Self Organizing

Introduction
The defining property of self-organizing learning is that it operates without requiring an explicit training signal from the environment. This can be contrasted with error backpropagation, which requires target patterns to compare against the output states in order to generate an error signal. Thus, many people regard self-organizing learning as more biologically or psychologically plausible, since it is often difficult to imagine where the explicit training signals necessary for error-driven learning come from. Further, there is some evidence that neurons actually use something like a Hebbian learning rule, which is commonly used in self-organizing learning algorithms.

There are many different flavors of self-organizing learning. Indeed, one of the main differences between self-organizing algorithms and error-driven learning is that they need to make more assumptions about what good representations should be like, since they do not have explicit error signals telling them what to do. Thus, different self-organizing learning algorithms make different assumptions about the environment and how best to represent it.

One assumption that is common to many self-organizing learning algorithms is that events in the environment can be clustered together according to their "similarity." Thus, learning amounts to trying to find the right cluster in which to represent a given event. this is often done by enforcing a competition between a set of units, each of which represents a different cluster. The competitive learning algorithm (CL) of Rumelhart and Zipser, 1985 is a classic example of this form of learning, where the single unit which is most activated by the current input is chosen as the "winner" and therefore gets to adapt its weights in response to this input pattern.

The current implementation of self-organizing learning, called So, includes competitive learning and several variations of it, including "soft" competitive learning (Nowlan, 1990), which replaces the "hard" competition of standard competitive learning with a more graded activation function. Also included are a couple of different types of modified Hebbian learning rules that can be used with either hard or soft activation functions.

An additional assumption that can be made about the environment is that there is some kind of topology or ordered relationship among the different clusters. This notion is captured in the self-organizing map (SOM) algorithm of Kohonen (1989, 1990, 1995). This algorithm adds to the basic idea of competition among the units that represent a cluster the additional assumption that units which are nearby in 2-D space should represent clusters that are somehow related. This spatial-relatedness constraint is imposed by allowing nearby units to learn a little bit when one of their neighbors wins the competition. This algorithm is also implemented in the So package.

The directory <file>demo/so</file> contains two projects which demonstrate the use of both the competitive-learning style algorithms, and the self-organizing maps.

Implementation Reference
The So implementation is designed to be used in a mix-and-match fashion. Thus, there are a number of different learning algorithms, and several different activation functions, each of which can be used with the other. The learning algorithms are implemented as different connection specs derived from a basic

Reference info for type <a href="http://grey.colorado.edu/gendoc/emergent/SoConSpec.html">SoConSpec</a>: Wiki | Emergent Help Browser

Reference info for type <a href="http://grey.colorado.edu/gendoc/emergent/SoCon.html">SoCon</a>: Wiki | Emergent Help Browser

type. They all use the same
Learning Functions (ConSpecs)

- Reference info for type SoConSpec: <a href="http://grey.colorado.edu/gendoc/emergent/SoConSpec.html">Wiki</a> | <a href="ta:.Type.SoConSpec">Emergent Help Browser</a>

(base spec): has a learning rate parameter lrate, and a range to keep the weight values in: wt_range. Unlike error-driven learning, many self-organizing learning algorithms require the weights to be forcibly bounded, since the positive-feedback loop phenomenon of associative learning can lead to infinite weight growth otherwise. Finally, there is a variable which determines how to compute the average and summed activation of the input layer(s), which is needed for some of the learning rules. If the network is fully connected, then one can set avg_act_source to compute from the LAYER_AVG_ACT, which does not require any further computation. However, if the units receive connections from only a sub-sample of the input layer, then the layer average might not correspond to that which is actually seen by individual units, so you might want to use COMPUTE_AVG_ACT, even though it is more computationally expensive.

- Reference info for type HebbConSpec: <a href="http://grey.colorado.edu/gendoc/emergent/HebbConSpec.html">Wiki</a> | <a href="ta:.Type.HebbConSpec">Emergent Help Browser</a>

This computes the most basic Hebbian learning rule, which is just the coproduct of the sending and receiving unit activations:

```
    cn->dwt += ru->act * su->act;
```

Though it is perhaps the simplest and clearest associative learning rule, its limitations are many, including the fact that the weights will typically grow without bound. Also, for any weight decrease to take place, it is essential that activations be able to take on negative values. Keep this in mind when using this form of learning. One application of this con spec is for simple pattern association, where both the input and output patterns are determined by the environment, and learning occurs between these patterns.

- Reference info for type CIConSpec: <a href="http://grey.colorado.edu/gendoc/emergent/CIConSpec.html">Wiki</a> | <a href="ta:.Type.CIConSpec">Emergent Help Browser</a>

This implements the standard competitive learning algorithm as described in (Rumelhart & Zipser, 1985). This rule can be seen as attempting to align the weight vector of a given unit with the center of the cluster of input activation vectors that the unit responds to. Thus, each learning trial simply moves the weights some amount towards the input activations. In standard competitive learning, the vector of input activations is normalized by dividing by the sum of the input activations for the input layer, sum_in_act (see avg_act_source above for details on how this is computed).

```
    cn->dwt += ru->act * ((su->act / cg->sum_in_act) - cn->wt);
```

The amount of learning is "gated" by the receiving unit's activation, which is determined by the competitive learning function. In the winner-take-all "hard" competition used in standard competitive learning, this means that only the
winning unit gets to learn. Note that if you multiply through in the above equation, it is equivalent to a Hebbian-like
term minus something that looks like weight decay:

\[ \text{cn->dwt} += \text{ru->act} \times (\text{su->act} / \text{cg->sum_in_act}) \] 
\[ \text{-(ru->act \times cn->wt);} \]

This solves both the weight bounding and the weight decrease problems with pure Hebbian learning as implemented
in the HebbConSpec described above.

- **Reference info for type SoftClConSpec:** [Wiki](http://grey.colorado.edu/gendoc/emergent/SoftClConSpec.html) | [Emergent Help Browser](http://grey.colorado.edu/gendoc/emergent/SoftClConSpec.html)

This implements the "soft" version of the competitive learning learning
rule (Nowlan, 1990). This is essentially the same as the "hard" version, except that it does not normalize the input activations. Thus, the weights move towards the center of the actual activation vector. This can be thought of in
terms of maximizing the value of a multi-dimensional Gaussian function of the distance between the weight vector
and the activation vector, which is the form of the learning rule used in soft competitive learning. The smaller the
distance between the weight and activation vectors, the greater the activation value.

\[ \text{cn->dwt} += \text{ru->act} \times (\text{su->act} - \text{cn->wt}); \]

This is also the form of learning used in the self-organizing map algorithm, which also seeks to minimize the
distance between the weight and activation vectors. The receiving activation value again gates the weight change. In
soft competitive learning, this activation is determined by a soft competition among the units. In the SOM, the
activation is a function of the activation kernel centered around the unit with the smallest distance between the
weight and activation vectors.

- **Reference info for type ZshConSpec:** [Wiki](http://grey.colorado.edu/gendoc/emergent/ZshConSpec.html) | [Emergent Help Browser](http://grey.colorado.edu/gendoc/emergent/ZshConSpec.html)

This implements the "zero-sum" Hebbian learning algorithm (ZSH)
(O'Reilly & McClelland, 1992), which implements a form of subtractive weight constraints, as opposed to the
multiplicative constraints used in competitive learning. Multiplicative constraints work to keep the weight vector
from growing without bound by maintaining the length of the weight vector normalized to that of the activation
vector. This normalization preserves the ratios of the relative correlations of the input units with the cluster
represented by a given unit. In contrast, the subtractive weight constraints in ZSH exaggerate the weights to those
inputs which are greater than the average input activation level, and diminish those to inputs which are below
average:

\[ \text{cn->dwt} += \text{ru->act} \times (\text{su->act} - \text{cg->avg_in_act}); \]

where `avg_in_act` is the average input activation level. Thus, those inputs which are above average have their
weights increased, and those which are below average have them decreased. This causes the weights to go into a
corner of the hypercube of weight values (i.e., weights tend to be either 0 or 1). Because weights are going towards
the extremes in ZSH, it is useful to introduce a "soft" weight bounding which causes the weights to approach the
bounds set by `@code{wt_range}` in an exponential-approach fashion. If the weight change is greater than zero, then
it is multiplied by `wt_range.max - cn->wt`, and if it is less than zero, it is multiplied by `cn->wt - wt_range.min`. This
is selected by using the `soft_wt_bound` option.

- **Reference info for type MaxInConSpec:** [Wiki](http://grey.colorado.edu/gendoc/emergent/MaxInConSpec.html) | [Emergent Help Browser](http://grey.colorado.edu/gendoc/emergent/MaxInConSpec.html)

This learning rule is basically just the combination of SoftCl and Zsh.
It turns out that both of these rules can be derived from an objective function which seeks to maximize the input information a unit receives, which is defined as the signal-to-noise ratio of the unit's response to a given input signal (O'Reilly, 1994). The formal derivation is based on a different kind of activation function than those implemented here, and it has a special term which weights the Zsh-like term according to how well the signal is already being separated from the noise. Thus, this implementation is simpler, and it just combines Zsh and SoftCl in an additive way:

\[
\text{cn->dwt} += \text{ru->act} \times (\text{su->act} - \text{cg->avg_in_act}) + \\
\text{k_scl} \times \text{ru->act} \times (\text{su->act} - \text{cn->wt});
\]

Note that the parameter `\text{k_scl}` can be adjusted to control the influence of the SoftCl term. Also, the soft_wt_bound option applies here as well.

**Activation Functions (UnitSpecs and LayerSpecs)**

Activity of units in the So implementation is determined jointly by the unit specifications and the layer specifications. The unit specifications determine how each unit individually computes its net input and activation, while the layer specifications determine the actual activation of the unit based on a competition that occurs between all the units in a layer.

The So implementation uses

Reference info for type *SoLayerSpec*: <a href="http://grey.colorado.edu/gendoc/emergent/SoLayerSpec.html">Wiki</a> | <a href="ta:.Type.SoLayerSpec">Emergent Help Browser</a>

objects extensively. These layer specifications implement competition among units in the same layer, which is central to the self-organizing algorithms. There are different layer specs all derived from a common SoLayerSpec.

All So Algorithms use the same

Reference info for type *SoUnit*: <a href="http://grey.colorado.edu/gendoc/emergent/SoUnit.html">Wiki</a> | <a href="ta:.Type.SoUnit">Emergent Help Browser</a>

The only thing this type adds to the basic Unit is the `act_i` value, which reflects the "independent" activation of the unit prior to any modifications that the layer-level competition has on the final activations. This is primarily useful for the soft Cl units, which actually transform the net input term with a Gaussian activation function, the parameters of which can be tuned by viewing the resulting `act_i` values that they produce.

There are three basic types of unit specifications, two of which derive from a common

Reference info for type *SoUnitSpec*: <a href="http://grey.colorado.edu/gendoc/emergent/SoUnitSpec.html">Wiki</a> | <a href="ta:.Type.SoUnitSpec">Emergent Help Browser</a>

The SoUnitSpec does a very simple linear activation function of the net input to the unit. It can be used for standard competitive learning, or for Hebbian learning on linear units.

- **Reference info for type ClLayerSpec**: <a href="http://grey.colorado.edu/gendoc/emergent/ClLayerSpec.html">Wiki</a> | <a href="ta:.Type.ClLayerSpec">Emergent Help Browser</a>

  -- hard competitive learning:

...
selects the winning unit (based on netin_type), and assigns it an activation value of 1, and it assigns all other units a value of 0. Thus, only the winning unit gets to learn about the current input pattern. This is a "hard" winner-take-all competition.

- Reference info for type SoftClLayerSpec: 
  
  Reference info for type SoftClUnitSpec: 

-- soft competitive learning: computes a Gaussian function of the distance between the weight and activation vectors. The variance of the Gaussian is given by the var parameter, which is not adapting and shared by all weights in the standard implementation, resulting in a fixed spherical Gaussian function. Note that the net variable on units using this spec is the distance measure, not the standard dot product of the weights and activations. The SoftClLayerSpec does not explicitly select a winning unit. Instead, it assigns each unit an activation value based on a SoftMax function: 

\[
\hat{a}_j = \frac{e^{g_j}}{\sum_k e^{g_k}}
\]

Where \(g_j\) is the Gaussian function of the distance between the unit's weights and activations (stored in act_i on the SoUnit object). Thus, the total activation in a layer is normalized to add up to 1 by dividing through by the sum over the layer. The exponential function serves to magnify the differences between units. There is an additional softmax_gain parameter which multiplies the Gaussian terms before they are put through the exponential function, which can be used to sharpen the differences between units even further. Note that @b{SoftClLayerSpec} can be used with units using the @b{SoUnitSpec} to obtain a "plain" SoftMax function of the dot product net input to a unit.

- Reference info for type SomLayerSpec: 

-- the self-organizing map:

simply computes a sum-of-squares distance function of the activations and weights, like the SoftClUnitSpec, but it does not apply a Gaussian to this distance. The winning unit in the SOM formalism is the one with the weight vector closest to the current input activation state, so this unit provides the appropriate information for the layer specification to choose the winner. The SomLayerSpec provides a means of generating a "neighborhood kernel" of activation surrounding the winning unit in a layer. First, the unit whose weights are closest to the current input pattern is selected (assuming the SomUnitSpec is being used, and the netin_type is set to MIN_NETIN_WINS). Then the neighbors of this unit are activated according to the neighborhood kernel defined on the spec. The fact that neighboring units get partially activated is what leads to the development of topological "map" structure in the network. The shape and weighting of the neighborhood kernel is defined by a list of NeighborEl objects contained in the neighborhood member. Each of these defines one element of the kernel in terms of the offset in 2-D coordinates from the winning unit (off), and the activation value for a unit in this position (act_val). While these can be created by hand, it is easier to use one of the built-in functions on the SomLayerSpec.
Article Sources and Contributors

Self Organizing  Source: https://grey.colorado.edu/emergent/index.php?oldid=4554  Contributors: Brian.mingus, Oreilly