Neural Network

MadaLine (Multiple Adaptive Linear Neuron)

Back Propagation

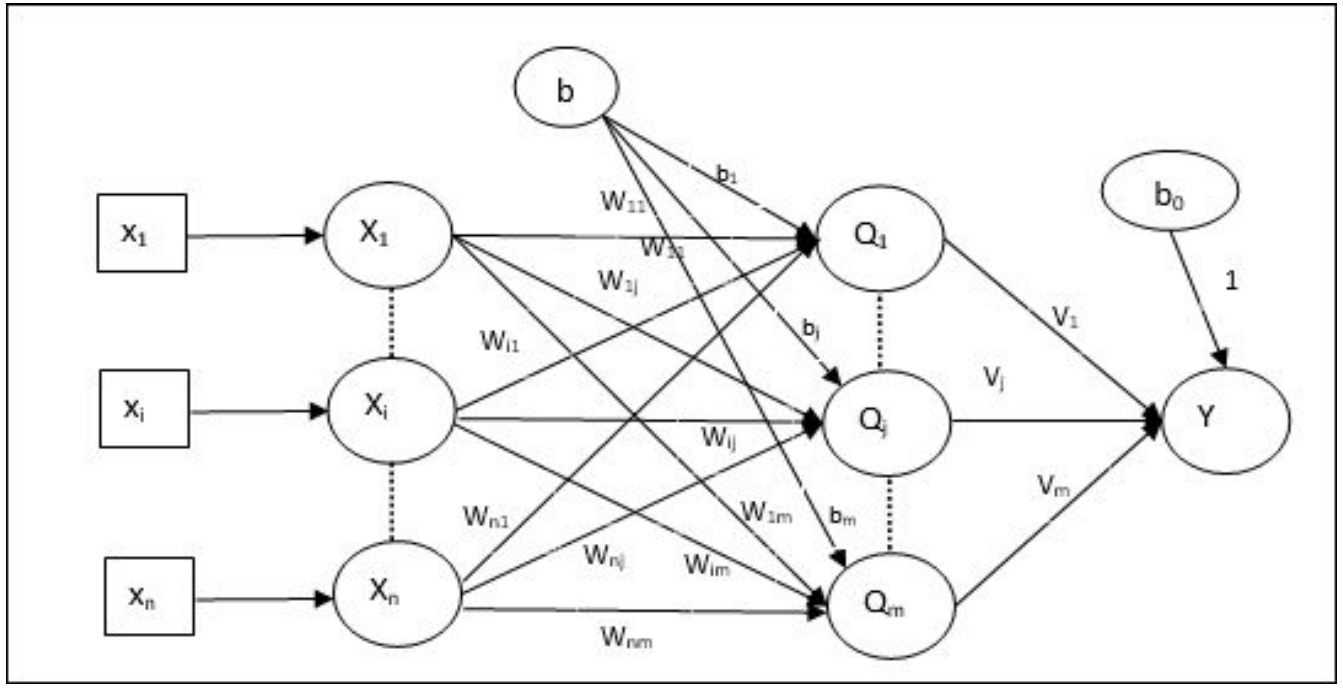
Khaleda Akther Papry
Assistant Professor,CSE,DUET
Email: papry.khaleda@duet.ac.bd
Room: 7023 (New academic building)

Madaline

- Madaline - Multiple Adaptive Linear Neuron, is a network which consists of many Adalines in parallel.
- It will have a single output unit.
- It is just like a multilayer perceptron, where Adaline will act as a hidden unit between the input and the Madaline layer.
- The weights and the bias between the input and Adaline layers, as in we see in the Adaline architecture, are adjustable.
- The Adaline and Madaline layers have fixed weights and bias of 1.
- Training can be done with the help of Delta rule.

Madaline

- The architecture of Madaline consists of
  - “n” neurons of the input layer,
  - “m” neurons of the hidden layer and
  - 1 neuron in the output layer, i.e. the Madaline layer.



Training Algorithm of Madaline

- Step 1 − Initialize the following to start the training −
  - Weights
  - Bias
  - Learning rate α

* For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

- Step 2 − Continue step 3-8 when the stopping condition is not true.
- Step 3 − Continue step 4-7 for every bipolar training vector x.
- Step 4 − Activate each input unit: x_i *(i=1ton)*
- Step 5 − Obtain the net input at each hidden layer with the following relation −

$$Q_{inj} = b_j + \sum_{i}^{n} x_i w_{ij} \quad j = 1\,to\,m$$

* Here 'b' is bias and 'n' is the total number of input neurons.

Training Algorithm of Madaline

- Step 6 – Apply the following activation function to obtain the final output.

$$f(y_{in}) = \begin{cases} 1 & if\, y_{in} \geqslant 0 \\ -1 & if\, y_{in} < 0 \end{cases}$$

Output at the hidden (Adaline) unit : $Q_j = f(Q_{inj})$

Final output of the network: $y = f(y_{in})$

i.e. $y_{inj} = b_0 + \sum_{j=1}^{m} Q_j v_j$

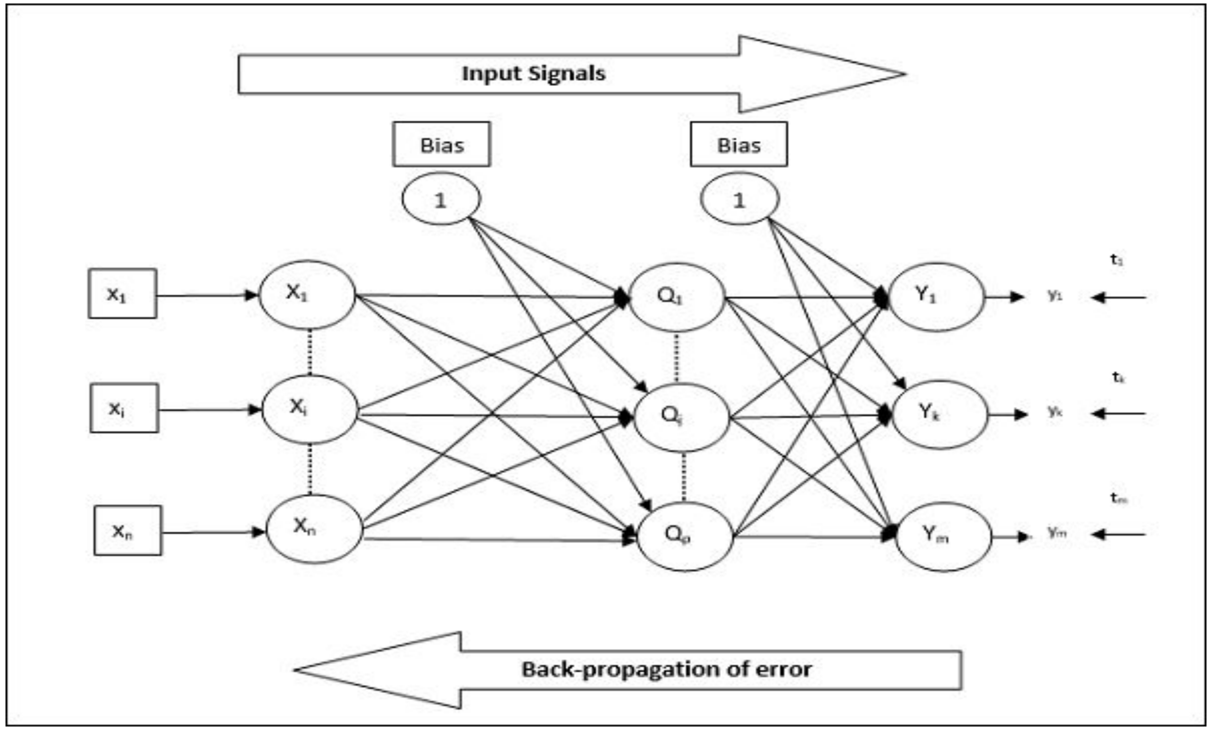
- Step 7 – Calculate the error and adjust the weights as follows –
  - **Case 1** – if y ≠ t and t = 1 then, $\boldsymbol{w_{ij}(new) = w_{ij}(old) + \alpha\,(t - Q_{inj})\, x_i}$
    $\boldsymbol{b_j(new) = b_j(old) + \alpha\,(t - Q_{inj})}$
  - **Case 2 –** if **y = t** then, There would be no change in weights
- Step 8 – Test for the stopping condition, which would happen when there is no change in weight.

Back Propagation

- Back Propagation Neural (BPN) is a multilayer neural network consisting of the input layer, at least one hidden layer and output layer.
- The error which is calculated at the output layer, by comparing the target output and the actual output, will be propagated back towards the input layer.
- The hidden layer as well as the output layer also has bias, whose weight is always 1, on them.

Back Propagation (Architecture)

- The working of BPN is in two phases:
    - One phase sends the signal from the input layer to the output layer
    - The other phase back propagates the error from the output layer to the input layer.



Back Propagation (Training Algorithm)

- For training, BPN will use binary sigmoid activation function. The training of BPN will have the following three phases.
  - **Phase 1** – Feed Forward Phase
  - **Phase 2** – Back Propagation of error
  - **Phase 3** – Updating of weights
- All these steps will be concluded in the algorithm as follows:
- Step 1 – Initialize the following to start the training –
  - Weights
  - Learning rate α
- For easy calculation and simplicity, take some small random values.
- Step 2 – Continue step 3-11 when the stopping condition is not true.
- Step 3 – Continue step 4-10 for every training pair.

Back Propagation (Training Algorithm)

Phase 1

- **Step 4** − Each input unit receives input signal **x_i** and sends it to the hidden unit for all **i = 1 to n**
- **Step 5** − Calculate the net input at the hidden unit using the following relation −

$$Q_{inj} = b_{0j} + \sum_{i=1}^{n} x_i v_{ij} \quad j = 1\, to\, p$$

Here **b_{0j}** is the bias on hidden unit, **v_{ij}** is the weight on **j** unit of the hidden layer coming from **i** unit of the input layer.

Now calculate the net output by applying the following activation function- $Q_j = f(Q_{inj})$

Send these output signals of the hidden layer units to the output layer units.

Back Propagation (Training Algorithm)

- **Step 6** – Calculate the net input at the output layer unit using the following relation –

$$y_{ink} = b_{0k} + \sum_{j=1}^{p} Q_j \, w_{jk} \quad k = 1 \, to \, m$$

Here **b_{0k}** is the bias on output unit, **w_{jk}** is the weight on **k** unit of the output layer coming from **j** unit of the hidden layer.

Calculate the net output by applying the following activation function- $y_k = f(y_{ink})$

- **Phase 2**

- Step 7 – Compute the error correcting term, in correspondence with the target pattern received at each output unit as follows –

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

- On this basis, update the weight and bias as follows – $\Delta v_{jk} = \alpha \delta_k Q_{ij}$ and $\Delta b_{0k} = \alpha \delta_k$

Then, send **δ_k** back to the hidden layer.

Back Propagation (Training Algorithm)

- Step 8 – Now each hidden unit will be the sum of its delta inputs from the output units-

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k \, w_{jk}$$

Error term can be calculated as follows – $\delta_j = \delta_{inj} f'(Q_{inj})$

On this basis, update the weight and bias as follows – $\Delta w_{ij} = \alpha \delta_j x_i$ and $\Delta b_{0j} = \alpha \delta_j$

Phase 3

- **Step 9** – Each output unit ***(y*$_k$*k = 1 to m)*** updates the weight and bias as follows –

$$v_{jk}(new) = v_{jk}(old) + \Delta v_{jk}$$

$$b_{0k}(new) = b_{0k}(old) + \Delta b_{0k}$$

Back Propagation (Training Algorithm)

- **Step 10** – Each output unit ***(z_j, $j = 1$ to p)*** updates the weight and bias as follows –

$$w_{ij}(new) = w_{ij}(old) + \Delta w_{ij}$$

$$b_{0j}(new) = b_{0j}(old) + \Delta b_{0j}$$

- **Step 11** – Check for the stopping condition, which may be either the number of epochs reached or the target output matches the actual output.

Back Propagation (Training Algorithm)

Generalized Delta Learning Rule

- Delta rule works only for the output layer. On the other hand, generalized delta rule, also called as back-propagation rule, is a way of creating the desired values of the hidden layer.
- Mathematical Formulation
- For the activation function $y_k=f(y_{ink})$ the derivation of net input on Hidden layer as well as on output layer can be given

$$y_{ink} = \sum_i z_i w_{jk} \text{ and } \qquad y_{inj} = \sum_i x_i v_{ij}$$

- Now the error which has to be minimized is

$$E = \frac{1}{2}\sum_k [t_k - y_k]^2$$

Back Propagation (Training Algorithm)

- Now By using the chain rule, we have

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}\sum_k [t_k - y_k]^2\right)$$

$$= \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}[t_k - t(y_{ink})]^2\right)$$

$$= -[t_k - y_k]\frac{\partial}{\partial w_{jk}} f(y_{ink})$$

$$= -[t_k - y_k] f(y_{ink}) \frac{\partial}{\partial w_{jk}}(y_{ink})$$

$$= -[t_k - y_k] f^{'}(y_{ink}) z_j$$

Back Propagation (Training Algorithm)

- Now let us say, $\delta_k = -[t_k - y_k]f'(y_{ink})$
- The weights on connections to the hidden unit z_j can be given by −

$$\frac{\partial E}{\partial v_{ij}} = -\sum_k \delta_k \frac{\partial}{\partial v_{ij}}(y_{ink})$$

- Putting the value of y_{ink} we will get the following- $\delta_j = -\sum_k \delta_k w_{jk} f'(z_{inj})$
- Weight updating can be done as follows −

For the output unit −

$$\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}}$$

$$= \alpha\, \delta_j\, x_i$$

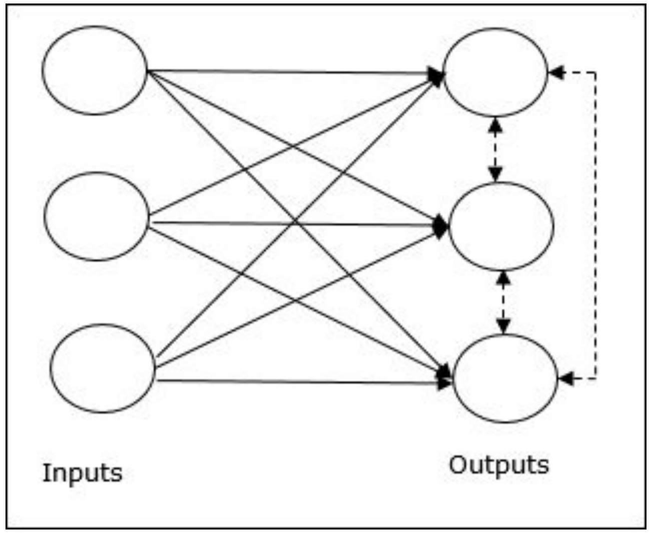
For the hidden unit −

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}}$$

$$= \alpha\, \delta_k\, z_j$$

Competitive Learning in NN

- It is concerned with unsupervised training in which the output nodes try to compete with each other to represent the input pattern. To understand this learning rule we will have to understand competitive net which is explained as follows −
- Basic Concept of Competitive Network
  - This network is just like a single layer feed-forward network having feedback connection between the outputs. The connections between the outputs are inhibitory type, which is shown by dotted lines, which means the competitors never support themselves.



Basic Concept of Competitive Learning Rule

- There would be competition among the output nodes
  - During training, the output unit that has the highest activation to a given input pattern, will be declared the winner. This rule is also called Winner-takes-all because only the winning neuron is updated and the rest of the neurons are left unchanged.
- Mathematical Formulation

  Following are the three important factors for mathematical formulation of this learning rule −

  - Condition to be a winner: Suppose if a neuron ***y_k*** wants to be the winner, then there would be the following condition

$$y_k = \begin{cases} 1 & if\, v_k > v_j \, for\, all\;\; j,\, j \neq k \\ 0 & otherwise \end{cases}$$

    - It means that if any neuron, say, y_k wants to win, v_k, must be the largest among all the other neurons in the network.

Basic Concept of Competitive Learning Rule

- Condition of the sum total of weight
  - Another constraint over the competitive learning rule is the sum total of weights to a particular output neuron is going to be 1. For example, if we consider neuron k then

$$\sum_{k} w_{kj} = 1 \quad for\, all\;\; k$$

- Change of weight for the winner
  - If a neuron does not respond to the input pattern, then no learning takes place in that neuron. However, if a particular neuron wins, then the corresponding weights are adjusted as follows –

$$\Delta w_{kj} = \begin{cases} -\alpha(x_j - w_{kj}), & if\, neuron\, k\, wins \\ 0 & if\, neuron\, k\, losses \end{cases}$$

Here α is the learning rate.

This clearly shows that we are favoring the winning neuron by adjusting its weight and if a neuron is lost, then we need not bother to re-adjust its weight.

THE END