Neural Networks: Basic

Dokuz Eylül University
Computer Engineering Department

(25 September 2000 – 5 January 2001)

□ Contents:

1 What is Neural Network?

☐ A system with *neurons* connected with *synapses*,

where

- synapse:

  transmits input state value (usually either one or zero) with its weight value being multiplied to the input; and

- neuron:

  takes the weighted sum of fan-in inputs, and outputs its function value of the specified *transfer function* $f(x)$.

☐ Typically categorized into two forms:

1. Layered (feed forward)
2. Fully-connected

Keywords:

- **neuron, synapse, weight, transfer-function, layered, feed-forward, fully-connected**

1.1

Perceptron

□ An example of single-layered feed forward neural network.

Suppose:

- M inputs: $x_1, \cdots x_M$;
- N outputs: $y_1, \cdots y_N$;
- each N output neurons are connected with all M input neurons;
- weight values (of the synapse) from (input) neuron j to (output) neuron k is w_{jk};

then

$$y_i = f(\sum_{k=1}^{N} w_{ki} x_k), \tag{1}$$

where

$$f(x) = \mathrm{sgn}(x) \quad \text{or} \quad f(x) = \tan^{-1}(x). \tag{2}$$

Keywords:

- **perceptron**

□ Analogy from nature:

- from sensors at retina to sensori-motor-output in brain.

□ What can we do with this?

- prediction of sun-spots, stock market, etc
- character recognition
- $\cdots$

□ How can we specify weight values?

- Learning by themselves (a sort of Machine Learning).
  - Supervised Learning (= Learning with Teacher)
  - Unsupervised Learning (= Learning without Teacher)

Keywords:

- **learning, supervised, unsupervised**

1.1.1

The simplest example.

1. AND with two inputs and one output

   e.g.,

   * $w_{00} = w_{01} = 0.7$

2. How about OR?

   e.g.,

   * $w_{00} = w_{01} = 1.4$

3. How about XOR?

   – Impossible by single layer perceptron.

Keywords:

- **synaptic plasticity**

1.1.2

Perceptron with hidden layers

Adding intermediate layer(s) called *hidden layer(s)* between input and output.

- Suppose K hidden layers.
- weight from neuron i of the k-th layer to neuron j of the next layer is denoted as w^k_{ij}.

- XOR revisited

  e.g.,

  * $w^0_{00} = w^0_{01} = 0.7, \quad w^0_{10} = w^0_{11} = 1.4,$

    $w^1_{10} = -1.4, \ w^1_{11} = 1.4.$

- Separability

  - If we can separate the solution space by hyperplane,

    (To be more specific, we can separate 2D solution space of 2-input AND/OR by a straight line, but not for XOR.)

    then we can solve the problem by single perceptron.

  - Multi-layer allows us to separate the space by more flexible surface, not necessarily the plane.

Keywords:

- **hidden layer, separability, solution-space**

1.2

Examples of Learning

- Perceptron Learning
- Gradient Learning
- Error Back Propagation
- Evolutionary computation (such as GA)
- etc

1.2.1

Perceptron Learning

- Assume M input and N output neurons.

- Also assume that we have p examples

$$\sigma_i^\mu, \quad \mu = 1, 2, \cdots, p, \quad i = 1, 2, \cdots, M, \tag{3}$$

whose desired output are known, i.e.,

$$\zeta_j^\mu, \ i = 1, 2, \cdots, N. \tag{4}$$

while real output was

$$S_i^\mu, \quad i = 1, 2, \cdots, N. \tag{5}$$

- Then we modify w_{ij} by adding:

$$\delta w_{ik} = \epsilon(\zeta_i^\mu - S_i^\mu)\sigma_k^\mu, \tag{6}$$

in each iteration of giving example, and we obtain

$$w_{ik}^{\text{new}} = w_{ik}^{\text{old}} + \delta w_{ik}, \tag{7}$$

(implying *if correct do nothing, otherwise modify.)*

- Note that each of $\sigma_k^\mu, \ S_i^\mu, \ and \ \zeta_k^\mu$ takes either 1 or -1.

□ Try to calculate the below by hand.

- Three binary inputs and two binary outputs.

- If the number of $+1$ is even then output should be $-1+1$
  If the number of $+1$ is odd then output should be $+1-1$
  (that is to say, y_1 is *odd-flag* and y_2 is *even-flag*.)

- Pick up 6 examples out of 8 possible input to give them for adjusting w_{ij}'s.

- Apply the Perceptron Learning Algorithm with

  $$\epsilon = 0.05$$
  $$f(x) = \mathrm{sgn}(x-1)$$

- Repeat giving the examples until w_{ij} won't change any more. (You may use one example multiple time if needed.)

- Test the results with 2 remaining possible inputs.

□ Try to make program for the one with 10 inputs.

□Analog vs Binary Neurons

- Binary:
  - Typical Transfer function:
    $$y_i = \mathrm{sgn}(\sum_k w_{ik} x_k - \theta)$$
  - What does this imply?
  - θ is called *threshold.*

- Analog:
  - $y_i = f(\sum_k w_{ik} x_k - \theta)$
  - $f(h) = \tanh(\beta h)$

From the definitions:

$$e^{jx} = \cos x + j \sin x; \tag{8}$$

$$e^{-jx} = \cos x - j \sin x, \tag{9}$$

we obtain

$$\cos x = \frac{e^{jx} + e^{-jx}}{2}; \tag{10}$$

$$\sin x = \frac{e^{jx} - e^{-jx}}{2}; \tag{11}$$

We extend them to

$$\cosh x = \frac{e^{x} + e^{-x}}{2}; \tag{12}$$

$$\sinh x = \frac{e^{x} - e^{-x}}{2}; \tag{13}$$

So,

$$\tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{14}$$

□ **Gradient Learning** (Learning for analog neurons.)

Squared Error of outputs:

$$D = \sum_\mu \sum_i (S_i^\mu - \zeta_i^\mu), \tag{15}$$

where

$$S_i^\mu = f(\sum_k w_{ik}\sigma_k^\mu); \tag{16}$$

$$f(h) = \tanh(\beta h) \tag{17}$$

defines an *error surface* of high dimensionality.

Then, the *gradient* of D w.r.t. w_{ij} modifies w_{ij} as:

$$\delta w_{ik} \equiv -\epsilon \frac{\partial D}{\partial w_{ik}} = \cdots = \epsilon \sum_\mu (\zeta_i^\mu - S_i^\mu) f'(\sum_k w_{ik}\sigma_k^\mu), \tag{18}$$

where

$$f'(x) = \beta\{1 - f(x)^2\}. \tag{19}$$

Though there exist many alternatives to find minima of a non-linear function $g(\mathbf{x})$, *steepest descent*

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \epsilon \nabla g(\mathbf{x}_n) \tag{20}$$

is sufficient in most cases.

Back Propagation Algorithm

□ Definition of variables:

- L layers of neurons.
- k_r neurons in the r-th layer. $\quad r = 1, 2, \cdots, L.$
- l input neurons ($k_0 = l$).
- w_{jk}^r: weight to the j-th neuron in the r-th layer from the k-th neuron in the previous layer.
- N training pairs: $(\mathbf{x}(i), \mathbf{y}(i)), \quad i = 1, 2, \cdots, N.$
- $y_k^r(i)$: the output of the k-th neuron in the r-th layer when the i-th training sample is given.
- fan-in's to j-th neuron in the r-th layer when the i-th training sample is given.

  $$h_j^r(i) \equiv \sum_{k=1}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i) + w_{j0}^r = \sum_{k=0}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i),$$

  where $y_0^{r-1} = 1$ and

  $$(y_1^0(i), \cdots, y_l^0(i)) = (x_1(i), \cdots, x_l(i)) = \mathbf{x}(i).$$

- Each output neuron's error:

  $$e_m(i) \equiv y_m(i) - \hat{y}_m(i), \quad i = 1, 2, \cdots, N.$$

  where $(\hat{y}_1(i), \cdots, \hat{y}_{k_L}(i)) = \mathbf{y}(i)$

- Error function to be minimized:

  $$D = \frac{1}{2} \sum_{i=1}^{N} \sum_{m=1}^{k_L} e_m(i)^2$$

□ The algorithm:

1. *Initialization:* Initialize all the weights with small random values.
2. *Forward computations:* For each of the training vectors

   $$\mathbf{x}(i), \quad i = 1, 2, \cdots, N$$

   compute all the

   $$y_j^r(i) = f(h_j^r(i))$$

   for $j = 1, 2, \cdots, k_r$ and $r = 1, 2, \cdots, L$.
3. Backward computations: For each $i = 1, 2, \cdots, N$, and $j = 1, 2, \cdots, k_L$ compute

   $$\delta_j^L(i) = e_j(i) f'(h_j^L(i))$$

   then compute

   $$\delta_j^{r-1}(i) = e_j^{r-1}(i) f'(h_j^{r-1}(i))$$

   for $r = L, L-1, \cdots, 2$ and $j = 1, 2, \cdots, k_r$ where

   $$e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$$
4. Update the weights for $r = 1, 2, \cdots, L$ and $j = 1, 2, \cdots, k_r$

   $$w_{jk}^r\,[\text{new}] = w_{jk}^r\,[\text{old}] + \triangle w_{jk}^r,$$

   where

   $$\triangle w_{jk}^r = -\epsilon \sum_{i=1}^{N} \delta_j^r(i) y_k^{r-1}(i).$$

□ Rationale:

From the *gradient descent*'s principle,

$$\triangle w_{jk}^{r} = -\epsilon \frac{\partial D}{\partial w_{jk}^{r}} \tag{21}$$

where

$$D = \sum_{i=1}^{N} \varepsilon(i) \tag{22}$$

and

$$\begin{aligned} \varepsilon(i) &= \frac{1}{2} \sum_{i=1}^{N} \sum_{m=1}^{k_L} e_m(i)^2 && (23) \\ &= \frac{1}{2} \sum_{i=1}^{N} \sum_{m=1}^{k_L} (y_m(i) - \hat{y}_m(i))^2. && (24) \end{aligned}$$

Hence

$$\frac{\partial \varepsilon(i)}{\partial w_{jk}^{r}} = \frac{\partial \varepsilon(i)}{\partial h_j^r} \frac{\partial h_j^r}{\partial w_{jk}^{r}}. \tag{25}$$

Here we define

$$\frac{\partial \varepsilon(i)}{\partial h_j^r} \equiv \delta_j^r(i). \tag{26}$$

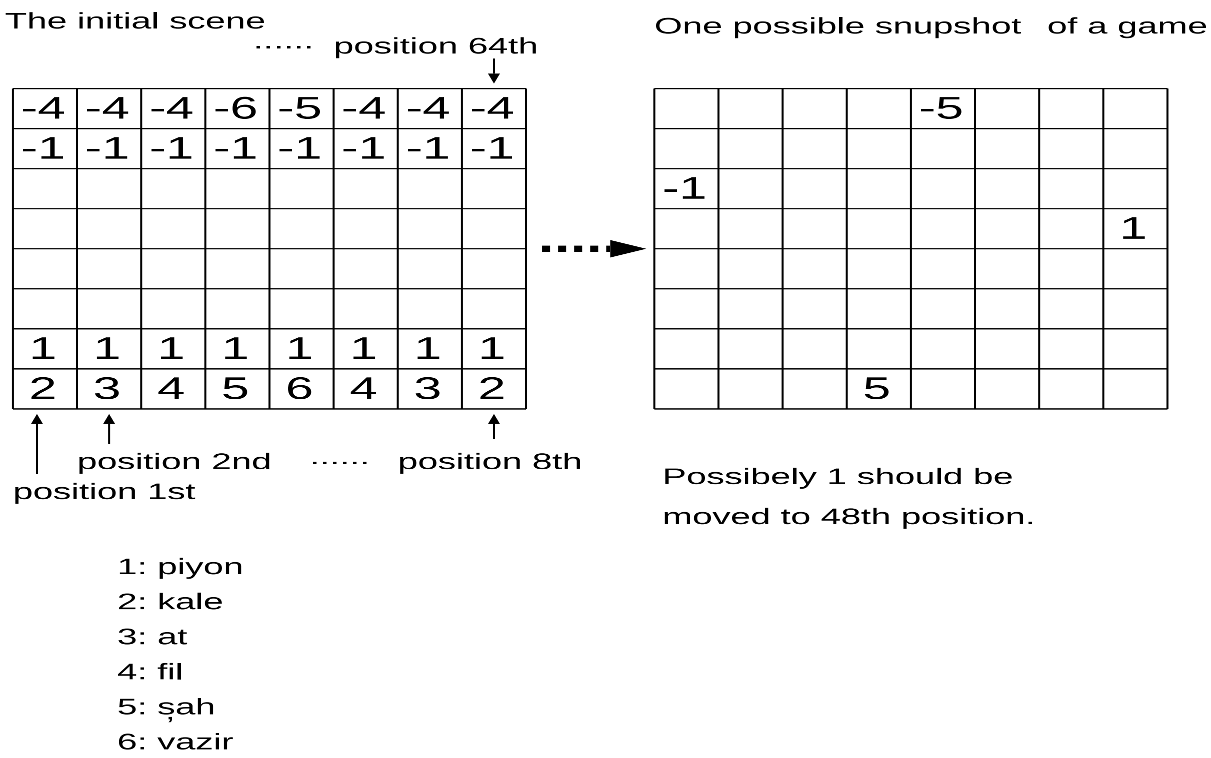
Then

$$\triangle w_{jk}^{r} = -\epsilon \sum_{i=1}^{r} (i) \delta_j^r(i) y_k^{r-1}(i). \tag{27}$$

Applications of Neural Networks

- Learning to steer a car simulator.
  - ⋆ D. A. Pomerleau (1991) *"Rapidly Adapting Artificial Neural Networks for Autonomous Navigation."* R. P. Lipman et al. (Eds.) Advances in Neural Information Processing Systems 3, Morgan Kaufmann, p429.
- Learning to pronounce English Text (NET-Talk)
  - ⋆ T. J. Sejnowski, and C. R. Rosenberg (1987) *"Parallel Networks that learn to Pronounce English Texts."* Complex Systems 1, p145.
- Stock Price Prediction
  - ⋆ E. Schöneburg (1990) *"Stock Price Prediction using Neural Networks: A Project Report."* Neurocomputing 2, p17.
- Learning to Play Backgammon
  - ⋆ G. Tesauro, T. J. Sejnowski (1988) *"A Neural Netwokr that Learns to play Backgammon."* D. Z. Anderson(Ed.) Neural Information Processing Systems. American Institute of Physics.

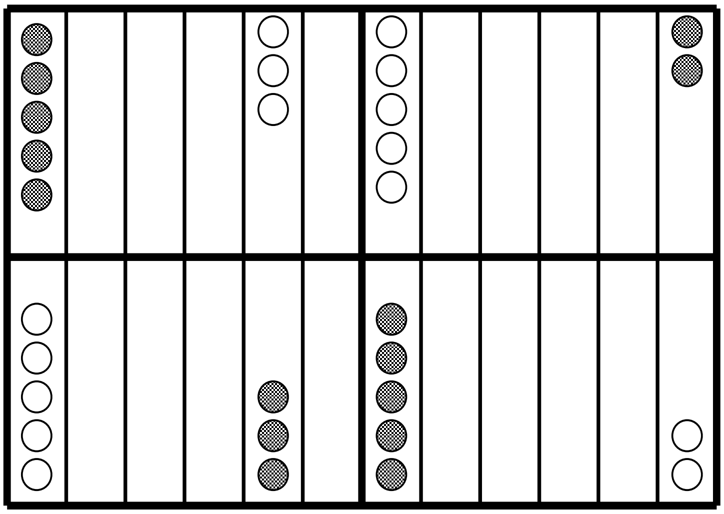
□ Chess (Satranç) by Neural Networks



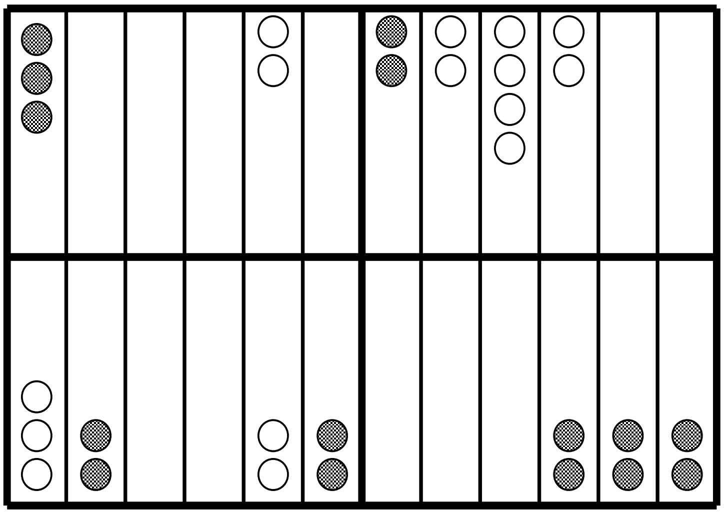
- One possible implementation:
  - 64 input neurons each of which takes a value from –6 to 6 representing a stone, plus one input binary neuron representing whose turn, which corresponds to one scene of a game.
  - 2 output neurons: one takes a value from –6 to 6 and the other from 1 to 64, representing a possible best movement, that is, which stone should be moved to where? (e.g., output 1 and 48 means that the stone-1 should be moved to

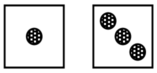
□ Backgammon (Tavla) by Neural Networks

The initial scene.



A possible one scene in a game.



When  what is the possible best movements?

- One possible implementation:
  - a total of 29 input neurons;
    - ⋆ 26 input neurons[1] each of which takes a value from -15 to 15, representing a number of stones at each cite, positive ones are mine and negative ones are opponent's, plus two neurons with each indicating the figure of two dice, and one more neuron representing whose turn.

- 4 output neurons[2]
  - ⋆ each of which takes a value from 1 to 6 which indicates how many positions the corresponding stone proceeds.

or

- 24 output neurons
  - ⋆ indicating the next scene as the 24 input neurons for the purpose above.

[1] two of which are for the outside position of mine and opponent's, respectively
[2] Usually we need two but if the figures of two dice are identical, the number of movement is four.

□ Reward and Penalty Learning:

An example on perceptron with one hidden layer.

- N_0 *analog* and *deterministic* output neurons:

$$y_i^2 = f(\sum_j w_{ij}^2 y_j^1) \tag{28}$$

  where

$$f(x) = \frac{1}{1+\exp(-x)} \tag{29}$$

- *Binary* and *stochastic* hidden neurons:

$$\text{prob}(y_i^1 = 1) \; = \; f(\sum_j w_{ij}^1 x_j) \tag{30}$$

$$\text{prob}(y_i^1 = 0) \; = \; 1 - f(\sum_j w_{ij}^1 x_j) \tag{31}$$

- Using

$$D = \frac{1}{N_0} \sum_i (\hat{y}_i^2 - y_i^2)^2 \tag{32}$$

  – modify weights to output layer by *gradient* method as

$$\delta w_{ij}^{\text{out}} = \epsilon(\hat{z}_i - z_i) f'(\sum_k w_{ik}^{\text{out}} y_k) y_j \tag{33}$$

  – and weights to hidden layer in a stochastic way as

$$\begin{aligned} \delta w_{ij}^{\text{hidden}} \; &= \; r\epsilon(y_i - f(\sum_k w_{ik}^{\text{hidden}} x_k) x_j \\ &+ \; (1-r)\lambda\epsilon(1 - y_i - f(\sum_k w_{ik}^{\text{hidden}} x_k) x_j \end{aligned} \tag{34}$$

where r is signal meaning success/failure of the task, i.e.,

- $\star$ $r = 1$ with probability $1 - D$;
- $\star$ $r = 0$ with probability D.

IF $rand() < 1 - D$;
THEN $r = 1$;
ELSE $r = 0$;

- Note: Parameter notations of the previous few pages:

The i-th output neuron outputs z_i
while desirerd output is $\hat{z}_i$.

$$\Uparrow$$

The i-th output neuron takes

$$\sum_{k}^{N_{\text{hidden}}} w_{ik}^{\text{out}} y_k$$

from all the hidden neurons.

$$\Uparrow$$

Each of hidden neuron outputs y_k,

$$\Uparrow$$

The i-th hidden neuron takes

$$\sum_{k}^{N_{\text{input}}} w_{ik}^{\text{hidden}} x_k$$

from the input layer.

$$\Uparrow$$

Each input neuron takes an input x_k, and
send to all neurons in the hidden layer.

- Apply this to a task of symmetry detection.
  - n^2 input, 12 hidden neurons, and 3 output.
  - if the $n \times n$ pixels of the input are
    * horizontally symmetric
    * vertically symmetric
    * diagonally symmetric

    the 1st, 2nd and 3rd neuron should be acctivated.

- Interpretation of the reward and penalty learning.

□ Unsupervised Learning:

Unsupervised learning can only do anything useful when there is redundancy in the input data. *In this sence redundancy provides knowledge.*

- Learning by Competition.
- Kohonen's Self-organizing Map.
- Reinforcement Learning.

(1) Unsupervised Learning through Competition:

Let $\mathbf{x} = (x_1, ..., x_n)$ be a normalized input vector, i.e.,

$$\sum_{i=1}^{n} x_i = 1, \tag{35}$$

in n-dimensional space which we want to classify in one of k different clasters.

- The network consitsts of n real-valued input neurons and k binary output neurons (with all neurons zero-threshold.)
- Each output neuron receives
  $$h_i = \sum_{j=1}^{n} w_{ij} x_j \tag{36}$$
  from all the neurons in the input layer.

- Then the only neuron which recieves
  $$h_m = \max_i (h_i) \tag{37}$$
  will be on.

- Learning Algorithm:

1. Initialize all the weights w_{ij} at random, and normaliz them such that

$$\sum_{j=1}^{n} w_{ij} = 1 \quad \text{for} \quad \forall i = 1, ..., k \tag{38}$$

2. Selct a vector $\mathbf{x}^{\mu}$ at random from the training sample, and compute

$$h_i = \sum_{j=1}^{n} w_{ij} x_j \quad \text{for} \quad \forall i = 1, ..., k \tag{39}$$

3. **The Winner Takes All Procedure**

   * Modify only the weight w_{mj} which recieves

$$h_m = \max_i h_i \tag{40}$$

   as

$$w_{mj}^{\text{new}} = w_{mj}^{\text{old}} + \delta w_{mj} \tag{41}$$

   where

$$\delta w_{mj} = \epsilon (x_k - w_{ik}) \tag{42}$$

   and normalize them again.

4. Repeat 2 and 3 until a predetermined number of steps is reached.

Note that

· The difference between this algorithm and perceptron learning is that the input set cannot be classified a priori in a positive or a negative set or in any of several different clusters.

· When we monitor the number of updates for each weight vector in order to make the process as balanced as possible, we call this learning with conscience.

- A numerical excersise:

  Assume three input neurons and three output neurons with w_{ij}

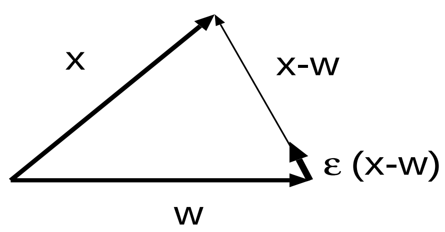
$$\begin{pmatrix} 0.23 & -0.72 & -0.51 \\ -0.49 & 0.33 & -0.16 \\ 0.23 & 0.68 & 0.84 \end{pmatrix}. \tag{43}$$

  Then under an input $(-1, 1, -1)$,
  - $*$ (1) obtain h_m;
  - $*$ (2) modify weights.

- How it works?
  - When we take $\mathbf{x}$ and $\mathbf{w}$ as

$$\begin{aligned} \mathbf{x} &= (x_1, x_2, \cdots, x_{N_i},) \\ \mathbf{w} &= (w_{i1}, w_{i2}, \cdots, w_{iN_i},) \end{aligned}$$

    the equation (42) implies a vector such as: Therefore



    $\mathbf{w} + \delta\mathbf{w}$ makes $\mathbf{w}$ become closer to $\mathbf{x}$, and

  - the more $\mathbf{w}$ is closer to $\mathbf{x}$, the larger the inner product h_i becomes.

  - What will happen when multiple $\mathbf{x}^\mu$ is repeatedly given?

(2) Kohonen's Self-organizing Network:

Simulation of topology preserving maps in the brain.

- Kohonen Learning Algorithm

  The n-dimensional weight vectors $w_1, .., w_m$ of the m computing neurons are selected at random. An initial radius r, a learning constant η, and a neighborhood funciton ϕ are selected.

1. Select an input vector ζ using the desired probability distribution over the input space.

2. The neuron k with the maximum excitation is selected, that is, for which the distabnce between w_i and ζ is minimal for $i = 1, \cdots, m$.

3. The weight vectors are updated using the nighborhood function and the update rule

$$w_i = w_i + \eta\phi(i, k)(\zeta - w_i)\mathrm{for} i = 1, \cdots, m \tag{44}$$

4. Stop if the maximum number of iterations has been reached, otherwise modify η and ϕ as sceduled and continue with step 1.

- Examples:
  - (i) from 2D to 1D
    - ∗ points inside of a triangle in 2-dimensional space into 11 classes two inputs for x1,x2 representing coordinate eleven outputs corresponding to (wi1,wi2) which is closest to the (x1,x2) will be on
    - ∗ plot (w,w)
  - (ii) from 2D to 2D

(3) Unsupervised reinforcement learning

- Assume that a set of empirical data is given which consists of the n-dimensional vectors $(x_1, \cdots, x_m)$.
- *The first principal component* of this set of vectors is a vector $\mathbf{w}$ which maximizes

$$\frac{1}{m}\sum_{i=1}^{m} \|w_i x_i\|^2 \tag{45}$$

- The second principal component is computed by subtracting from each vector x_i its projection on the first principal component.

- architecture
  - n input neurons which are given $x_1, \cdots, x_n$
  - one ouput neuron which receives

$$\sum_{i=1}^{n} w_i x_i \tag{46}$$

- Algorithm — Computation of the first principal component
  - Let $\mathbf{x}$ be a n-dimensional vector.
  - The vector $\mathbf{w}$ is initialized randomly ($w_i \neq 0$).
  - A learning constant γ with $0 < \gamma \leq 1$ is selected.
  - A vector x is selected randomly from x
  - The scalar product $\phi = xw$ is computed the new weight vector is $w + \gamma\phi(x - \phi w)$
  - go to update making γ smaller.