2020

Machine Learning

Color SOM on C++

User Manual

Applied Computational Intelligence, M. Sc. 1

Emanuel Bîscă

What is a Self-organizing Map and how can we use it?

Analyzing high dimensional data can be very difficult because we can only visualize data in 2 or 3 dimensions. Before and after training machine learning models on large high-dimensional datasets, it is desirable to have some intuitive understanding of the relationships and patterns in a dataset to guide the learning process and to help interpret results.

The Self-Organizing Map is a dimensionality reduction technique that can give us insights about high dimensional data with minimal required computing. Self-Organizing Maps can be used for exploratory data analysis, clustering problems, and visualization of high dimensional datasets.

SOMs were invented by Teuvo Kohonen in 1980 and have been the subject of many research papers. SOMs are built using a pre-defined $2D$ lattice of nodes. This lattice of nodes has a structure that is defined by giving each node a location in $\mathbb{R}^2$, represented as a vector l_i where i is the index of that node. A visual of the lattice space can be seen in Fig. 1 on the left. Each node in the lattice also has a position in the input data space, represented as a vector $w_i \in \mathbb{R}^d$ where d is the dimension of the input data. Each lattice node is a connection between $\mathbb{R}^d$ and $\mathbb{R}^2$. Fig. 1 shows how the structure of the lattice is preserved in a higher dimensional space.

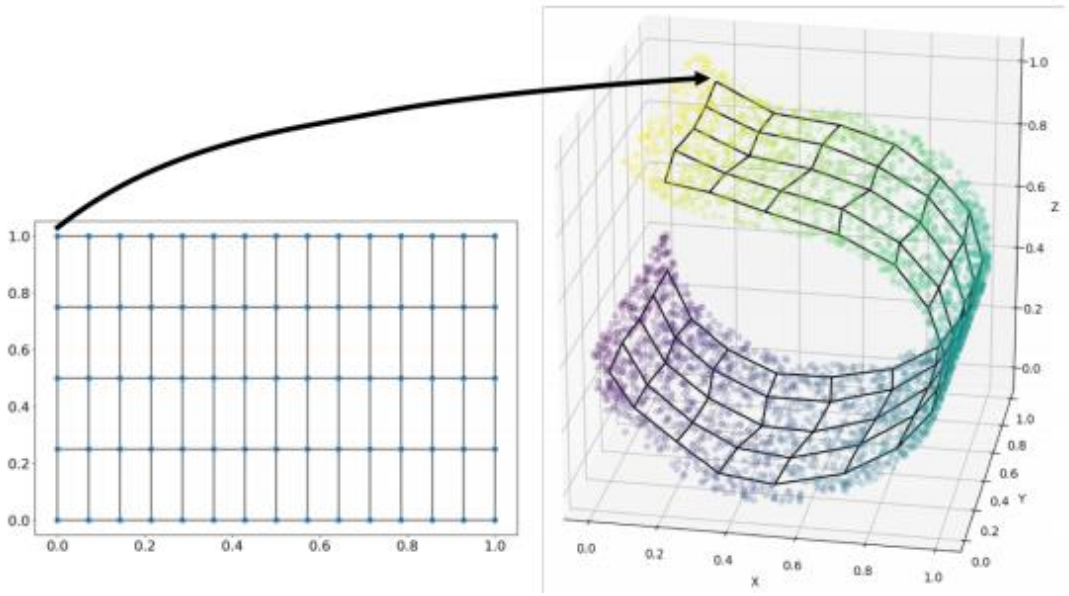

Fig. 1

On the left: A plot of a 15×5 lattice in the lattice space, $\mathbb{R}^2$, the black lines indicate neighbors and the blue points indicate the location of the node in lattice space. On the right: a plot in the input data space, $\mathbb{R}^3$, of a SOM trained with the lattice on the left, fit to $3D$ data that has a non-linear relationship. The circles are data points, the color of each data point is its position along a non-linear manifold. The arrow shows the connection between a node's lattice vector at $(1, 0)$ in the lattice space and weight vector at $(0.17, 0.88, 0.94)$ in the data space.

Training of a Self-Organizing Map

Algorithm 1: Main Idea Behind the SOM Training Algorithm

```
initialize lattice nodes;
initialize weight vectors;
N ← Iteration count;
for i ← 1 to N do
    x ← pick a random point in the dataset;
    c ← the lattice node closest to x;
    move the weight vector of c closer to x;
    move the weight vectors of the neighbors of c slightly closer to x;
end
```

The desired 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.

More information about that can be found in the other documents accompanying this project.

Applications of Self-organizing Maps

SOMs have been applied in many areas. Here are just some of them.

- Bibliographic classification,
- Image browsing systems,
- Medical Diagnosis
- Interpreting seismic activity
- Speech recognition
- Data compression
- Separating sound sources
- Environmental modelling

The Experiment

The aim of this project was to give an insight of the main ideas behind how SOMs are mapping the colors from their three-dimensional components - red, green and blue, into two dimensions.

For my project, I used the Code::Blocks IDE, the 20.03 version, with GNU GCC compiler. The code was implemented in C++.

Although a color is rendered by the computer using values for each component (red, green and blue) from 0 to 255, the input vectors have been adjusted so that each component has a value between 0 and 1. (This is to match the range of the values used for the node's weights).

We can choose to use the training set shown in Fig. 1 or a training set made up of random colors by uncommenting the line `"//#define RANDOM_TRAINING_SETS"` found in `"constants.h"` file. It's obvious that different values for the number of iterations and the initial learning rate would affect the algorithm. For my experiment I considered the number of epochs desired for the training to be 1000, while the value of the learning rate at the start of training is 0.1.