UNSUPERVISED LEARNING

* *Supervised* learning is based on data consisting of example input-output pairs. Learning amounts to adjusting a system to approximate as closely as possible the desired output for a given input.
* *Unsupervised* learning assumes that training is only based on inputs. In such a case, one can basically only try to discover similarities and redundancy in the data. This can be used for *data compression* both by means of *dimensionality reduction* and *clustering.*
* Two types of unsupervised learning exist:
  + *Reinforcement learning:* weights are updated for pairs of neurons that are simultaneously active.
  + *Competitive learning:* weights of those neurons are updated that have the best response on given inputs.

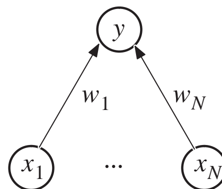
PRINCIPAL COMPONENTS ANALYSIS

Summary:

* Abbreviated by PCA; also called *Karhunen-Loève transformation*.
* Question: given a random vector (a set of sampled vectors), find an orthonormal base that maximizes the variance along the subsequent dimensions of the base.
* The largest variance is achieved when the eigenvector corresponding to the largest eigenvalue of the covariance matrix of the data is chosen.
* When the eigenvalues are sorted in decreasing order, the corresponding eigenvectors give the dimensions that are of decreasing importance.

HEBBIAN LEARNING (1)

* Already encountered in Hopfield associative memories.
* Principle: increase the weight between two neurons if these two neurons fire simultaneously: reinforcement learning.



* Weight update rule:

$$w_j(t+1) = w_j(t) + \eta y x_j$$

* Assuming that all neuron outputs are positive, the weights will continually grow larger (a solution for this problem follows later).

HEBBIAN LEARNING (2)

* Assume that there is a single linear neuron described by:

$$y = \sum_{i=1}^{N} w_i x_i = \mathbf{w}^T \mathbf{x} = \mathbf{x}^T \mathbf{w}$$

* Then the weight update can be written as:

$$\Delta w_j = \eta y x_j = \eta \sum_{i=1}^{N} w_i x_i x_j$$

* Or, in vector notation:

$$\Delta \mathbf{w} = \eta \mathbf{x} y = \eta \mathbf{x}\mathbf{x}^T \mathbf{w}$$

* The patterns in the training set are the vectors $\mathbf{x}^p$, $p = 1, \ldots, Q$.
* The average update due to all patterns is (C is the correlation matrix; the covariance matrix if the $\mathbf{x}^p$ have mean $\mathbf{0}$):

$$\Delta \mathbf{w} = \eta \frac{1}{Q} \sum_{p=1}^{Q} \mathbf{x}^p \mathbf{x}^{pT} \mathbf{w} = \eta C \mathbf{w}$$

* Claim: the weight vector will converge towards the *eigenvector belonging to the largest eigenvalue* of the correlation matrix (*first principal component* of the data)!

LINEAR-SYSTEMS EXPLANATION

* The update rule can be seen as a differential equation:

$$\frac{d}{dt}\boldsymbol{w} = \eta C\boldsymbol{w}$$

* The solution of this equation can be found by first applying a coordinate transformation (T is a matrix of the eigenvectors of C):

$$\boldsymbol{w}^* = T^{-1}\boldsymbol{w}$$

* Then, the solution is:

$$\boldsymbol{w}^*(t) = \begin{bmatrix} w_1^*(0)e^{\lambda_1 t} \\ \vdots \\ w_N^*(0)e^{\lambda_N t} \end{bmatrix}; \boldsymbol{w}(t) = T\boldsymbol{w}^*(t)$$

* Because C is symmetric and positive semidefinite, all eigenvalues are positive. So, the system is unstable. $\boldsymbol{w}(t)$ will asymptotically align with the eigenvector associated with the largest eigenvalue.

GRADIENT-DESCENT EXPLANATION

* Consider the objective function:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{p=1}^{Q}\left[\boldsymbol{x}^{pT}\boldsymbol{w}\right]^2$$

* This function tries to find the vector $\boldsymbol{w}$ which optimally projects to the vectors in the training set, in other words the first principal component.
* The weights that optimize the objective function can be found by the gradient-descent method:

$$\Delta w_j = \eta\frac{\partial J}{\partial w_j} = \eta\left[\sum_{i=1}^{Q}\boldsymbol{x}^{pT}\boldsymbol{w}\right]x_j = \eta y x_j$$

* However, this update rule is exactly the Hebbian learning rule!

WEIGHT-VECTOR NORMALIZATION (1)

* In order to prevent that the weight vector grows indefinitely, one can normalize its size to 1 at each update:

$$w_j(t+1) = \frac{w_j(t) + \eta y x_j}{\sqrt{\sum_{j=1}^{N}\left(w_j(t) + \eta y x_j\right)^2}}$$

* When the learning rate η is sufficiently small, second and higher-order terms can be neglected:

$$\sum_{j=1}^{N}\left(w_j(t) + \eta y x_j\right)^2 \simeq \sum_{j=1}^{N}\left[w_j^2(t) + 2\eta y x_j w_j(t)\right] = 1 + \sum_{j=1}^{N} 2\eta y x_j w_j(t)$$
$$= 1 + 2\eta y^2$$

WEIGHT-VECTOR NORMALIZATION (2)

* Continuing the approximation, one gets:

$$\frac{1}{\sqrt{1 + 2\eta y^2}} \simeq \frac{1}{1 + \eta y^2} \simeq 1 - \eta y^2$$

* So, the update rule becomes:

$$w_j(t+1) = \left(w_j(t) + \eta y x_j\right)\left(1 - \eta y^2\right) \simeq w_j(t) + \eta y\left(x_j - y w_j(t)\right)$$

* This rule looks very much like the original rule with the difference that the input x_j has been replaced by $x_j' = x_j - y w_j(t)$. x_j' is called the *effective input.*

MULTIPLE NEURONS (1)

* Consider a two-layer feedforward network with N inputs and M outputs ($M < N$). All neurons are linear:

$$y_i = \sum_{j=1}^{N} w_{ij}(t)x_j,\ i = 1, \dots, M$$

* The rule for updating weights is different for each neuron:

$$w_{ij}(t+1) = w_{ij}(t) + \eta y_i \left[x_j - \sum_{k=1}^{i} y_k w_{kj}(t) \right],\ i = 1, \dots, M$$

* In vector form:

$$\Delta \boldsymbol{w}_i = \eta y_i \left[\boldsymbol{x} - \sum_{k=1}^{i} y_k \boldsymbol{w}_k(t) \right] = \eta y_i \left(\boldsymbol{x}^* - y_i \boldsymbol{w}_i(t) \right),\ \boldsymbol{x}^* = \boldsymbol{x} - \sum_{k=1}^{i-1} y_k \boldsymbol{w}_k(t)$$

MULTIPLE NEURONS (2)

* So: $\Delta \boldsymbol{w}_i = \eta y_i \left(\boldsymbol{x}^* - y_i \boldsymbol{w}_i(t) \right),\ \boldsymbol{x}^* = \boldsymbol{x} - \sum_{k=1}^{i-1} y_k \boldsymbol{w}_k(t)$
* When $i = 1$, $\boldsymbol{x}^* = \boldsymbol{x}$ which corresponds to the single-neuron case investigated before. The weights of the first neuron will converge to the first principal component.
* Consider $i = 2$ and assume that the weights of the first neuron have converged to their final value. Then, $\boldsymbol{x}^* = \boldsymbol{x} - y_1 \boldsymbol{w}_1(t)$, which means that the neuron sees an input from which the first principal component has been removed (y_1 is exactly the projection of $\boldsymbol{x}$ on $\boldsymbol{w}_1(t)$). Therefore, the weights $\boldsymbol{w}_2(t)$ will converge to the second principal component from the training set.

MULTIPLE NEURONS (3)

* Following a similar reasoning, it is not difficult to conclude that the network weights will converge to the M largest principal components of the network.
* In reality the weights will not converge one by one but simultaneously.
* This method of computing principle components can be far more efficient than the computation via eigenvectors for large N.

SIMPLE COMPETITIVE LEARNING (1)

* Consider again a two-layer feedforward network with N inputs and M outputs.
* The neurons are unconventional: instead of computing a weighted sum followed by the application of an activation function, they compute the distance between the weight vector and the input vector (e.g. the Euclidean distance):

$$y_i = \|\boldsymbol{w}_i - \boldsymbol{x}\|,\ i = 1, \dots, M$$

* The neuron that has the minimal output (whose weight vector is closest to the input) is called the *winning neuron.*
* Only the weight vector of the winning neuron is updated according to:

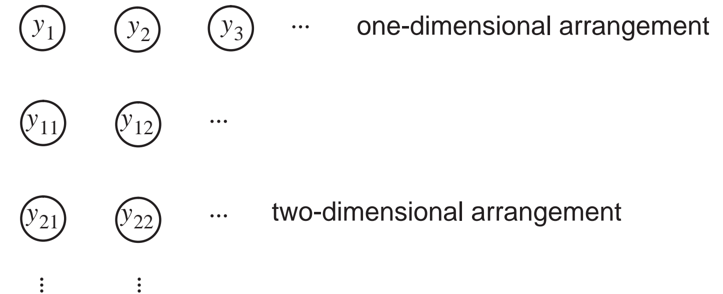
$$\boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t) + \eta(\boldsymbol{x} - \boldsymbol{w}_i)$$

SIMPLE COMPETITIVE LEARNING (2)

* The effect of this type of learning is that the weight vectors move towards clusters of input vectors.
* After convergence some kind of *clustering* has been achieved (compare this approach to the *k-means* clustering algorithm). The *prototypes* are given by the weight vectors.
* The algorithm has the disadvantage that an optimal clustering cannot always be found: weights of neurons that never become winning neurons are never modified. These are the so-called *dead neurons.*

KOHONEN LEARNING (1)

* *Self-organizing map:* a two-layered network (with an input and an output layer) in which the neurons have *positions* with respect to each other. Examples:



KOHONEN LEARNING (2)

* Linear and two-dimensional arrangements of neurons are often used. However, higher-dimensional arrangements or embeddings in non-Euclidean spaces are also possible.
* Because neurons have positions, a distance $d(i, j)$ $(i, j = 1, \dots, M)$ between pairs of neurons can be defined.
* The computation performed is similar to simple competitive learning. Determine first the winning neuron k:

$$k = \arg \max_{i=1}^{M} \|\mathbf{w}_i - \mathbf{w}\|.$$

* The rule for updating weights is:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta h(k, i)(\mathbf{x} - \mathbf{w}_i)$$

* $h(k, i)$ is called a *neighborhood function*. It should decrease with increasing distance $d(k, i)$.

KOHONEN LEARNING (3)

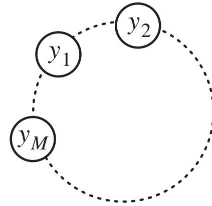
* Using a neighborhood function eliminates dead neurons.
* A frequently used function for the neighborhood function is a Gaussian:

$$h(i, j) = \exp\left[\frac{-d(i, j)}{2\sigma^2}\right]$$

* For reasons of sufficient exploration of the solution space on one hand and convergence on the other, it is a good idea to make σ time dependent with large values at the beginning and small ones at the end.
* One of the main applications is clustering and vector quantization. During the training, the prototype points are determined. Inputs offered to the network later on are mapped to one of the classes found during training, viz. the class associated with the winning neuron.

APPLICATION TO TSP

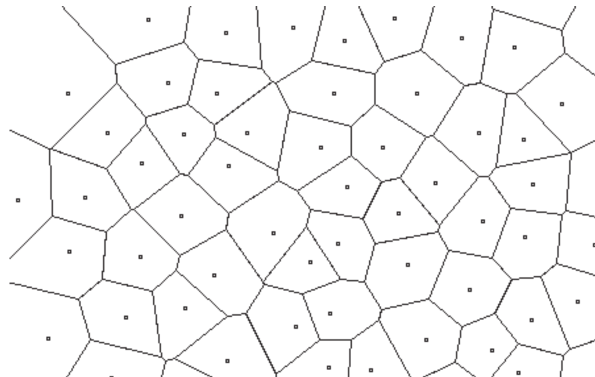
* Given is a set of cities with coordinates $\mathbf{x}^p = (x_1^p, x_2^p),\ p = 1, \ldots, Q$.
* The goal is to visit all cities once and return to the original one using the shortest possible tour.
* Use a Kohonen self-organizing map that is circular (i.e. a linear arrangement in which the last neuron is a direct neighbor of the first).
* There are as many neurons as there are cities. The algorithm may add new neurons to break ties.
* Eventually, the weights will be mapped on city coordinates.
* There are, however, better ways to solve TSP.



TERMINOLOGY

* Clearly distinguish:
  + *Clustering* (sometimes also called *vector quantization*): the process of trying to find *prototypes* in a set of unlabeled data.
  + *Classification:* try to determine to which of the predetermined subsets a data item belongs.
* Clustering is normally applied to a “fresh” set of data.
* A classifier is normally built by finding appropriate parameter values derived from a *training set*. Once the parameter values have been fixed, it is used for classifying new data.

VORONOI DIAGRAMS (REMINDER)



* Subdivide the plane in *Voronoi* regions by perpendicular bisectors between neighboring pairs of points.

LEARNING VECTOR QUANTIZATION (1)

* *Learning vector quantization* (LVQ) can be applied to build classifiers. It uses a labeled training set: the class to which a data item belongs is known in advance. The main idea is to use clustering techniques first and then use the labels to modify cluster boundaries.
* LVQ is a *supervised* learning method.

LEARNING VECTOR QUANTIZATION (2)

The algorithm:

* Apply clustering ignoring the class labels, e.g. by using the Kohonen method or k-means clustering. The result is a set of prototypes or centroids $\boldsymbol{w}_i$, $i = 1, \ldots, M$. The centroids define Voronoi regions $\mathcal{V}^i$.
* Assign a class to each centroid using *voting:* the class to which the majority of points in $\mathcal{V}^i$ belongs determines the class of $\boldsymbol{w}_i$.
* Repetitively present inputs $\boldsymbol{x}^p$, $p = 1, \ldots, Q$ from the training set. Denote the class of $\boldsymbol{x}^p$ by $\mathcal{C}(\boldsymbol{x}^p)$ and the class of $\boldsymbol{w}_i$ by $\mathcal{C}(\boldsymbol{w}_i)$.
  + Consider $\boldsymbol{w}_i$ for which $\boldsymbol{x}^p \in \mathcal{V}^i$. If $\mathcal{C}(\boldsymbol{x}^p) = \mathcal{C}(\boldsymbol{w}_i)$:
    $\boldsymbol{w}_i(t + 1) = \boldsymbol{w}_i(t) + \eta(\boldsymbol{x}^p - \boldsymbol{w}_i)$ (move centroid closer to $\boldsymbol{x}^p$).
  + Otherwise:
    $\boldsymbol{w}_i(t + 1) = \boldsymbol{w}_i(t) - \eta(\boldsymbol{x}^p - \boldsymbol{w}_i)$ (move centroid away from $\boldsymbol{x}^p$).

FURTHER READING

* A thorough proof of the convergence of the Hebbian learning rule for a single neuron towards the first principle component is given in [1].
* An explanation of the multiple-output PCA neural network can also be found in [1].
* Many of the issues of this lecture are also covered in [2].

[1] Haykin, S., Neural Networks, A Comprehensive Foundation, Prentice Hall International, Upper Saddle River, New Jersey, Second Edition, (1999).

[2] Jang, J.S.R., C.T. Sun and E. Mizutani, Neuro–Fuzzy and Soft Computing. A Computational Approach to Learning and Machine Intelligence, Prentice Hall, Upper Saddle River, NJ, (1997).