Neural Network Approach To Solving The Traveling Salesman Problem

The Traveling Salesman

- The shortest route for a salesman to visit every city, without stopping at the same city twice.

Figure 1-1: Map of Germany from which latitude and longitude were manually measured.

Methods

- Random
  - An algorithm must be better than this to be worthwhile
- Continuous Hopfield Network
  - Fully-Connected
  - Self-Associative
- Kohonen Self-Organizing Map
  - Topologically Preserving

The Hopfield Net

- Created by James Hopfield, originally published in 1982.
- Self-Associative.
- Single-Layer Network.
- Is allowed to run until it stabilizes.
- Training data represents "attractor states"
- Output is binary.

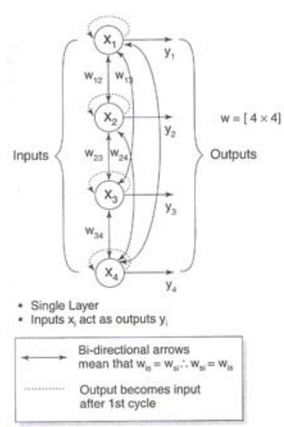
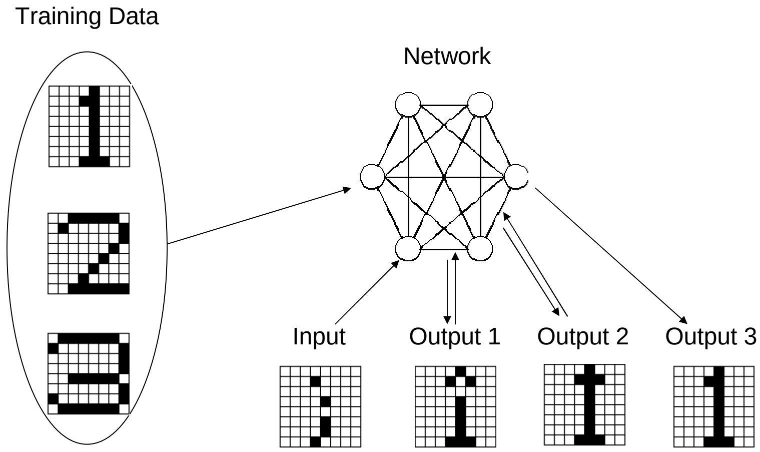


FIGURE 3.9.10. A four-node Hopfield autoassociative neural net.

A Simple Hopfield Example



The Math

- One node per characteristic (pixels in our example).
- The net is initialized (i.e. trained) to a square weight matrix.
- The entry at position X,Y is the value of the weight from X to Y.
- If for all nodes X and Y, entry X,Y = entry Y,X implies that the net WILL stabilize.

The Math + Pascal

- $T_{i,j}$ is the weight from node i to node j.
- $T_{i,j}$=0 if i=j.
- Ti,j= $\sum_{s=1}^{M} x_i^s \cdot x_j^s$ If i does not equal j, where x_i^s is element i in training example s.

```
procedure assign_connection_weights;
var sum,i,j,s : integer;
begin for i:=1 to PATTERN_LENGTH do
      for j:=1 to PATTERN_LENGTH do
       if i = j
        then t[i,j]:=0
        else begin sum:=0;
                   for s:=1 to MAX_CLASSES do
                    sum:=sum + X[s,i] * X[s,j];
                   t[i,j]:=sum
             end
end;
```

Running The Network

- We have a 2-dimensional array m.
- M(j,t) = the value of output j at time t.
- Initialize M(j,0) for all j to be our input.
- The weights in T are multiplied by the corresponding values in M, summed together. This is the new value stored in M.
- Remember: in basic Hopfield networks values in M and T are binary.

More Pascal Psuedocode

```
const MAX_TIME = 10;   {Maximum no. of time slots before
    convergence}

var mu : array [1..MAX_TIME,1..PATTERN_LENGTH] of integer;

procedure copy_input_pattern;
var i : integer;
begin for i:=1 to PATTERN_LENGTH do
      mu[0,i]:=input_pattern[i]      {Each element +1 or -1}
end;
```

More Pascal Psuedocode

```
procedure iterate;
var tt : integer;     {Time slot}
   i,j : integer;    {General loop variables}
   sum : integer;
begin for tt:=1 to MAX_TIME do
      begin for j:=1 to PATTERN_LENGTH do
            begin sum:=0;
                  for i:=1 to PATTERN_LENGTH do
                   sum:=sum + t[i,j] * mu[tt-1,i];
  {Now pass sum through the hard-limiter, so it is 1 or -1}
                  if sum > 0
                   then mu[tt,j]:=1
                   else mu[tt,j]:=-1
            end
      end
end;
```

Hopfield As Applied To The Traveling Salesman

1. Initialize all units according to "The Willshaw Initialization"
   - Cities on opposite sides of the map should be placed on opposite sides of the tour:
   - Bias in terms of the ith city and the jth position, with coordinates x_i and y_i

$$\mu = bias(i, j) = \cos(\arctan(\frac{y_i - 0.5}{x_i - 0.5}) + \frac{2\pi(j-1)}{n})\sqrt{(x_i - 0.5)^2 + (y_i - 0.5)^2}$$

Continuous Hopfield cont.

2. We will perform steps 3-7 until our net stabilizes.
3. Perform steps 4-6 n^2 times, where n is the number of cities.
4. Choose a node at random.
5. Update M for this time step on the selected unit.

Continuous Hopfield cont.

6. Apply the output function to see how close this node is to a city, potentially fixing its location.
7. Check for stabilization

- Use a square matrix
  – Rows correspond to cites
  – Columns correspond to a cities place in the tour

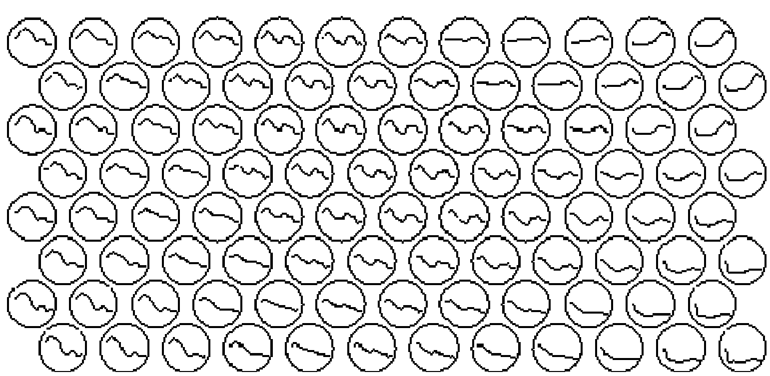
Kohonen Self-Organizing Map (SOM)

- Unsupervised learning artificial neural network.
- Is known to perform well on classification problems.
- Commonly used for:
  – Visualization of statistical data, analysis of electrical signals from the brain, cloud classification from satellite, clinical voice analysis, and automatic speech recognition.

Some Important Characteristics of Self Organizing Maps

- Topography preserving.
  - Keeps relationships with other nodes intact.
  - Hopfield is fully connected.
- Similar to the brain in which neurons in the same cluster have a stronger connection than to those outside of the cluster.
- No other artificial neural network has this property.

Finnish Phonetics



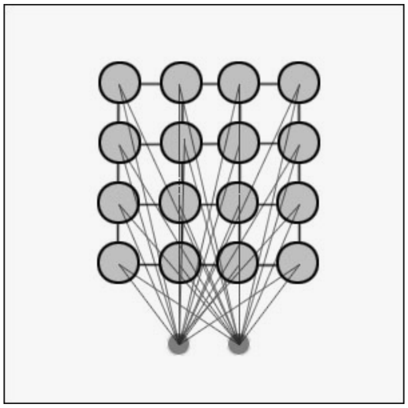
Network Structure

•Inputs are connected to all neurons

•Neurons are NOT connected to each other.

•Neurons DO contain information pertinent to their topographical location.

•Neurons, as usual, do contain weights.



Training

- Initialize each nodes weights.
- Choose an element from the training set.
- Choose a Best Matching Unit (BMU).
- Find nodes close to the BMU and update them to be more like the BMU.
- Repeat.

Initialization Code

```
class CNode
{
private:

  //this node's weights
  vector<double> m_dWeights;

  //its position within the lattice
  double m_dX, m_dY;

  //the edges of this node's cell. Each node, when
  //draw to the client
  //area, is represented as a rectangular cell. The
  //colour of the cell
  //is set to the RGB value its weights represent.
  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;
  }
  ...
};
```

Finding The BMU

- Euclidean distance is commonly used (V is the current input vector and W is the node's weight vector)

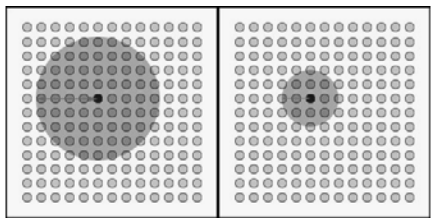
$$Dist = \sqrt{\sum_{i=0}^{i=n} (V_i - W_i)^2}$$

```
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;
  }
  ...
};
```

The BMU's Neighborhood

- Look at all neurons within a certain radius of the BMU.
- The radius decreases the longer the net has been run.



$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1, 2, 3...$$

Sigma-0 denotes the width of the net at time t-0. Lambda is a time constant.

Updating The Nodes

- Every node has its weight vector updating according to the following equation.
  - t : time step
  - L(t): learning rate at time t $L(t) = L_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1, 2, 3...$
  - W(t): weight at time t
  - V: input vector
  - theta(t): "proportionalizes" the effect of the learning rate. $\Theta(t) = \exp\left(-\frac{dist^2}{2\sigma^2(t)}\right) \qquad t = 1, 2, 3...$
- W(t+1)=W(t)+L(t)(V(t)-W(t))

Some code?

```
bool Csom::Epoch(const vector<vector<double> > &data)
{
  //make sure the size of the input vector
  //matches the size of each node's
  //weight vector
  if (data[0].size() != constSizeOfInputVector) return false;

  //return if the training is complete
  if (m_bDone) return true;

  //enter the training loop
  if (--m_iNumIterations > 0)
{
   //chose a vector at random from the
   //training set to be
   //this time-step's input vector
   int ThisVector = RandInt(0, data.size()-1);

   //present the vector to each node and determine
   //the BMU
   m_pWinningNode =
FindBestMatchingNode(data[ThisVector]);
 //calculate the width of the neighbourhood for this timestep
   m_dNeighbourhoodRadius = m_dMapRadius * exp(-
(double)m_iIterationCount/m_dTimeConstant);

//Now to adjust the weight vector of the BMU and its
//neighbours. For each node calculate the m_dInfluence
//(Theta from equation 6 in the tutorial. If it is greater than
//zero adjust the node's weight accordingly
```

```
for (int n=0; n<m_SOM.size(); ++n)
   {
     //calculate the Euclidean distance (squared) to this node
     //from the BMU
     double DistToNodeSq = (m_pWinningNode->X()-
m_SOM[n].X()) * (m_pWinningNode->X()-m_SOM[n].X()) +
(m_pWinningNode->Y()-m_SOM[n].Y()) *
(m_pWinningNode->Y()-m_SOM[n].Y());

     double WidthSq = m_dNeighbourhoodRadius *
m_dNeighbourhoodRadius;

     //if within the neighbourhood adjust its weights
     if (DistToNodeSq < (m_dNeighbourhoodRadius *
m_dNeighbourhoodRadius))
     {
       //calculate by how much its weights are adjusted
       m_dInfluence = exp(-(DistToNodeSq) / (2*WidthSq));
       m_SOM[n].AdjustWeights(data[ThisVector],
                         m_dLearningRate,
                         m_dInfluence);
     }
   }//next node
  //reduce the learning rate
   m_dLearningRate = constStartLearningRate * exp(-
(double)m_iIterationCount/m_iNumIterations);
   ++m_iIterationCount;
  }
  else  { m_bDone = true;  }
  return true;
}
```

Applying Kohonen To The Traveling Salesman

For setup:

- Place a “neuron” at each town on the map.
- Place a second set, of cardinality greater than or equal to the number of towns, of neurons in a circular formation around the first set of neurons.
  - These neurons will be stored in a 1-dimensional array.

Running The Network

- Repeatedly present a town-neuron and it's weights to the other neurons.
- Find and update the BMU.
- Update all nodes around the BMU.
- Run until we converge to a path.

An Example Result

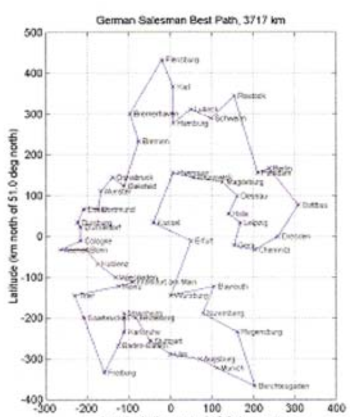


Figure 3-4: The shortest path obtained in 100 runs by the Kohonen Self-Organizing Map.

References

- Neural Network Approach To Solving The Traveling Salesman Problem. Ralph Reilly and Plamen Tchimev.
- The Self-Organizing Map. Teuvo Kohonen.
- Game Programming Gems – A Neural Network Primer. Andre LaMothe.
- http://richardbowles.tripod.com/neural/hopfield/hopfield.htm Richard Bowles
- http://www.ai-junkie.com/ann/som/som1.html
- http://www.comp.nus.edu.sg/~pris/AssociativeMemory/HopfieldModel.html