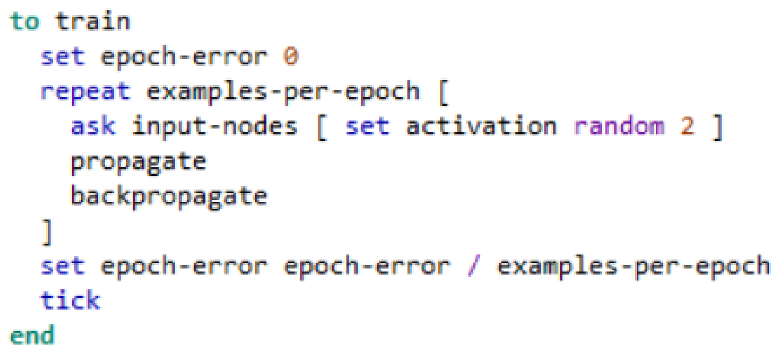


Figure 13: Error vs Epochs graph

We proceed then to calculate the error for the ***hidden-nodes***, while doing so, we try to adjust these error values to the weights of each ***hidden-node***, by entering in them into the error calculation formula, which becomes:
$Err = Y * (1 - Y) * sum(Wn * Yn)$

Where ***Yn*** is the activation of the node on the other side of the link weighted with ***Wn***.
Lastly we adjust the links of the MLP through the formula:
$Wn = Wn + learningRate * ErrRightSide * YLeftSide$

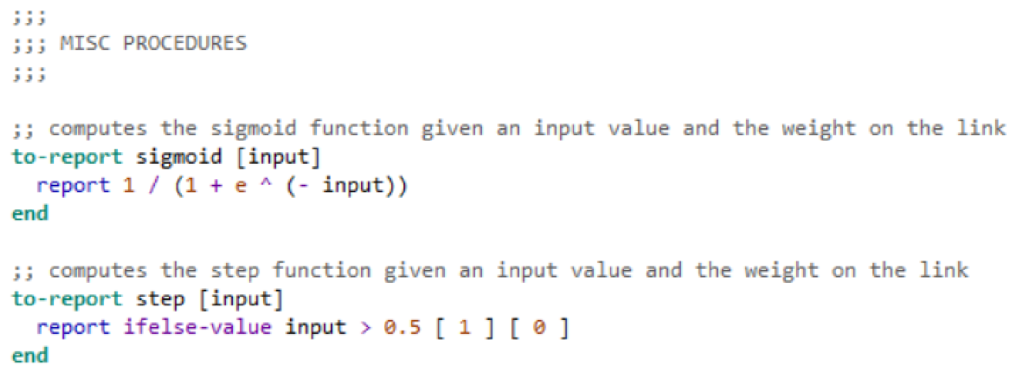
And like that, we finish one iteration on ***back-propagation***. The concept is that we keep running the cycle of forward to back propagation until the weights converge or for a certain number of epochs.
In this model, we apply this through the following procedure, which is attached to a button to use it directly:



```
to train
  set epoch-error 0
  repeat examples-per-epoch [
    ask input-nodes [ set activation random 2 ]
    propagate
    backpropagate
  ]
  set epoch-error epoch-error / examples-per-epoch
  tick
end
```

Figure 14: Code screenshot, training proc

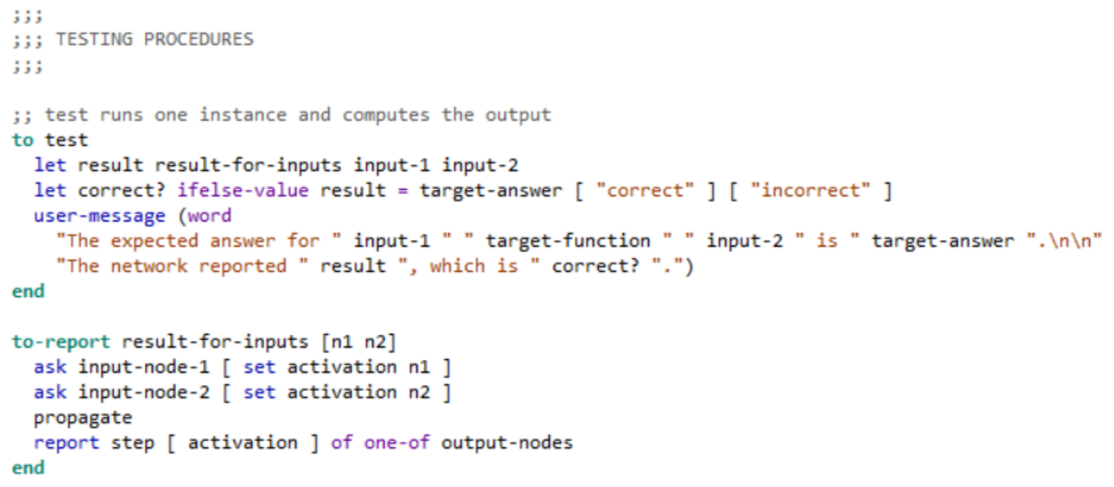
Misc-Procedures



```
;;;
;;; MISC PROCEDURES
;;;

;; computes the sigmoid function given an input value and the weight on the link
to-report sigmoid [input]
  report 1 / (1 + e ^ (- input))
end

;; computes the step function given an input value and the weight on the link
to-report step [input]
  report ifelse-value input > 0.5 [ 1 ] [ 0 ]
end
```

Figure 15: Code screenshot, activation functions

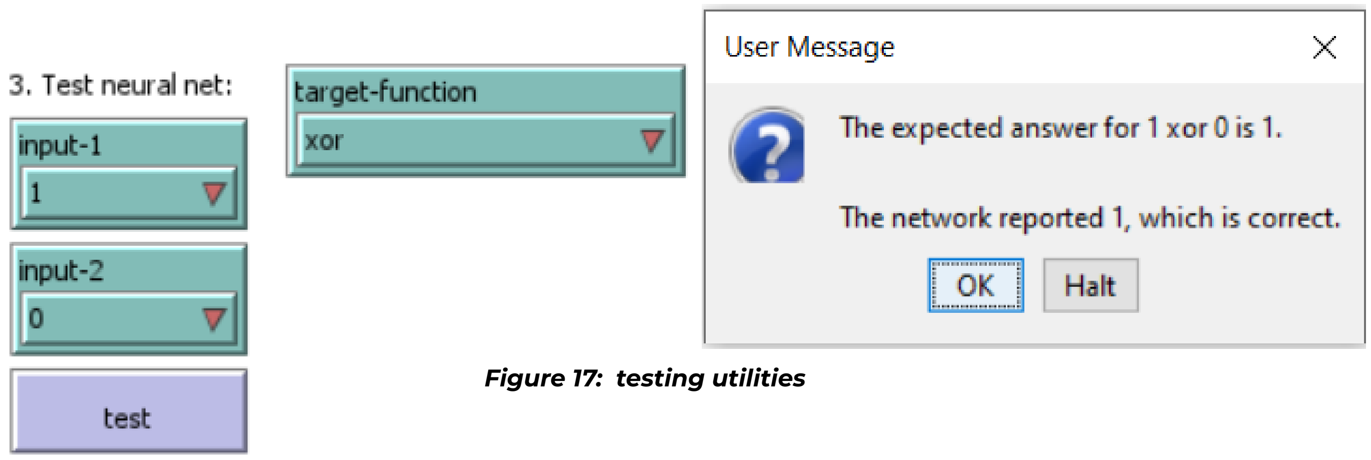
Testing Procedures



```
;;;
;;; TESTING PROCEDURES
;;;

;; test runs one instance and computes the output
to test
  let result result-for-inputs input-1 input-2
  let correct? ifelse-value result = target-answer [ "correct" ] [ "incorrect" ]
  user-message (word
    "The expected answer for " input-1 " " target-function " " input-2 " is " target-answer ".\n\n"
    "The network reported " result ", which is " correct? ".")
end

to-report result-for-inputs [n1 n2]
  ask input-node-1 [ set activation n1 ]
  ask input-node-2 [ set activation n2 ]
  propagate
  report step [ activation ] of one-of output-nodes
end
```

Figure 16: Code screenshot, testing code

The code was used to test the model as follows, considering that we're testing an ***MLP*** with as a target-function:



The expected answer for 1 xor 0 is 1.

The network reported 1, which is correct.

Figure 17: testing utilities

2) *Modifying the original model*

As a modification or a manipulation to add to this model, we have decided to build an Auto Encoder, which *is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning).[1] The encoding is validated and refined by attempting to regenerate the input from the encoding. The autoencoder learns a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore insignificant data ("noise").*

An autoencoder has two main parts: an encoder that maps the input into the code, and a decoder that maps the code to a reconstruction of the input, and that piece of code is called the bottleneck (comes right at the middle). source

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the latent-space representation.

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code. Source

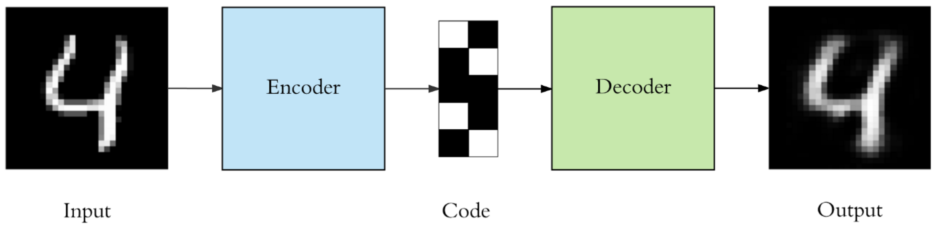
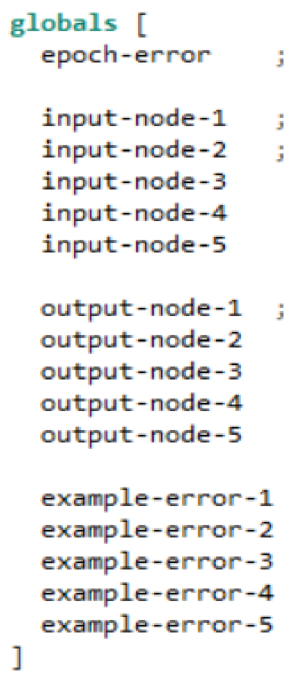


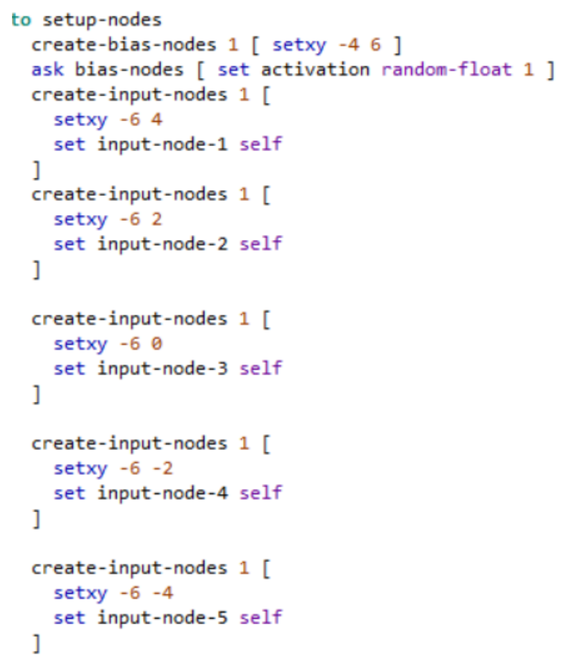
Figure 18: Autoencoder schema

Structure

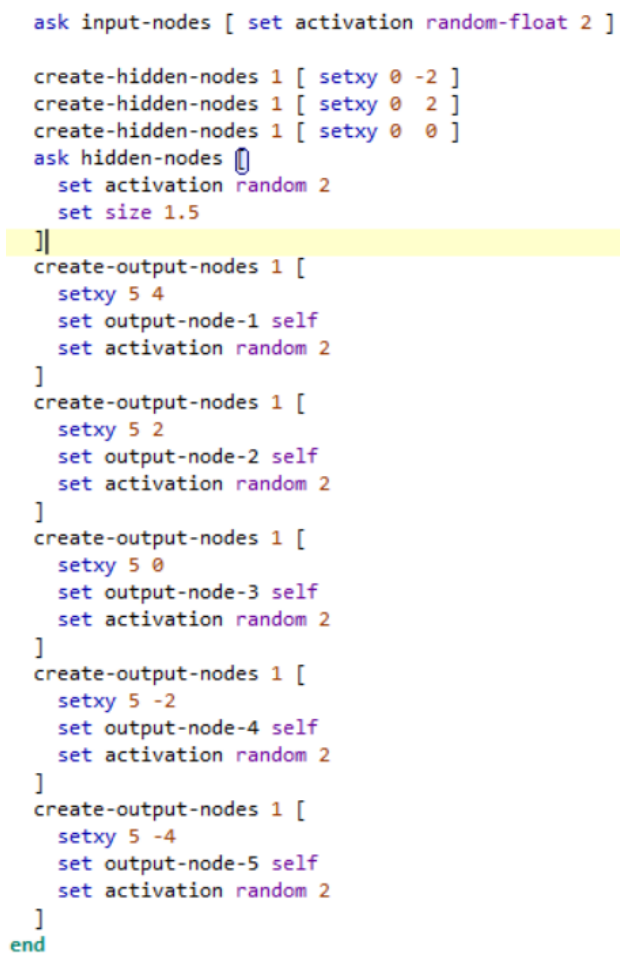
To create our autoencoder, we started first by changing the number of nodes in the neural network, into 5 input-nodes, 3 hidden-nodes and 5 output-nodes through the following code modifications:



```
globals [
  epoch-error     ;

  input-node-1    ;
  input-node-2    ;
  input-node-3
  input-node-4
  input-node-5

  output-node-1   ;
  output-node-2
  output-node-3
  output-node-4
  output-node-5

  example-error-1
  example-error-2
  example-error-3
  example-error-4
  example-error-5
]
```



```
to setup-nodes
  create-bias-nodes 1 [ setxy -4 6 ]
  ask bias-nodes [ set activation random-float 1 ]
  create-input-nodes 1 [
    setxy -6 4
    set input-node-1 self
  ]
  create-input-nodes 1 [
    setxy -6 2
    set input-node-2 self
  ]

  create-input-nodes 1 [
    setxy -6 0
    set input-node-3 self
  ]

  create-input-nodes 1 [
    setxy -6 -2
    set input-node-4 self
  ]

  create-input-nodes 1 [
    setxy -6 -4
    set input-node-5 self
  ]
```



```
  ask input-nodes [ set activation random-float 2 ]

  create-hidden-nodes 1 [ setxy 0 -2 ]
  create-hidden-nodes 1 [ setxy 0  2 ]
  create-hidden-nodes 1 [ setxy 0  0 ]
  ask hidden-nodes [
    set activation random 2
    set size 1.5
  ]
  create-output-nodes 1 [
    setxy 5 4
    set output-node-1 self
    set activation random 2
  ]
  create-output-nodes 1 [
    setxy 5 2
    set output-node-2 self
    set activation random 2
  ]
  create-output-nodes 1 [
    setxy 5 0
    set output-node-3 self
    set activation random 2
  ]
  create-output-nodes 1 [
    setxy 5 -2
    set output-node-4 self
    set activation random 2
  ]
  create-output-nodes 1 [
    setxy 5 -4
    set output-node-5 self
    set activation random 2
  ]
end
```

Figure 19: Code screenshots, autoencoder node creation

No big changes so far, just using the same code to create more nodes. We end up having the following neural network:

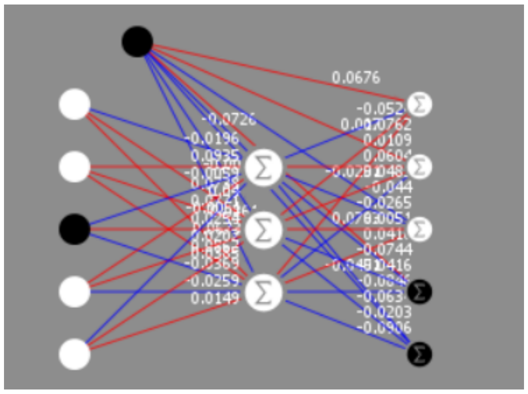


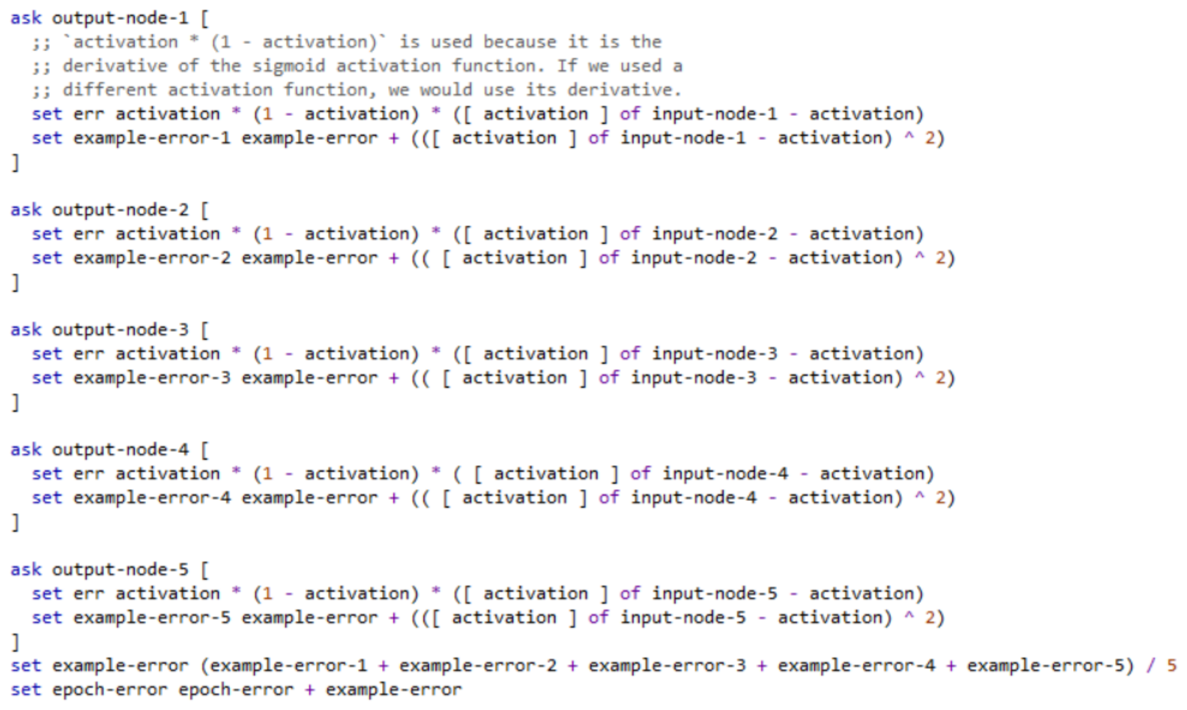
Figure 20: image of the autoencoder in NetLogo

Loss function

The second major modification is going to be at the backpropagation procedure, and specifically at the loss/cost function.
In the original model, the loss function is essentially based on the difference between the recent ***activation*** (as named in NetLogo) value (***Y'***) and the ***answer*** (as named in NetLogo) or expected value (***Y***). As in the formula,

$$Err = Y * (1 - Y) * (Y' - Y)$$

On other hand, in autoencoders, as mentioned already, what's important is to generate a set of output data points that is similar to a high accuracy to the original input data points that were entered into the autoencoder.
Since the input and output layers are identical in size in autoencoders, we could use the average difference between ***X1 and Y1, X2 and Y2, ..., Xn and Yn*** - where ***X*** is the input vector and ***Y*** is the output vector - as our loss function, since minimizing that value would mean that the input and output values are getting close to each other, which is exactly what we want.
This has been implemented through the following code:



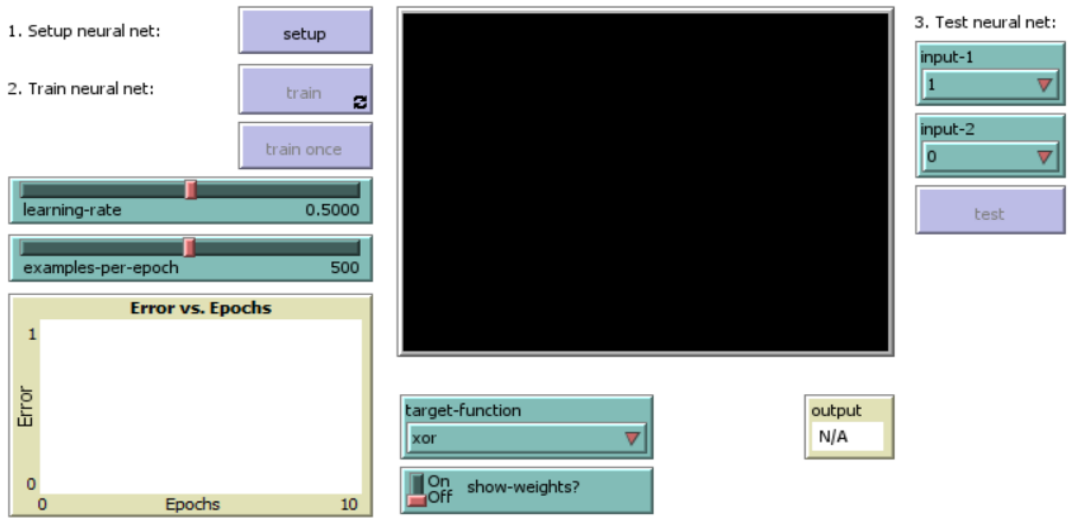
```
ask output-node-1 [
  ;; `activation * (1 - activation)` is used because it is the
  ;; derivative of the sigmoid activation function. If we used a
  ;; different activation function, we would use its derivative.
  set err activation * (1 - activation) * ([ activation ] of input-node-1 - activation)
  set example-error-1 example-error + (([ activation ] of input-node-1 - activation) ^ 2)
]

ask output-node-2 [
  set err activation * (1 - activation) * ([ activation ] of input-node-2 - activation)
  set example-error-2 example-error + (( [ activation ] of input-node-2 - activation) ^ 2)
]

ask output-node-3 [
  set err activation * (1 - activation) * ([ activation ] of input-node-3 - activation)
  set example-error-3 example-error + (( [ activation ] of input-node-3 - activation) ^ 2)
]

ask output-node-4 [
  set err activation * (1 - activation) * ( [ activation ] of input-node-4 - activation)
  set example-error-4 example-error + (( [ activation ] of input-node-4 - activation) ^ 2)
]

ask output-node-5 [
  set err activation * (1 - activation) * ([ activation ] of input-node-5 - activation)
  set example-error-5 example-error + (([ activation ] of input-node-5 - activation) ^ 2)
]
set example-error (example-error-1 + example-error-2 + example-error-3 + example-error-4 + example-error-5) / 5
set epoch-error epoch-error + example-error
```

Figure 21: Code Screenshot, autoencoder loss function

In the code above, we apply the concept to the five input and output nodes, where ***[activation] of input -node-X*** is our ***Y*** and ***activation*** is our ***Y'***.

Using the new model

To use the new modified model, we had to change the interface from this:



learning-rate 0.5000
examples-per-epoch 500
input-1 1
input-2 0
target-function xor
output N/A
On Off show-weights?

Figure 21: an image for the old interface of the MLP model

To this:

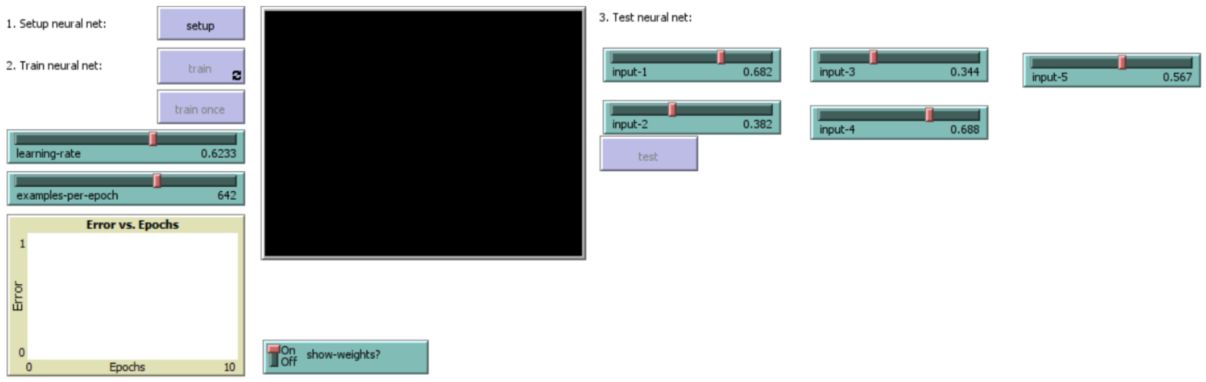
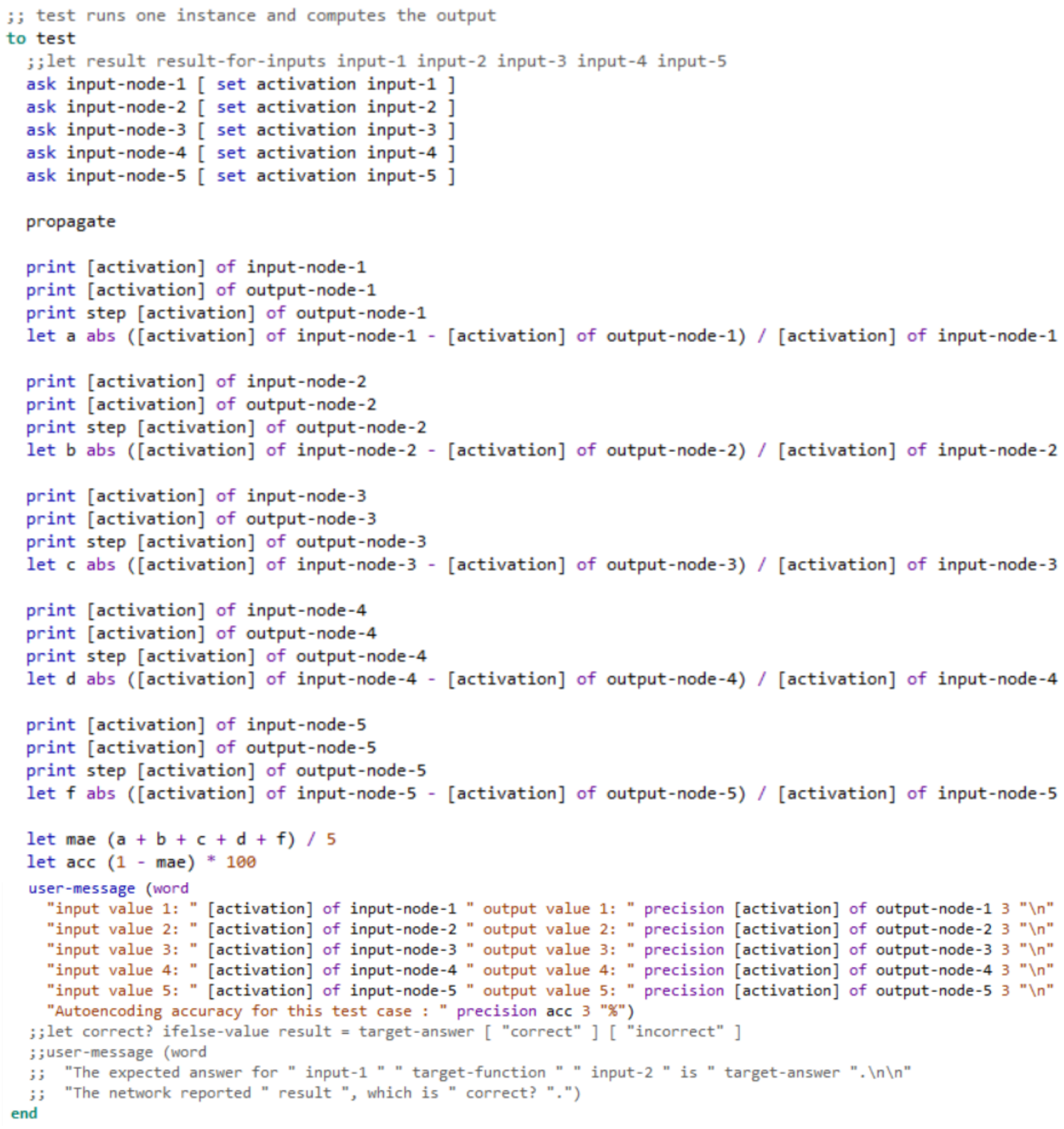


Figure 22: an image for the new interface of the autoencoder

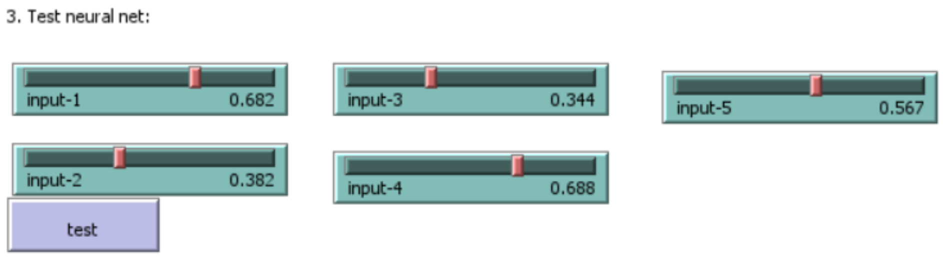
Instead of having 2 inputs, now we have 5, where the input option values aren’t only 1 or 0, but a value that is between them with 3 digits after the floating point.
We as well got rid of the target function and the output boxes, since they’re no longer needed in this type of neural network.
These final modifications have been done using the code below:

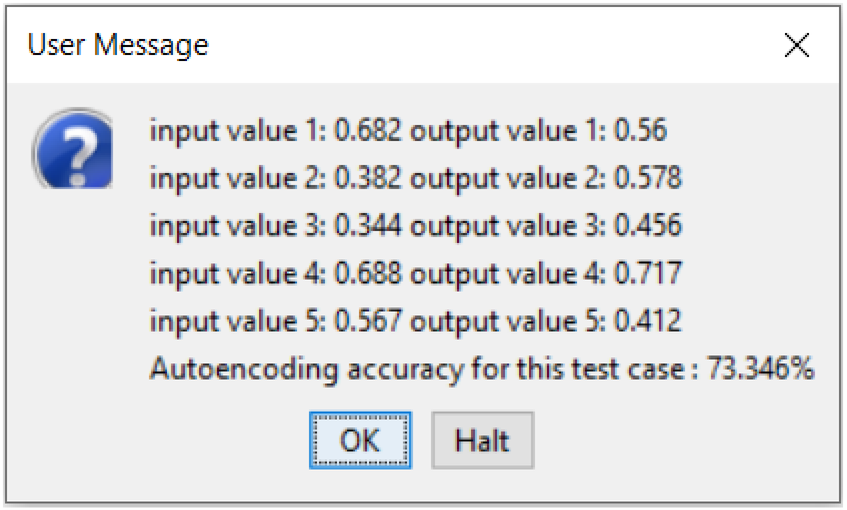


```
;; test runs one instance and computes the output
to test
  ;;let result result-for-inputs input-1 input-2 input-3 input-4 input-5
  ask input-node-1 [ set activation input-1 ]
  ask input-node-2 [ set activation input-2 ]
  ask input-node-3 [ set activation input-3 ]
  ask input-node-4 [ set activation input-4 ]
  ask input-node-5 [ set activation input-5 ]

  propagate

  print [activation] of input-node-1
  print [activation] of output-node-1
  print step [activation] of output-node-1
  let a abs ([activation] of input-node-1 - [activation] of output-node-1) / [activation] of input-node-1

  print [activation] of input-node-2
  print [activation] of output-node-2
  print step [activation] of output-node-2
  let b abs ([activation] of input-node-2 - [activation] of output-node-2) / [activation] of input-node-2

  print [activation] of input-node-3
  print [activation] of output-node-3
  print step [activation] of output-node-3
  let c abs ([activation] of input-node-3 - [activation] of output-node-3) / [activation] of input-node-3

  print [activation] of input-node-4
  print [activation] of output-node-4
  print step [activation] of output-node-4
  let d abs ([activation] of input-node-4 - [activation] of output-node-4) / [activation] of input-node-4

  print [activation] of input-node-5
  print [activation] of output-node-5
  print step [activation] of output-node-5
  let f abs ([activation] of input-node-5 - [activation] of output-node-5) / [activation] of input-node-5

  let mae (a + b + c + d + f) / 5
  let acc (1 - mae) * 100
  user-message (word
    "input value 1: " [activation] of input-node-1 " output value 1: " precision [activation] of output-node-1 3 "\n"
    "input value 2: " [activation] of input-node-2 " output value 2: " precision [activation] of output-node-2 3 "\n"
    "input value 3: " [activation] of input-node-3 " output value 3: " precision [activation] of output-node-3 3 "\n"
    "input value 4: " [activation] of input-node-4 " output value 4: " precision [activation] of output-node-4 3 "\n"
    "input value 5: " [activation] of input-node-5 " output value 5: " precision [activation] of output-node-5 3 "\n"
    "Autoencoding accuracy for this test case : " precision acc 3 "%")
  ;;let correct? ifelse-value result = target-answer [ "correct" ] [ "incorrect" ]
  ;;user-message (word
  ;;  "The expected answer for " input-1 " " target-function " " input-2 " is " target-answer ".\n\n"
  ;;  "The network reported " result ", which is " correct? ".")
end
```

Figure 21: Code Screenshot, testing code

Testing the model





```
input value 1: 0.682 output value 1: 0.56
input value 2: 0.382 output value 2: 0.578
input value 3: 0.344 output value 3: 0.456
input value 4: 0.688 output value 4: 0.717
input value 5: 0.567 output value 5: 0.412
Autoencoding accuracy for this test case : 73.346%
```

Figure 21: Code Screenshot, testing utilities

The model prints out the input values and the corresponding generated output value, then proceeds to calculate the autoencoding accuracy.