Brief Design Documentation – Self-organizing Map on C++

- **The weights initializing**

Before any training step, each node corresponding weights need to be initialized. Typically, these will be set to small standardized random values. The weights in the SOM accompanying my project are initialized so that $0 < w < 1$. Nodes are defined in the source code by the class `CNode`. Here are the relevant parts of that class:

```
class CNode
{
private:
  //this node's weights
  vector<double>      m_dWeights;
  //its position within the lattice
  double              m_dX,
                      m_dY;
  int                 m_iLeft;
  int                 m_iTop;
  int                 m_iRight;
  int                 m_iBottom;
public:
  CNode(int lft, int rgt, int top, int bot, int NumWeights):m_iLeft(lft),
                                                            m_iRight(rgt),
                                                            m_iBottom(bot),
                                                            m_iTop(top)
  {
    //initialize the weights to small random variables
    for (int w=0; w<NumWeights; ++w)
    {
      m_dWeights.push_back(RandFloat());
    }
    //calculate the node's center
    m_dX = m_iLeft + (double)(m_iRight - m_iLeft)/2;
    m_dY = m_iTop  + (double)(m_iBottom - m_iTop)/2;
  }
```

As it can be seen the weights are initialized automatically by the constructor when a `CNode` object is instantiated. The member variables `m_iLeft`, `m_iRight`, `m_iTop` and `m_iBottom` are only used to render the network to the display area – each node is represented by a rectangular cell described by these values.

- **Calculating the Best Matching Unit – BMU**

In order to find out the best matching unit, one method would be to iterate through all the nodes and calculate the *Euclidean distance* between the current input vector each node's weight vector. The node with the closest weight vector to the input vector is going to be flagged as the BMU.

The *Euclidean distance* is defined as below:

$$\sqrt{\sum_{i=0}^{n}(V_i - W_i)^2} \quad (1)$$

where V is the current input vector and W is the node's weight vector. In the code this equation translates to:

```
double CNode::GetDistance(const vector<double> &InputVector)
{
  double distance = 0;
  for (int i=0; i<m_dWeights.size(); ++i)
  {
    distance += (InputVector[i] - m_dWeights[i]) * (InputVector[i] - m_dWeights[i]);
  }
  return sqrt(distance);
}
```

- **Studying the BMU's local neighborhood**

Once we determine the BMU, the next step is to calculate which of the other nodes are within the BMU's neighborhood. All these nodes will have their weight vectors altered in the next step. Firstly, we need to calculate the value of what the neighborhood radius should be and then it's a simple application of Pythagoras to determine if each node is within the radial distance or not. **Fig. 4** shows an example of the size of a typical neighborhood close to the commencement of training.

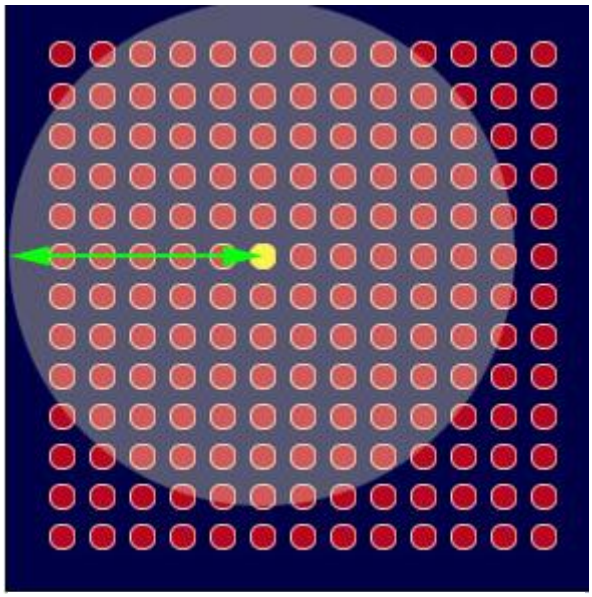

Fig. 1 BMU Neighborhood

It is easy to observe, that the neighborhood shown above is centered around the BMU (yellow colored) and encompasses most of the other nodes. The green arrow shows the radius.

A unique feature of the Kohonen learning algorithm is that the area of the neighborhood shrinks over time. This is accomplished by making the radius of the neighborhood shrink over time. The following exponential decay function is responsible for this feature:

$$\sigma(t) = \sigma_0 e^{\left(-\frac{t}{\lambda}\right)}, \quad t = 1, 2, 3, \ldots \tag{2}$$

where σ_0 denotes the width of the lattice at the time t_0, while λ denotes a time constant, t is the current time-step (basically the iteration of the loop). We have `m_dMapRadius` as the σ and it will be equal to σ_0 in the beginning of training. It is computed as it follows:

```
m_dMapRadius = max(constWindowWidth, constWindowHeight)/2;
```

The value of λ is dependent on σ and the number of iterations chosen for the algorithm to run. There is `m_dTimeConstant` as the λ and it is calculated by the line:

```
m_dTimeConstant = m_iNumIterations/log(m_dMapRadius);
```

`m_iNumIterations` is the number of iterations the learning algorithm will perform; it looks as it follows:

```
m_dNeighbourhoodRadius = m_dMapRadius * exp(-(double)m_iIterationCount/m_dTimeConstant);
```

We can use `m_dTimeConstant` and `m_dMapRadius` to calculate the neighborhood radius for each iteration of the algorithm using equation (2).

The next figure shows how the neighborhood in **Fig. 1** decreases over time.

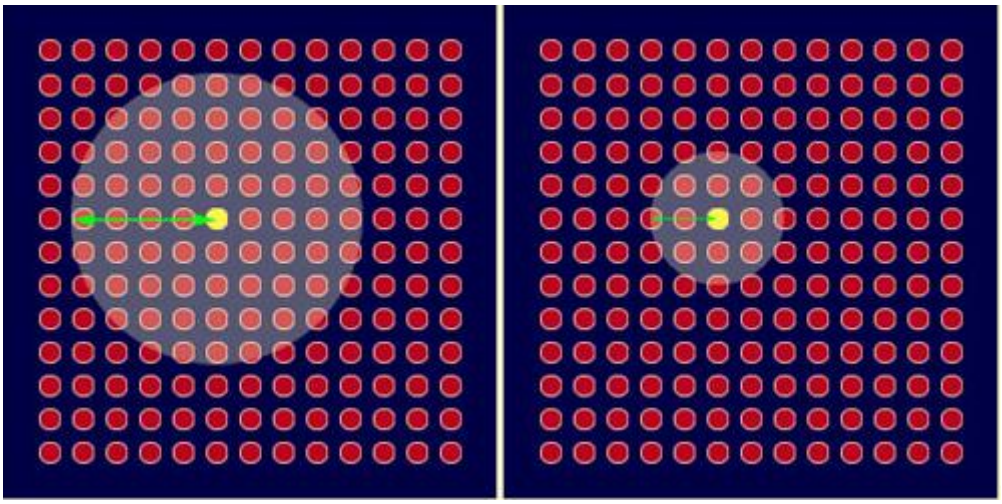

Fig. 2 The shrinking radius

As a remark, the figure is drawn assuming the neighborhood remains centered on the same node, in practice the BMU will move around according to the input vector being presented to the network.

Over time the neighborhood will shrink to the size of just one node... the BMU.

Now we know the radius, it's a simple matter to iterate through all the nodes in the lattice to determine if they lay within the radius or not. If a node is found to be within the neighborhood then its weight vector needs adjustments.

- **Weights adjustments**

Every node within the BMU's neighborhood (BMU included, of course) has its weight vector adjusted according to the following equation:

$$W(t+1) = W(t) + L(t)\big(V(t) - W(t)\big) \tag{3}$$

Where t represents the time-step and L is a small variable called the learning rate, which decreases with time. Basically, what this equation is saying, is that the new adjusted weight for the node is equal to the old weight (W), plus a fraction of the difference (L) between the old weight and the input vector (V).

The decay of the learning rate is calculated each iteration using the following equation:

$$L(t) = L_0 e^{\left(-\frac{t}{\lambda}\right)}, \quad t = 1, 2, 3, \ldots \tag{4}$$

It easy to notice this is the same as the exponential decay function described in equation (2), except this time we're using it to decay the learning rate. In code it looks like this:

```
m_dLearningRate = constStartLearningRate * exp(-(double)m_iIterationCount/m_iNumIterations);
```

The learning rate at the start of training, `constStartLearningRate`, is set in the constants.h file as 0.1. It then gradually decays over time so that during the last few iterations it is close to zero.

We need to remarks something very interesting, not only does the learning rate have to decay over time, but also, the effect of learning should be proportional to the distance a node is from the BMU. Indeed, at the edges of the BMUs neighborhood, the learning process should have barely any effect at all! Ideally, the amount of learning should fade over distance similar to the Gaussian decay shown in **Fig. 3**.

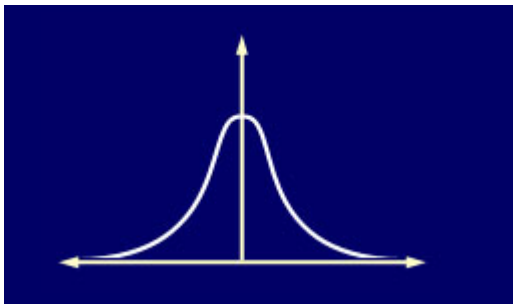

Fig. 3 The Gaussian decay

To achieve this, all it takes is a slight adjustment to equation (3).

$$W(t+1) = W(t) + \theta(t)L(t)\big(V(t) - W(t)\big) \tag{5}$$

where θ, to represent the amount of influence a node's distance from the BMU has on its learning. $\theta(t)$ is given by

$$\theta(t) = \exp\left(-\frac{\text{dist}^2}{2\sigma^2(t)}\right), t = 1, 2, 3, \ldots \tag{6}$$

Where dist is the distance a node is from the BMU and σ, is the width of the neighbourhood function as calculated by equation 2. Additionally, please note that θ also decays over time.