Outline

- Growing Cell Structures (GCS)
- Comparison between k-means, SOM and GCS
- Learning Vector Quantization (LVQ)
  - LVQ1
  - LVQ2.1

Growing Cell Structures (GCS)

  - “Growing Cell Structures – a Self-Organizing Network for Unsupervised and Supervised Learning”
  - “Some Competitive Learning Methods”

GCS - Algorithm

1. Choose the dimensionality of the simplex \( m \)
   - \( 1 \) – line, \( 2 \) – triangle (typically), \( 3 \) – tetrahedron, higher – hypertetrahedron
   - each simplex contains \( m+1 \) cells (competitive neurons)

2. Initialize the weight vectors for these neurons (typically: to the first \( m \) examples)

3. Input vector presentation – \( x(t) \)
4. Distance calculation

   \[ d_j = \sum_{i=1}^{n} (x_{i}(t) - w_{ij}(t))^2 \]

5. Determine the winning unit \( j^* \)
6. Increment the local counter \( E \) of \( j^* \)
7. Adapt the weights for \( j^* \) and its direct topological neighbors
   - \( w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_{i}(t) - w_{ij}(t)) \)
   - winner \( j^* \)
   - \( w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_{i}(t) - w_{ij}(t)) \)
     direct neighbors \( j \), i.e. \( j \) in neighborhood of \( j^* \)
GCS – Algorithm (cont. 2)

8. If the number of input vectors applied is an integer multiple of a user specified parameter $\lambda$, insert a new neuron
   - Determine the neuron $q$ with maximum local counter
   - Insert a new node $r$ by splitting the longest edge emanating from $q$ (let this is the edge to the node $f$).
   - Insert the connections $q(r$ & $r)f$ and remove the original $(q,f)$.
   - Connect the new node $r$ with all common neighbors of $q$ and $f$.
   - Interpolate the weight vector of $r$ from the weight vectors of $q$ and $f$:
     \[ w_r = \left( w_q + w_f \right) / 2 \]

GCS – Algorithm (cont. 3)

- Redistribute the local counters of all neighbors $j$ of the winner $j^*$ to donate fractions of their value for the counter of the new node $r$:
  - Donation of each neighboring neuron $j$:
    \[ \Delta E_j = \frac{1}{|E|} \sum_{j \in E} \left( f_j - r \right) \]
  - Local counter of $r$ is set to the total value of the donation:
    \[ E_r = \sum_{j \in E} \Delta E_j \]
- At the end of each epoch check the stopping criteria (network size or error measure, and/or maximum number of epochs). If not satisfied, go to 3.
- Label the network

Growing Cell Structures - Discussion

- Why do we insert a new neuron close to the neuron which was a winner most often?
  - Our objective is a structure with neurons (weight vectors) that are distributed according to the probability distribution of the input vectors but is achieved when every neuron has the same probability of being winner for the current input.
  - We don’t know what is the probability distribution of the input vectors; the local variables of each neuron are an estimate of it.
  - Instead of local variables corresponding to how many times a neuron was a winner, other local measures can be used, e.g. the error between the input vector and the winner.
  - In general - the local measure should be something which one is interested to reduce and which is likely to be reduced by an insertion of new units.

GCS, K-Means and SOM - Comparison

- Which of them define a mapping from a high dimensional space to a lower dimensional space?
  - What is the high and the lower dimensional space?
- Which of them are capable to learn data distribution?
  - What does to learn data distribution mean?
- Which of them require the number of clusters to be specified in advance?
  - How does the number of clusters relate to the number of competitive neurons in SOM and GCS and seeds in k-means?
- Which of them are hard-competitive learning algorithms and which are soft competitive learning algorithms?
  - Which of them can learn topology?
    - What does to learn topology mean?
    - Why is it useful?

K-means, SOM and GCS – Comparative Example

- From Fritzke, Some Competitive Learning Methods
  http://www.ki.inf.tu-dresden.de/~fritzke/research/incremental.html
- First picture: Simulation sequence for a ring-shaped uniform probability distribution
- Second picture: Simulation results after 4000 input examples (iterations) for 3 different probability distributions.

K-means
Learning Vector Quantization (LVQ)

- LVQ is a hybrid network – uses both supervised and unsupervised learning
  - competitive layer (1st layer)
  - 2nd layer – typically linear
- LVQ network – example:

W1 matrix

- W1 – weight matrix between the input and competitive layers
- Specifies the positions of the initial codebook vectors
- Each row of W1 – one codebook vector
- At the beginning W1 is initialized to small random weights or to the first k input vectors (k = number of competitive neurons)
- The output of the 1st layer is exactly like in SOM – the neuron which is closest to the input vector will output 1, the other neurons - 0

LVQ’s Architecture in Matlab

- Each neuron in the first layer learns a prototype vector (like in SOM)
  - Like in SOM: W1 represents the positions of the initial codebook vectors
  - Each row of W1 – one codebook vector
  - At the beginning W1 is initialized to small random weights or to the first k input vectors (k = number of competitive neurons)
  - The output of the 1st layer is exactly like in SOM – the neuron which is closest to the input vector will output 1, the other neurons - 0
- However, there is a difference in the interpretation of the winning neuron
  - In SOM it represents the class the input belongs to
  - In LVQ – a subclass rather than a class;
  - There might be several different neurons (subclasses) which make up each class
###LVQ – Second Layer

- The second layer of the LVQ network combines subclasses into a single class.
  - The columns of $W_2$ matrix between the competitive and output layer represent subclasses, and the rows - classes.
  - $W_2$ has a single 1 in each column, with the other elements set to 0; the row in which the 1 occurs indicates which class the appropriate subclass belongs to.
  - Example (refer to the LVQ picture):
    - Subclasses 1, 3 and 4 belong to class 1.
    - Subclass 2 belongs to class 2.
    - Subclasses 5 and 6 belong to class 3.

###LVQ – Second Layer (cont)

- Before learning can occur, each node in the 1st layer is assigned to an output neuron.
  - Node in the 1st layer = codebook vector = weight vector = prototype.
- Once $W_2$ is defined, it is never changed!
- How many subclasses do we need to assign for a class?
  - To ensure that each class is assigned an appropriate amount of competitive neurons, the number of subclasses is set proportional to the number of input vectors in each class.
- As at least one or (typically) several neurons of the 1st layer are assigned to the same class (i.e., one neuron in the second layer) ⇒ the number of neurons in the 1st layer is at least as large as the number of neurons in the 2nd layer (usually larger).

###LVQ – Decision Boundaries

- A single-layer competitive network (classical competitive, SOM) can create convex classification regions.
- The second layer of the LVQ network can combine the convex regions to create more complex boundaries.

###LVQ Learning

- LVQ is supervised learning - combines competitive learning with supervision.
- It requires a training set of examples of proper network behavior: 
  \[ \{p_1, t_1 \}, \{p_2, t_2 \}, \ldots, \{p_Q, t_Q \} \]
- Initialization of $W_1$ and $W_2$:
  - $W_1$ represents the positions of the competitive neurons; at the beginning it is initialized to small random weights or to the first k input vectors (k-number of competitive neurons).
  - $W_2$ represents which sub-classes are connected to which classes; each weight is initialized to either 0 or 1.
- At each iteration the input vector $p$ is presented and the distance from $p$ to each prototype is computed.
- The 1st layer neurons compete and the neuron $i^*$ wins the competition. 
  - Its output $a_{i^*}$ is non zero, indicating that $p$ is being assigned to class $k^*$.
- If the input vector $p$ is classified correctly, then the winning weight vector $i^*$ is moved toward the input vector according to the Kohonen ‘s rule:
  \[
  w_i(t) \leftarrow w_i(t-1) + \eta \cdot \mathbf{p}(t) - w_i(t-1) 
  \]
- If the input vector $p$ is classified incorrectly, then the winning weight vector $i^*$ is moved away from the input vector:
  \[
  w_i(t) \leftarrow w_i(t-1) - \eta \cdot \mathbf{p}(t) - w_i(t-1) 
  \]
- No change of the weights for the non-winning neurons.

###LVQ Learning – cont.

####LVQ Learning – cont.

- The output vector $a_1$ at the competitive layer is multiplied by $W_2$ (the matrix between the competitive and output layer) to get the final net output $a_2$.
- Again, only 1 output $k^*$ is non zero, indicating that $p$ is being assigned to class $k^*$.
- If the input vector $p$ is classified correctly, then the winning weight vector $i^*$ is moved toward the input vector according to the Kohonen ‘s rule:
  \[
  w_i(t) \leftarrow w_i(t-1) + \eta \cdot \mathbf{p}(t) - w_i(t-1) 
  \]
- If the input vector $p$ is classified incorrectly, then the winning weight vector $i^*$ is moved away from the input vector:
  \[
  w_i(t) \leftarrow w_i(t-1) - \eta \cdot \mathbf{p}(t) - w_i(t-1) 
  \]
- No change of the weights for the non-winning neurons.

###LVQ1 Algorithm – cont.

- $0 < \eta < 1$ - learning rate, may be constant or monotonically decreasing with time.
- Stopping criteria:
  - Codebook vectors have stabilized or
  - Maximum number of epochs has been reached.

---

**COMP4302/5322 Neural Networks, w7, s2**

2003
Example

• 4 input vectors, 2 classes

• Step 1: assign target vectors to each input
  • Target vectors should be binary – each one contains only zeros except
    for a single

\[
\begin{array}{ccc}
  p_1 = [1, 0, 0, 0] & p_2 = [0, 1, 0, 0] & p_3 = [0, 0, 1, 0] & p_4 = [0, 0, 0, 1]
\end{array}
\]

• Step 2: Choose how many sub-classes will make up each of the 2 classes – e.g. 2 for each class
  • Note: typically the prototype vectors << input vectors

\[
\begin{array}{c}
  W_2 = \begin{bmatrix}
    1 & 1 & 0 & 0 \\
    0 & 0 & 1 & 1
  \end{bmatrix}
\end{array}
\]

=> Neurons 1 and 2 are connected to class 1, neurons 3 and 4 to class 2

Example – Iteration 1, First Layer

• Step 4 – At each iteration of the training, we present an input vector, find its response, and then adjust the weights
  • Iteration 1 – present \( p_1 \) output of the first layer

\[
\begin{array}{c}
  a_1(0) = \text{comp} \begin{bmatrix}
    0.25 \cdot 0.75 \cdot 0.75 \cdot 0.75 \\
    0.75 \cdot 0.25 \cdot 0.25 \cdot 0.25
  \end{bmatrix}
\end{array}
\]

This is the correct class, therefore the weight vector is moved toward the input vector (learning rate=0.5):

\[
\begin{array}{c}
  w_1 = 0.25 \cdot 0.75 \cdot 0.75 \cdot 0.75 \\
  w_2 = 0.75 \cdot 0.25 \cdot 0.25 \cdot 0.25
\end{array}
\]

Example – Iteration 1, Second Layer

\[
\begin{array}{c}
  a_2(0) = W_2 a_1(0) = \begin{bmatrix}
    1 & 1 & 0 & 0 \\
    0 & 0 & 1 & 1
  \end{bmatrix} \begin{bmatrix}
    0.25 \\
    0.75
  \end{bmatrix} = \begin{bmatrix}
    1 \\
    0
  \end{bmatrix}
\end{array}
\]

Example - Final Decision Regions

• Find the 1st and 2d winner – \( i \) and \( j \)
• Adapt their weights simultaneously, if they belong to different classes (i.e. one to the correct class, the other to the incorrect class) and if the input pattern falls into a window near the midplane of these 2 vectors
  • Move the winner (\( i \) or \( j \)) whose class is the same as the input vector towards the input
  • Move the winner (\( i \) or \( j \)) whose class is different than the input vector away from the input

\[
\begin{array}{c}
  d_i \cdot \text{Euclidean distance from } p \text{ to the winner } i \\
  d_j \cdot \text{Euclidean distance from } p \text{ to the winner } j
\end{array}
\]

Definition of window \( w \):

\[
\begin{array}{c}
  w = \min \left( \frac{d_i}{d_j}, \frac{d_j}{d_i} \right) \cdot 1 + \frac{1}{1 + w}
\end{array}
\]

\[
\begin{array}{c}
  d_i \cdot \text{Euclidean distance from } p \text{ to the winner } i
\end{array}
\]

Typically \( w = 0.2 - 0.3 \); if \( w = 0.2 \) \( \Rightarrow w = 0.67 \)

Example – cont. 1

• Step 3 – \( W_1 \) is initialized to small random values
  The weight vectors belonging to the 2 competitive neurons which define class 1 are marked with circles; class 2- squares

\[
\begin{array}{c}
  W(0) = \begin{bmatrix}
    0.25 & 0.75 \\
    0.75 & 0.25 \\
    0.25 & 0.75 \\
    0.75 & 0.25
  \end{bmatrix}
\end{array}
\]

Example - Final Decision Regions

• Find the 1st and 2d winner – \( i \) and \( j \)
• Adapt their weights simultaneously, if they belong to different classes (i.e. one to the correct class, the other to the incorrect class) and if the input pattern falls into a window near the midplane of these 2 vectors
  • Move the winner (\( i \) or \( j \)) whose class is the same as the input vector towards the input
  • Move the winner (\( i \) or \( j \)) whose class is different than the input vector away from the input

\[
\begin{array}{c}
  d_i \cdot \text{Euclidean distance from } p \text{ to the winner } i \\
  d_j \cdot \text{Euclidean distance from } p \text{ to the winner } j
\end{array}
\]

Definition of window \( w \):

\[
\begin{array}{c}
  w = \min \left( \frac{d_i}{d_j}, \frac{d_j}{d_i} \right) \cdot 1 + \frac{1}{1 + w}
\end{array}
\]

\[
\begin{array}{c}
  d_i \cdot \text{Euclidean distance from } p \text{ to the winner } i
\end{array}
\]

Typically \( w = 0.2 - 0.3 \); if \( w = 0.2 \) \( \Rightarrow w = 0.67 \)
**LVQ2.1**
- LVQ2.1 is applied only after LVQ1 has been applied using a small learning rate and small number of iterations.
- It optimizes the relative distance of the codebook vectors from the class borders \(\Rightarrow\) the results are typically more robust.
- Try nnd14lv1 and nnd14lv2!

**LVQ - Issues**
- Initialization of the codebook vectors
  - Random
  - Dead neurons – too far away from the training examples, never take the competition
  - Solution: conscience mechanism
- To the first training examples – typically used
- How many codebook vectors for each class?
  - It is set proportional to the number of input vectors in each class
- How many epochs?
  - Depends on the complexity of the data, learning rate
  - Try nnd14v1!

**Summary and Conclusions**
- LVQ is a supervised learning algorithm
  - Classifies input vectors using competitive layer to find subclasses of input vectors that are then combined into classes.
  - Can classify any set of input vector (linearly separable and non-linearly separable, convex regions and non-convex regions) if:
    - there are enough neurons in the competitive layer
    - each class is assigned enough sub-classes (i.e. enough competitive neurons)