Self-Organizing Map*

Jason Brownlee
jasonb@CleverAlgorithms.com
The Clever Algorithms Project
http://www.CleverAlgorithms.com

November 24, 2010
Technical Report: CA-TR-20101124a-1

Abstract

The Clever Algorithms project aims to describe a large number of Artificial Intelligence algorithms in a complete, consistent, and centralized manner, to improve their general accessibility. The project makes use of a standardized algorithm description template that uses well-defined topics that motivate the collection of specific and useful information about each algorithm described. This report describes the Self-Organizing Map using the standardized algorithm description template.

Keywords: `Clever, Algorithms, Description, Self-Organizing, Map`

1 Introduction

The Clever Algorithms project aims to describe a large number of algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics in a complete, consistent and centralized manner [1]. The project requires all algorithms to be described using a standardized template that includes a fixed number of sections, each of which is motivated by the presentation of specific information about the technique [2]. This report describes the Self-Organizing Map using the standardized algorithm description template.

2 Name

Self-Organizing Map, SOM, Self-Organizing Feature Map, SOFM, Kohonen Map, Kohonen Network

3 Taxonomy

The Self-Organizing Map algorithm belongs to the field of Artificial Neural Networks and Neural Computation. More broadly it belongs to the field of Computational Intelligence. The Self-Organizing Map is an unsupervised neural network that uses a competitive (winner-take-all) learning strategy. It is related to other unsupervised neural networks such as the Adaptive Resonance Theory (ART) method. It is related to other competitive learning neural networks such as the the Neural Gas Algorithm, and the Learning Vector Quantization algorithm, which

is a similar algorithm for classification without connections between the neurons. Additionally, SOM is a baseline technique that has inspired many variations and extensions, not limited to the Adaptive-Subspace Self-Organizing Map (ASSOM).

4 Inspiration

The Self-Organizing Map is inspired by postulated feature maps of neurons in the brain comprised of feature-sensitive cells that provide ordered projections between neuronal layers, such as those that may exist in the retina and cochlea. For example, there are acoustic feature maps that respond to sounds to which an animal is most frequently exposed, and tonotopic maps that may be responsible for the order preservation of acoustic resonances.

5 Strategy

The information processing objective of the algorithm is to optimally place a topology (grid or lattice) of codebook or prototype vectors in the domain of the observed input data samples. An initially random pool of vectors is prepared which are then exposed to training samples. A winner-take-all strategy is employed where the most similar vector to a given input pattern is selected, then the selected vector and neighbors of the selected vector are updated to closer resemble the input pattern. The repetition of this process results in the distribution of codebook vectors in the input space which approximate the underlying distribution of samples from the test dataset. The result is the mapping of the topology of codebook vectors to the underlying structure in the input samples which may be summarized or visualized to reveal topologically preserved features from the input space in a low-dimensional projection.

6 Procedure

The Self-Organizing map is comprised of a collection of codebook vectors connected together in a topological arrangement, typically a one dimensional line or a two dimensional grid. The codebook vectors themselves represent prototypes (points) within the domain, whereas the topological structure imposes an ordering between the vectors during the training process. The result is a low dimensional projection or approximation of the problem domain which may be visualized, or from which clusters may be extracted.

Algorithm 1 provides a high-level pseudo-code for preparing codebook vectors using the Self-Organizing Map method. Codebook vectors are initialized to small floating point values, or sampled from the domain. The Best Matching Unit (BMU) is the codebook vector from the pool that has the minimum distance to an input vector. A distance measure between input patterns must be defined. For real-valued vectors, this is commonly the Euclidean distance:

$$dist(x, c) = \sum_{i=1}^{n} (x_i - c_i)^2 \tag{1}$$

where n is the number of attributes, x is the input vector and c is a given codebook vector.

The neighbors of the BMU in the topological structure of the network are selected using a neighborhood size that is linearly decreased during the training of the network. The BMU and all selected neighbors are then adjusted toward the input vector using a learning rate that too is decreased linearly with the training cycles:

$$c_i(t + 1) = learn_{rate}(t) \times (c_i(t) - x_i) \tag{2}$$

where $c_i(t)$ is the i^{th} attribute of a codebook vector at time t, $learn_{rate}$ is the current learning rate, an x_i is the i^{th} attribute of a input vector.

The neighborhood is typically square (called bubble) where all neighborhood nodes are updated using the same learning rate for the iteration, or Gaussian where the learning rate is proportional to the neighborhood distance using a Gaussian distribution (neighbors further away from the BMU are updated less).

Algorithm 1: Pseudo Code for the Self-Organizing Map algorithm.

Input: InputPatterns, $iterations_{max}$, $learn_{rate}^{init}$, $neighborhood_{size}^{init}$, $Grid_{width}$, $Grid_{height}$
Output: CodebookVectors

```
CodebookVectors ← InitializeCodebookVectors(Grid_width, Grid_height, InputPatterns);
for i = 1 to iterations_max do
    learn^i_rate ← CalculateLearningRate(i, learn^init_rate);
    neighborhood^i_size ← CalculateNeighborhoodSize(i, neighborhood^init_size);
    Pattern_i ← SelectInputPattern(InputPatterns);
    Bmu_i ← SelectBestMatchingUnit(Pattern_i, CodebookVectors);
    Neighborhood ← Bmu_i;
    Neighborhood ← SelectNeighbors(Bmu_i, CodebookVectors, neighborhood^i_size);
    foreach Vector_i ∈Neighborhood do
        foreach Vector_i^attribute ∈ Vector_i do
            Vector_i^attribute ← Vector_i^attribute + learn^i_rate × (Pattern_i^attribute −
            Vector_i^attribute)
        end
    end
end
return CodebookVectors;
```

7 Heuristics

- The Self-Organizing Map was designed for unsupervised learning problems such as feature extraction, visualization and clustering. Some extensions of the approach can label the prepared codebook vectors which can be used for classification.
- SOM is non-parametric, meaning that it does not rely on assumptions about that structure of the function that is is approximating.
- Real-values in input vectors should be normalized such that $x \in [0, 1)$.
- Euclidean distance is commonly used to measure the distance between real-valued vectors, although other distance measures may be used (such as dot product), and data specific distance measures may be required for non-scalar attributes.
- There should be sufficient training iterations to expose all the training data to the model multiple times.
- The more complex the class distribution, the more codebook vectors that will be required, some problems may need thousands.
- Multiple passes of the SOM training algorithm are suggested for more robust usage, where the first pass has a large learning rate to prepare the codebook vectors and the second pass has a low learning rate and runs for a long time (perhaps 10× more iterations).

- The SOM can be visualized by calculating a Unified Distance Matrix (U-Matrix) shows highlights the relationships between the nodes in the chosen topology. A Principle Component Analysis (PCA) or Sammon's Mapping can be used to visualize just the nodes of the network without their inter-relationships.
- A rectangular 2D grid topology is typically used for a SOM, although toroidal and sphere topologies can be used. Hexagonal grids have demonstrated better results on some problems and grids with higher dimensions have been investigated.
- The neuron positions can be updated incrementally or in a batch model (each epoch of being exposed to all training samples). Batch-mode training is generally expected to result in a more stable network.
- The learning rate and neighborhood size parameters typically decrease linearly with the training iterations, although non-linear functions may be used.

8 Code Listing

Listing 1 provides an example of the Self-Organizing Map algorithm implemented in the Ruby Programming Language. The problem is a feature detection problem, where the network is expected to learn a predefined shape based on being exposed to samples in the domain. The domain is two-dimensional $x, y \in [0, 1]$, where a shape is pre-defined as a square in the middle of the domain $x, y \in [0.3, 0.6]$. The system is initialized to vectors within the domain although is only exposed to samples within the pre-defined shape during training. The expectation is that the system will model the shape based on the observed samples.

The algorithm is an implementation of the basic Self-Organizing Map algorithm based on the description in Chapter 3 of the seminal book on the technique [5]. The implementation is configured with a 4×5 grid of nodes, Euclidean distance measure is used to determine the BMU and neighbors, a Bubble neighborhood function is used. Error rates are presented to the console, and the codebook vectors themselves are described before and after training. The learning process is incremental rather than batch, for simplicity.

An extension to this implementation would be to visualize the resulting network structure in the domain - shrinking from a mesh that covers the whole domain, down to a mesh that only covers the pre-defined shape within the domain.

```
def random_vector(minmax)
  return Array.new(minmax.length) do |i|
    minmax[i][0] + ((minmax[i][1] - minmax[i][0]) * rand())
  end
end

def initialize_vectors(domain, width, height)
  codebook_vectors = []
  width.times do |x|
    height.times do |y|
      codebook = {}
      codebook[:vector] = random_vector(domain)
      codebook[:coord] = [x,y]
      codebook_vectors << codebook
    end
  end
  return codebook_vectors
end

def euclidean_distance(v1, v2)
  sum = 0.0
  v1.each_with_index do |v, i|
```

```
    sum += (v1[i]-v2[i])**2.0
  end
  return Math.sqrt(sum)
end

def get_best_matching_unit(codebook_vectors, pattern)
  best, b_dist = nil, nil
  codebook_vectors.each do |codebook|
    dist = euclidean_distance(codebook[:vector], pattern)
    best,b_dist = codebook,dist if b_dist.nil? or dist<b_dist
  end
  return [best, b_dist]
end

def get_vectors_in_neighborhood(bmu, codebook_vectors, neigh_size)
  neighborhood = []
  codebook_vectors.each do |other|
    if euclidean_distance(bmu[:coord], other[:coord]) <= neigh_size
      neighborhood << other
    end
  end
  return neighborhood
end

def update_codebook_vector(codebook, pattern, lrate)
  codebook[:vector].each_with_index do |v,i|
    error = pattern[i]-codebook[:vector][i]
    codebook[:vector][i] += lrate * error
  end
end

def train_network(codebook_vectors, shape, iterations, learning_rate, neighborhood_size)
  iterations.times do |iter|
    pattern = random_vector(shape)
    lrate = learning_rate * (1.0-(iter.to_f/iterations.to_f))
    neigh_size = neighborhood_size * (1.0-(iter.to_f/iterations.to_f))
    bmu,dist = get_best_matching_unit(codebook_vectors, pattern)
    neighbors = get_vectors_in_neighborhood(bmu, codebook_vectors, neigh_size)
    neighbors.each do |node|
      update_codebook_vector(node, pattern, lrate)
    end
    puts ">training: neighbors=#{neighbors.size}, bmu_dist=#{dist}"
  end
end

def summarize_vectors(vectors)
  minmax = Array.new(vectors.first[:vector].length){[1,0]}
  vectors.each do |c|
    c[:vector].each_with_index do |v,i|
      minmax[i][0] = v if v<minmax[i][0]
      minmax[i][1] = v if v>minmax[i][1]
    end
  end
  s = ""
  minmax.each_with_index {|bounds,i| s << "#{i}=#{bounds.inspect} "}
  puts "Vector details: #{s}"
end

def test_network(codebook_vectors, shape)
  error = 0.0
  100.times do
    pattern = random_vector(shape)
    bmu,dist = get_best_matching_unit(codebook_vectors, pattern)
```

```
    error += dist
  end
  error /= 100.0
  puts "Finished, average error=#{error}"
end

def run(domain, shape, iterations, learning_rate, neighborhood_size, width, height)
  codebook_vectors = initialize_vectors(domain, width, height)
  summarize_vectors(codebook_vectors)
  train_network(codebook_vectors, shape, iterations, learning_rate, neighborhood_size)
  test_network(codebook_vectors, shape)
  summarize_vectors(codebook_vectors)
end

if __FILE__ == $0
  # problem definition
  domain = [[0.0,1.0],[0.0,1.0]]
  shape = [[0.3,0.6],[0.3,0.6]]
  # algorithm parameters
  iterations = 100
  learning_rate = 0.3
  neighborhood_size = 5
  width, height = 4, 5
  # execute the algorithm
  run(domain, shape, iterations, learning_rate, neighborhood_size, width, height)
end
```

Listing 1: Self-Organizing Map algorithm in the Ruby Programming Language

9 References

9.1 Primary Sources

The Self-Organizing Map was proposed by Kohonen in 1982 in a study that included the mathematical basis for the approach, summary of related physiology, and simulation on demonstration problem domains using one and two dimensional topological structures [3].

9.2 Learn More

Kohonen provides a detailed introduction and summary of the Self-Organizing Map in a journal article [4]. Kohonen, et al. provide a practical presentation of the algorithm and heuristics for configuration in the technical report written to accompany the released SOM_PAK implementation of the algorithm for academic research [6]. The seminal book on the technique is "Self-Organizing Maps" by Kohonen, which includes chapters dedicated to the description of the basic approach, physiological interpretations of the algorithm, variations, and summarizes of application areas [5].

10 Conclusions

This report described the Self-Organizing Map using the standardized algorithm description template.

11 Contribute

Found a typo in the content or a bug in the source code? Are you an expert in this technique and know some facts that could improve the algorithm description for all? Do you want to get

that warm feeling from contributing to an open source project? Do you want to see your name as an acknowledgment in print?

Two pillars of this effort are i) that the best domain experts are people outside of the project, and ii) that this work is subjected to continuous improvement. Please help to make this work less wrong by emailing the author 'Jason Brownlee' at jasonb@CleverAlgorithms.com or visit the project website at http://www.CleverAlgorithms.com.

References

[1] Jason Brownlee. The clever algorithms project: Overview. Technical Report CA-TR-20100105-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[2] Jason Brownlee. A template for standardized algorithm descriptions. Technical Report CA-TR-20100107-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[3] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[4] T. Kohonen. The self-organizing maps. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[5] Teuvo Kohonen. *Self-Organizing Maps*. Springer, 1995.

[6] Teuvo Kohonen, Jussi Hynninen, Jari Kangas, and Jorma Laaksonen. $SOM_PAK : The self-organizing map program package. Technical Report A31, Helsinki University of Technology, Laboratory of C$