Problem Analysis – Self-organizing maps

A *self-organizing map* does not need an output target to be specified, unlike a great number of other types of networks. Instead of this, the area of the lattice in which the node weights match the input vector, needs to be selectively optimized for a closer resemblance of the data of the class where the input vector is a member of.

Over many iterations, from the initially considered distribution of random weights, the self-organizing map eventually settles into a map featuring stable zones. Each of these zones, can be identified effectively as a feature classifier, so you can think about the graphical output as a type of a feature map of the input space. If we take another look at the trained network shown in Fig. 1 from the *component (a)*, we observe that the blocks of similar color are actually representing the individual zones just discussed. Any new, previously unseen input vectors presented to the network will stimulate nodes in the zone with similar weight vectors.

Learning Algorithm

The purpose of learning in the self-organizing map, is causing different parts of the network to respond similarly to certain input patterns. The motivation for using this, is how auditory, visual or other sensory information is handled in separate parts of the cerebral cortex in the human brain.

Each neuron has a corresponding weight, each of them are initialized either to small random values or even sampled from the subspace spanned by the two largest principal component eigenvectors. Learning is much faster, because a good approximation of SOM weights is given by the initial weights. A large number of vectors that represent as good as possible, the type of vectors being expected during mapping, must be fed to the network. Usually, they're given many times as iterations.

Competitive learning is used in the training. Every time a training example is given to the network, the corresponding Euclidean distance to all weight vectors is computed. The neuron whose weight vector is the most similar to the input, is named the best matching unit (BMU). So, now the weights related to BMU and the neurons close to it in the grid of the self-organizing map will be adjusted towards the input vector. Magnitude of the change is decreasing together with time and grid-distance relating to BMU. The new formula for a neuron v with weight vector $W_v(s)$ is:

$$W_v(s+1) = W_v(s) + \theta(u, v, s) \cdot \alpha(s) \cdot \big(D(t) - W_v(s)\big),$$

where s is the step index, t an index into the training sample, u is the index of the BMU for the input vector $D(t)$, $\alpha(s)$ is a monotonically decreasing learning coefficient; $\theta(u, v, s)$ is the neighborhood function which gives the distance between the neuron u and the neuron v in step s. Depending on the implementations, t can scan the training data set systematically (t is $0, 1, 2 \ldots T - 1$, then repeat, T being the training sample's size), be randomly drawn from the data set (bootstrap sampling), or implement some other sampling method (such as jackknifing).

The neighborhood function $\theta(u, v, s)$ (also called function of lateral interaction) depends on the grid-distance between the BMU (neuron u) and neuron v. In the simplest form, it is 1 for all neurons close enough to BMU and 0 for others, but the Gaussian and Mexican-hat functions are common choices, too. Regardless of the functional form, the neighborhood function shrinks with time. At the beginning when the neighborhood is broad, the self-organizing takes place on the global scale. When the neighborhood has shrunk to just a couple of neurons, the weights are converging to local estimates. In some implementations, the learning coefficient α and the neighborhood function θ decrease steadily with increasing s, in others (in particular those where t scans the training data set) they decrease in step-wise fashion, once every T steps.

This process is repeated for each input vector for a – usually large – number of cycles λ. The network winds up associating output nodes with groups or patterns in the input data set. If these patterns can be named, the names can be attached to the associated nodes in the trained net. During mapping, there will be one single winning neuron: the neuron whose weight vector lies closest to the input vector. This can be simply determined by calculating the Euclidean distance between input vector and weight vector.

Because all this is theoretical information, the algorithm applied by me can be explained in some few simple steps which are iterated many times:

1. Weights are being initialized for every node.
2. A vector is randomly chosen from training dataset and presented to the lattice.
3. Every node is examined to calculate which corresponding weight is the most likely to be the input vector – the winning node is commonly known as the Best Matching Unit (BMU).
4. The neighborhood radius of the BMU is now calculated. This is a value that starts large, typically set to the lattice' *radius*, but decreases each step. Any nodes found within this radius are deemed to be inside the BMU's neighborhood.

5. Each neighboring node's (the nodes found in the previous step) weights are adjusted to make them more similar the input vector. The closer a node is to the BMU, the more its weights get altered.
6. Repeat step 2 for *n* iterations.