GUIDE TO MATLAB LEABRA

(By Sergio Verduzco-Flores)

Preamble

This document describes a basic implementation of the Leabra algorithms in Matlab. This implementation serves two main purposes. The first is to clarify the specific computations performed by the basic Leabra algorithms as they were in April 2015. Although a description of these algorithms is available online (https://grey.colorado.edu/emergent/index.php/Leabra), it is insufficient to reproduce the computations done by the Emergent implementation. The source code of the Emergent implementation is massive, so extracting its algorithms is not a trivial task.

The second purpose of this Leabra implementation is to allow any adept Matlab user to test and modify the Leabra algorithms. To this end, the implementation has been kept as simple as possible. Emergent is a very large, very flexible program that wants to do everything for you. This Matlab implementation is small, and you have to do everything yourself. This gives even more power, flexibility, and understanding, at the expense of having to do more work, risking bugs, and slower run times. Also, this means that people who are not comfortable with Matlab programming could run into trouble.

Overview

The Leabra implementation consists of three classes: *network*, *layer*, and *unit*. For each class there is one '.m' file where the class and its methods are defined. Another two files are included: 'rnd_assoc.m', and 'rnd_assoc_gui.m'. These two files are tutorials that show how to use the *network* class in order to build a random associator ('rnd_assoc.m'), and in order to build an autoassociator with a very simple graphical interface ('rnd_assoc_gui.m'). It is much simpler to explain how to use the *network* class when actually building a network that does something, so the file 'rnd_assoc.m' is profusely commented so as to clarify how to do this. Moreover, every part of the source code has detailed comments, so that anyone wanting to understand the algorithm can do it by reading the source. *The actual documentation for the program is thus in the source code*; this document only describes some general ideas about how each thing was implemented, and complements the comments in the source code. It is recommended that users first read the comments in here, and then the source code.

network class

The *network* class contains the *layer* objects that constitute a network, as well as a description of which layers are connected to which, and what relative weight scale is used for each projection. Layers can either be fully connected or not connected.

Notice that the *network* class is a handle object (and so are *layer* and *unit*), so that when any of its methods tries to modify a property of one of the layers, it doesn't modify that property in an independent copy. This also means that in all the method definitions the first argument is the network object itself (with the exception of the constructor).

network.cycle(inputs, clamp_inp)

This method performs one Leabra cycle. The *layer* and *unit* classes also have 'cycle' methods, and the basic idea is that the network calls 'cycle' for all its layers, and the 'cycle' method of each layer calls the 'cycle' method for all its units.

The argument 'inputs' is a 1-dimensional cell array, whose length is the number of layers. Each element of 'inputs' contains either an array with the inputs to a layer, or an empty matrix denoting no external inputs to the corresponding layer.

The 'clamp_inp' argument is a binary flag. 'clamp_inp = 1' indicates that the layer receiving input will have the activity of all their units clamped to the value of the input. This is the standard way external inputs are used in Leabra, and makes learning proceed nicely. When 'clamp_inp = 1', and a layer has an input, the method 'layer.clamped_cycle' is called instead of 'layer.cycle', since it is not necessary to calculate the unit activations.
Clamped activations of this type are biologically implausible, however, so 'clamp_inp = 0' indicates that external inputs should be considered as if arising from another layer, but applied directly to the units, without using synaptic weights.

A characteristic of the 'layer.cycle' method design is that all the synaptic weights of the layer are in a single matrix. Since a layer may or may not receive inputs from another layer, the input vector for a given layer has to be constructed according to this connectivity, so that the product of the weight matrix with the input vector produces the net inputs to the layer. Constructing this input vector is the most complicated piece of code in the 'network.cycle' method.

network.XCAL_learn
This method applies the XCAL learning algorithms. As seen in 'rnd_assoc.m', you call this method at the end of the plus phase, and it computes and applies the necessary weight changes based on the current value of the activity averages. The algorithms used are the ones described in:
https://grey.colorado.edu/ccnlab/index.php/Leabra_Hog_Prob_Fix#Adaptive_Contrast_Impl
Weight bounding and weight contrast enhancement are as described in:
https://grey.colorado.edu/emergent/index.php/Leabra
Before calculating weight changes, the 'network.XCAL_learn' method updates all the unit long-term averages of activity (the avg_l variables) by calling the 'layer.updt_avg_l' method for all layers.

The 'network.XCAL_learn' is vectorized so that operations are performed for all the weights in a projection. This makes the code a bit hard to read (and to write correctly), but it should also make the method faster.

network.updt_long_avgs
Leabra uses a very particular mechanism to scale the net inputs, so that the total contribution from each layer is (roughly) in the range between 0 and 1. The way net input is scaled is described in:
https://grey.colorado.edu/emergent/index.php/Leabra_Netin_Scaling

The averages used for netin scaling (acts_p_avg, pct_act_scale) are updated at the end of each plus phase, and this is the function that does the updating. You can see 'network.updt_long_avgs' being called at the end of every trial in 'rnd_assoc.m'.

An option is to use this function at the beginning of 'network.XCAL_learn' instead of having to call it after the plus phase. This is what is done for updating the long-term activity averages.

f = network.xcal(x,th)
This function implements the “check mark” function used by XCAL. An important thing is that the arguments 'x' and 'th' can be arrays, in which case the return value will also be an array. This is useful since it allows the 'network.XCAL_learn' method to calculate all the values of a projection at a time.

Notice that this function uses a dependent property called 'm1', which is the slope of the left-hand line segment in the check mark. I implemented this slope as a dependent property hoping that Matlab wouldn't calculate it again every time that 'network.xcal' was called.

network.reset
This function resets the activity of all units to zero, which is done at the beginning of each trial.

Originally 'network.reset' was meant to set the activity of all units at random values, but setting the activity to zero is consistent with the Emergent implementation. To make 'network.reset' set the activity to random values a single line needs to be modified in 'unit.reset', since 'network.reset' calls 'layer.reset' for all layers, and 'layer.reset' calls 'unit.reset' for all units.

network.set_weights(w)
This is a utility function that sets the weights of the network. The cell array 'w' has the same format as that used in the network constructor. This method also updates the contrast enhanced version of the weights, which is stored in a separate variable of the *layer* class.

w = network.get_weights
This method returns a cell array with all the weights of then network, in the same format used by the network constructor, and by the 'network.set_weights' method. This is useful for saving computation results (see Note 3).

layer class

This class contains all the *unit* objects of a layer, as well as all the methods to perform the computations that are appropriate to do at the layer level. This includes the computation of inhibition and of the net input that each unit receives in a cycle.

acts = layer.activities
Returns a vector with the activities of all the units.

out = layer.scaled_acts
Returns a vector with the activities of all the units scaled by the factor pct_act_scale (see https://grey.colorado.edu/emergent/index.php/Leabra_Netin_Scaling).

layer.cycle(raw_inputs, ext_inputs)
Performs one Leabra cycle for the layer. This involves obtaining the net inputs, obtaining the inhibition, and calling the 'unit.cycle' method for all units. The argument 'raw_inputs' consists of the scaled activities of the layers that project to this one. The argument 'ext_inputs' consists of external inputs, that will be applied directly to the units. Both arguments are set by the 'network.cycle' method, and 'ext_inputs' is only used when 'network.cycle' is used with 'clamp_inp = 0'.

A distinctive feature of this implementation is that there is a single weight matrix that contains the weights for all the inputs to the layer. This allows the computation of the net inputs using a single matrix multiplication, but complicates the calculation of the input vectors in 'network.cycle'.

layer.clamped_cycle(input)
Sets all unit activities equal to the input argument by calling the method 'unit.clamped_cycle' for all

units.

[avg_s, avg_m, avg_l] = layer.averages
This method returns vectors with all the short-term, medium-term, and long-term averages. These vectors are used by 'network.XCAL_learn' in order to calculate the weight changes. Notice that the values of of the averages are not updated before being returned.

l_avg_rel = layer.rel_avg_l
Returns the rel_avg_l value, which is rescaled version of the long-term average, used in the XCAL algorithm to calculate the amount of "BCM" learning.

layer.updt_avg_l
Updates the long-term average (avg_l) for all the units in the layer. Currently this method is called from the 'network.XCAL_learn' function.

layer.reset
Calls the 'unit.reset' method for all units.

avg_acts = get.acts_avg
'acts_avg' is a dependent property of the *layer* class, containing the average of all the current, unscaled unit activities. This method returns 'acts_avg'.

unit class

The objects of the unit class contain the activation variables, all their averages, and methods necessary to update them each cycle.

unit.cycle(net_raw, gc_i)
This method performs one Leabra cycle for the unit. The argument 'net_raw' is the excitatory input, and 'gc_i' is the FFFB inhibition. Both arguments are calculated by the 'layer.cycle' method.

This implementation is slightly different from the one in Emergent. The Emergent implementation does not include the value of adaptation when using the half-step method to update 'v_m'. Notice that neither implementation performs a real half-step, since the value of adaptation is not updated (to perform fewer computations, I guess).

unit.clamped_cycle(input)
Performs one cycle for the unit when its activity is clamped to an input value. The method just consists of setting the activity equal to the input, and updating the averages.

unit.updt_avg_l
Updates the long-term average (avg_l). Based on the description in:
https://grey.colorado.edu/ccnlab/index.php/Leabra_Hog_Prob_Fix#Adaptive_Contrast_Impl

unit.reset
sets the activity and all the averages to zero. Membrane voltage is set to a value close to the leak reversal potential. Changing one line can instead cause the activity to be set to a random value. This method is called by the 'layer.reset' method, which in turn is called by the 'network.reset' method.

f = unit.nxx1(points)
This function returns the values of the noisy x/(x+1) function evaluated at all points of the input vector 'points'. The noisy x/(x+1) function comes from the convolution x/(x+1) with a Gaussian function, whose standard deviation is set here, not as a property of the *unit* class.

To avoid calculating the convolution every time, the first time that 'unit.nxx1' is called the return values are calculated for all the points in a vector 'nxx1_dom', and stored in a vector called 'nxoxp1'. Each subsequent time that the function is called the return values are calculated by interpolating with these vectors.

Notes

1) In the tests I have performed this implementation of Leabra is not identical to the one in Emergent. In particular, the learning rates used in this implementation are much larger than those in Emergent.
2) Using parallel computation with a 'parfor' in the calls to 'cycle' functions does not currently improve performance. The reason is that units and layers are handle objects, so when calling the 'unit.cycle' method for all units of a layer in parallel will result in all of them wanting to communicate at the same time with their layer, and there being only one copy of the layer this results in large communication overhead.
3) One thing the tutorial programs 'rnd_assoc.m' and 'rnd_assoc_gui.m' do not do is to save and load learned weights. A network is sufficiently characterized by a cell array with its synaptic weights (as used by the constructor) and by its 'connections' matrix. The weights cell can be retrieved using the function 'network.get_weights'. In addition, weights can be set using the 'network.set_weights' function. For example, the command “net.set_weights(net.get_weights)” when introduced in the tutorial programs has no effect (except slowing things down).
   To save a network, you just save the 'connected' matrix and the 'w' cell array (obtained with the 'network.get_weights' function) using Matlab's 'save' command. Optionally, you can also save the 'dim_lays' cell array. To retrieve the network you use the 'load' command, and use the retrieved variables as arguments for the network constructor.
4) In 'rnd_assoc_gui.m', the 'bar3' function is used to plot the activities and weights of the network. While this is very convenient, it is also a bit slow, since the axes are redrawn each time the plot is updated. A faster approach is to 'set' the data of the plot and calling 'drawnow', as done for the subplot with the errors by epoch. This is easy to do with 2D plots, but hard to do with 'bar3'. I leave fast 3D plotting to someone who knows Matlab better than me.
5) This program was created with Matlab R2014b. Earlier versions may produce problems, especially with 'rnd_assoc_gui.m'.
6) I probably won't dedicate much more time to this project, but you can send me your comments at: sergio.verduzco@gmail.com