EID403: MACHINE LEARNING

Abhishikta Ummadi

GITAM Hyderabad

September 7, 2019

Outline

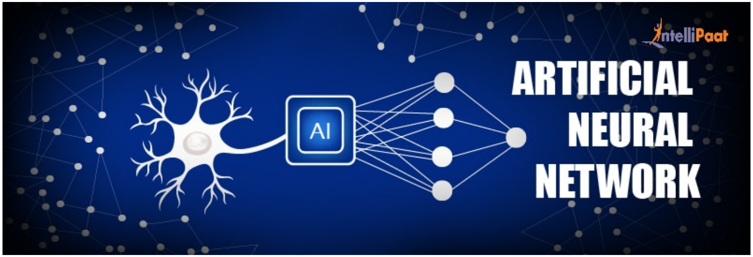
1. Artificial Neural Networks
   - Introduction
     - Biological Neurons
   - Neural Network Representations
   - Appropriate Problems For Neural Network Learning
   - Perceptrons
   - Gradient Descent and the Delta Rule
2. Multi Layer Feed Forward Neural Networks
3. Backpropagation
4. Advanced Topics in Neural Networks

Artificial Neural Networks

- Neurons and Biological Motivation
- Neural Network Representations
- Problems for Neural Network Learning
- Perceptrons
- Multilayer Networks
- Back Propagation Algorithm

Neural nets can be used to answer the following:

- Pattern recognition: Does that image contain a face?
- Classification problems: Is this cell defective?
- Prediction: Given these symptoms, the patient has disease X.
- Forecasting: predicting behavior of stock market.
- Handwriting: is character recognized?
- Optimization: Find the shortest path for the TSP.

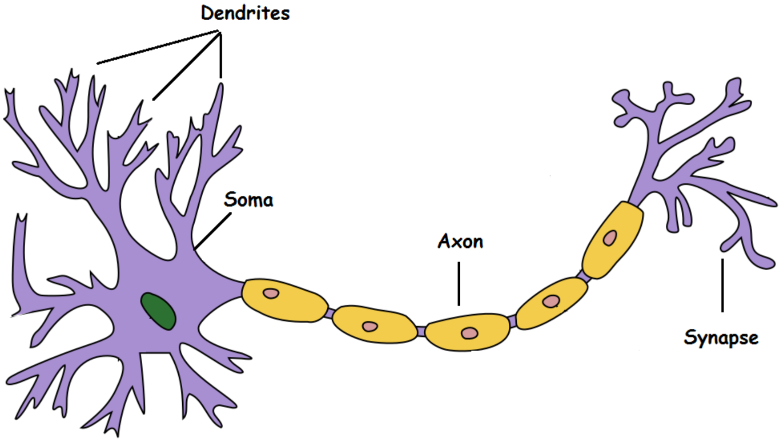


Roots of work on Neural Networks(NNs) are in:

- Neurological studies:
  - How do nerves behave when stimulated by different magnitudes of electric current?
  - Is there a minimal threshold needed for nerves to be activated?
  - How do different nerve cells communicate among each other?
- Psychological studies:
  - How do animals learn, forget, recognize and perform various types of tasks?
- Psycho-physical: Experiments help to understand how individual neurons and groups of neurons work.
- McCulloch and Pitts introduced the first mathematical model of single neuron, widely applied in subsequent work.

Biological Neurons

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons.



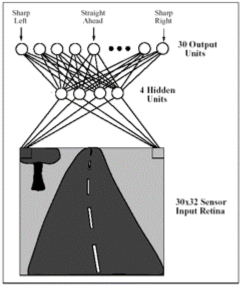
- Human information processing system consists of brain neuron: basic building block.
- Neuron: Cell that communicates information to and from various parts of body.
- Simplest model of a neuron: considered as a threshold unit – a processing element (PE).
- ANNs are built out of interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

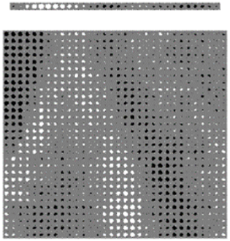
- Interconnected network of approximately $\sim 10^{11}$ neurons.
- Connections per neuron: $\sim 10^4$.
- Neuron switching time $\sim 10^{-3}$ seconds.
- Computer switching speeds : $\sim 10^{-10}$ seconds.
- Scene recognition time for human: $\sim 10^{-1}$ second.
- One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations.

Neural Network Representation

- A prototypical example of ANN learning is provided by Pomerleau's (1993) system **ALVINN**, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.
- Input: The input to the neural network is a $30x32$ grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- Output: The network output is the direction in which the vehicle is steered.
- Training: ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes.
- ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways.

Neural network learning to steer an autonomous vehicle





Network of ALVINN system

- Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.
- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image.
- These are called **"hidden" units** because their output is available only within the network and is not available as part of the global network output.
- Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs.
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit.
- Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

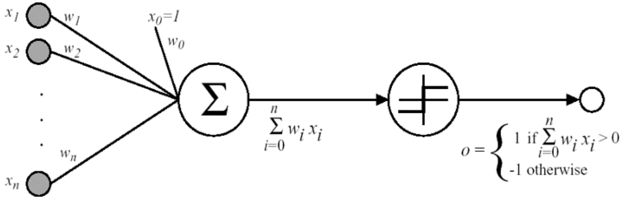
ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

- Input is high-dimensional discrete or real-valued : instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example.
- Target function output is discrete or real valued or a vector of several real- or discrete-valued attributes.: For example, in the ALVINN system the output is a vector of 30 attributes.
- The training examples may contain errors: ANN learning methods are quite robust to noise in the training data.

- Long training times are acceptable:NN algorithms require longer training times than, say, decision tree learning algorithms.
- Fast evaluation of the learned target function may be required:For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- Human readability of result is unimportant:Learned neural networks are less easily communicated to humans than learned rules.

Perceptrons

- One type of ANN system is based on a unit called a perceptron.
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- More precisely, given inputs x_1 through x_n,, the output $o(x_1, ..., x_n)$ computed by the perceptron is


x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

where each w_i is a real-valued constant, or weight, that determines the contribution of input w_i to the perceptron output.

- To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w}.\vec{x} > 0$.
- We will sometimes write the perceptron function as

  $$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

  where

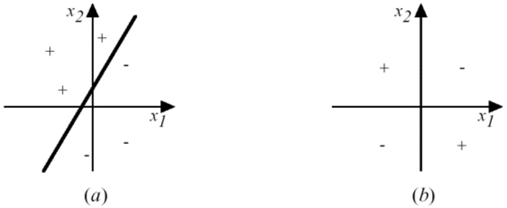
  $$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

- Learning a perceptron involves choosing values for the weights w_0, . . . , w_n.
- Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Representational Power of Perceptrons

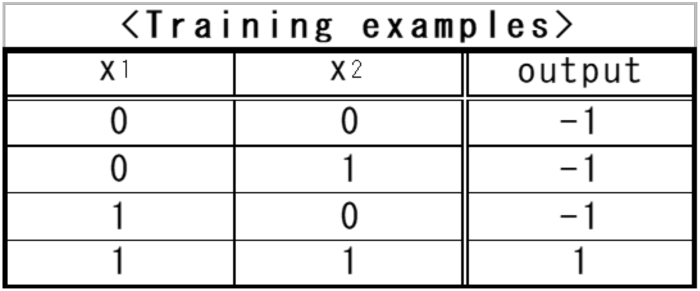
- The perceptron can be viewed by representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points).
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side.



(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable. x_l and x_2 are the Perceptron inputs.

- The equation for this decision hyperplane is $\vec{w}.\vec{x} = 0$.
- Those positive and negative examples that can be separated by hyperplane are called**linearly separable** sets of examples.
- A single perceptron can be used to represent many boolean functions.
- Perceptrons can represent all of the primitive boolean functions AND, OR, NAND ($\neg$ AND), and NOR ($\neg$ OR).
- Some Boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_l \neq x_2$.
- The set of linearly non-separable training examples shown in Figure-(b) corresponds to this XOR function.

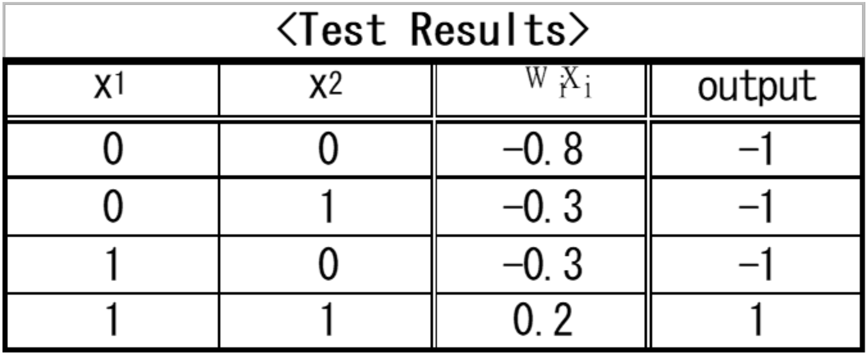
- For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights w_o = -0.8, and $w_1 = w_2 =$ 0.5.
- This perceptron can be made to represent the OR function instead by altering the threshold to w_o =-0.3.
- **AND function(linearly separable)** :



<Training examples>

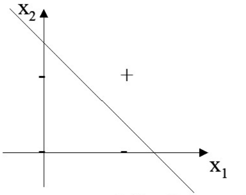
x1	x2	output
0	0	−1
0	1	−1
1	0	−1
1	1	1

Decision hyper plane : $w_0 + w_1x_1 + w_2x_2 = 0$ $-0.8 + 0.5x_1 + 0.5x_2 = 0$



<Test Results>

x1	x2	$W_i X_i$	output
0	0	−0.8	−1
0	1	−0.3	−1
1	0	−0.3	−1
1	1	0.2	1



$$-0.8 + 0.5\,x_1 + 0.5\,x_2 = 0$$

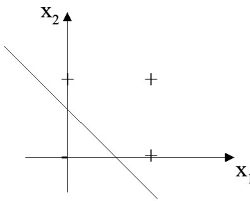
OR function (linearly separable):

<Training examples>

x_1	x_2	output
0	0	-1
0	1	1
1	0	1
1	1	1

<Test Results>

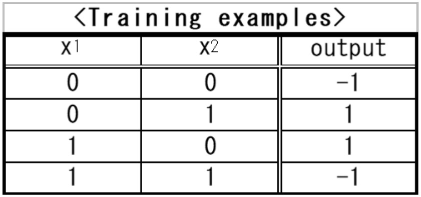
x_1	x_2	$\Sigma w_i x_i$	output
0	0	-0.3	-1
0	1	0.2	-1
1	0	0.2	-1
1	1	0.7	1



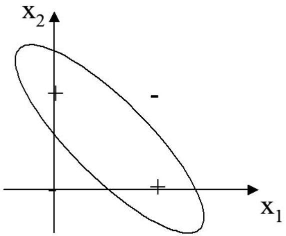
$-0.3 + 0.5\,x_1 + 0.5\,x_2 = 0$

Decision hyperplane : w0 + w1 x1 + w2 x2 = 0 -0.3 + 0.5 x1 + 0.5 x2

XOR function(Non linearly separable) :



<Training examples>		
x_1	x_2	output
0	0	−1
0	1	1
1	0	1
1	1	−1



- The ability of perceptrons to represent AND, OR, NAND, and NOR is important because every Boolean function can be represented by some network of interconnected units based on these primitives.
- Every Boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.
- One way is to represent the Boolean function in disjunctive normal form.

The Perceptron Training Rule

How to learn the weights for a single perceptron?

- The precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ± 1 output for each of the given training examples.
- Several algorithms are known to solve this learning problem. Here we consider two: **The perceptron rule and The delta rule**.
- One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.

- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(t - o)x_i$$

  Here t is the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the learning rate.
- The role of the learning rate is to moderate the degree to which weights are changed at each step.
- It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values?
consider some specific cases:

- Suppose the training example is correctly classified already by the perceptron. In this case, (t - o) is zero, making Δw_i zero, so that no weights are updated.
- Suppose the perceptron outputs a -1,when the target output is $+ 1$. To make the perceptron output a $+ 1$ instead of - 1 in this case, the weights must be altered to increase the value of $\vec{w}.\vec{x}$.
- For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying this example.
- Notice the training rule will increase w, in this case, because (t - o), η, and x_i are all positive.

- For example, if x_i = .8, η = 0.1, t = 1, and o = - 1, then the weight update will be

$$\Delta w_i = \eta(t - o)x_i = 0.1(1 - (1))0.8 = 0.16$$

- On the other hand, if t = -1 and o = 1, then weights associated with positive x_i will be decreased rather than increased.
- Perceptron training rule can converge to the target function provided the training examples are linearly separable and provided a sufficiently small η is used.
- If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

- Perceptron rule fail to converge if the examples are not linearly separable.
- If the training examples are not linearly separable, then the **delta rule** converges toward a best-fit approximation to the target concept.
- Key idea is to use **gradient descent** to search the hypothesis space of possible weight vectors to find weights that best fit the training data.
- Gradient descent provides basis for **BACKPROPAGATION algortihm**.
- Backpropagation algo. can learn networks with many interconnected units.

- The delta training rule is best understood by considering the task of training an *unthresholded* perceptron; that is, a *linear unit* for which the output O is given by

  $$o(\vec{x}) = \vec{w} \cdot \vec{x} \qquad (1)$$

  Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.
- To derive a weight learning rule for linear units, specify a measure for the training error of a hypothesis (weight vector), relative to the training examples.

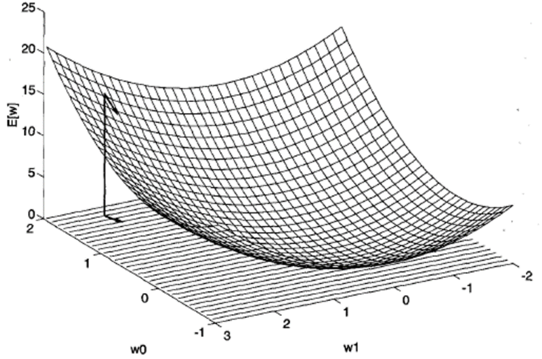
- There are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{2}$$

  where D is the set of training examples, t_d is the target output for training example d, and O_d is the output of the linear unit for training example d.
- Thus, $E(\vec{w})$ is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.
- Here we characterize E as a function of $\vec{w}$, because the linear unit output o depends on this weight vector.

VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, visualize the entire hypothesis space of possible weight vectors and their associated E values.



- The axes w_0 and w_l represent possible values for the two weights of a simple linear unit.
- The w_0,w_1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.

The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space. (i.e. minimum error)

- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.
- The specific parabola will depend on the particular set of training examples.

- Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
- At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface as depicted in Figure. This process continues until the global minimum error is reached.

DERIVATION OF THE GRADIENT DESCENT RULE

How can we calculate the direction of steepest descent along the error surface?

- This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$.
- This vector derivative is called the gradient of E with respect to $\vec{w}$, written $\bigtriangledown E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{3}$$

  Notice $\bigtriangledown E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i.
- When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E.
- The negative of this vector therefore gives the direction of steepest decrease.

- Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

  $$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

  where

  $$\vec{w} = -\eta \bigtriangledown E(\vec{w}) \qquad (4)$$

  The negative sign is present because we want to move the weight vector in the direction that decreases E.

- This training rule can also be written in its component form:

  $$w_i \leftarrow w_i + \Delta w_i$$

  where

  $$\Delta w_i = -\eta \frac{\delta E}{\delta w_i} \qquad (5)$$

  which makes it clear that steepest descent is achieved by altering each component w_i, of $\vec{w}$ in proportion to $\frac{\delta E}{\delta w_i}$.

To construct a practical algorithm for iteratively updating weights according to Equation (5), we need an efficient way of calculating the gradient at each step.

- The vector of $\frac{\delta E}{\delta w_i}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned} \tag{6}$$

where x_{id} denotes the single input component x_i for training example d.

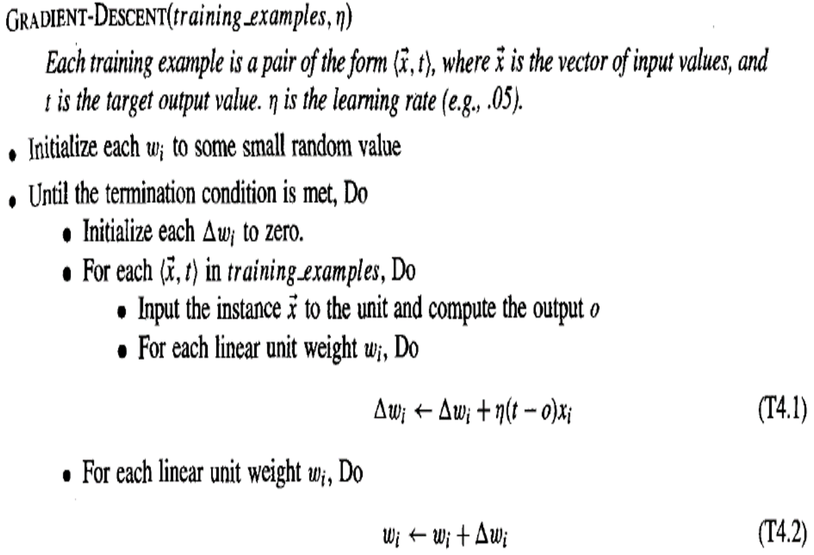
We now have an equation that gives $\frac{\delta E}{\delta w_i}$ in terms of the linear unit inputs x_{id}, outputs O_d, and target values t_d associated with the training examples.
Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \qquad (7)$$

Summarizing gradient descent algorithm for training linear units

- Pick an initial random weight vector. Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process.
- Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used.
- If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it.
- For this reason, one common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows.

Gradient Descent Algorithm

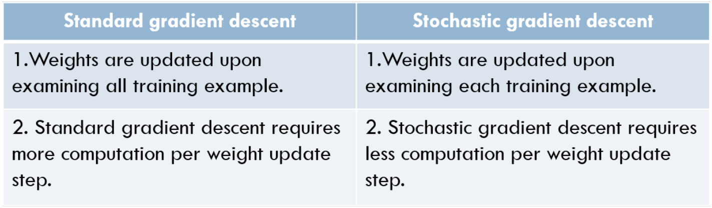


```
GRADIENT-DESCENT($training\_examples, \eta$)

    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).

• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
    • Initialize each $\Delta w_i$ to zero.
    • For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
        • Input the instance $\vec{x}$ to the unit and compute the output $o$
        • For each linear unit weight $w_i$, Do

                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$        (T4.1)

    • For each linear unit weight $w_i$, Do

                $w_i \leftarrow w_i + \Delta w_i$        (T4.2)
```

Stochastic Approximation to Gradient Descent

- The key practical difficulties in applying gradient descent are:
  - Converging to a local minimum can sometimes be quite slow
  - If there are multiple local minima in the error surface, no guarantee that the procedure will find the global minimum.
- One common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively **stochastic gradient descent**.
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example.

The key differences between standard gradient descent and stochastic gradient descent are:



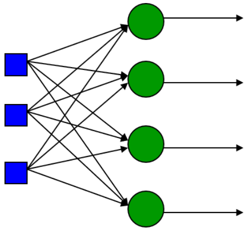
Standard gradient descent	Stochastic gradient descent
1.Weights are updated upon examining all training example.	1.Weights are updated upon examining each training example.
2. Standard gradient descent requires more computation per weight update step.	2. Stochastic gradient descent requires less computation per weight update step.

- The training rule $\Delta w_i = \eta(t - o)x_i$ is known as the delta rule, or sometimes the LMS (least-mean-square) rule, Adaline rule, or Widrow-Hoff rule (after its inventors).

Remarks

- The key difference between these algorithms is that the perceptron training rule updates weights based on the error in the thresholded perceptron output,whereas the delta rule updates weights based on the error in the unthresholded linear combination of inputs
- The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable
- The delta rule converges only asymptotically toward the minimum error hypothesis, requiring unbounded time, but converges regardless of whether the training data are linearly separable
- Linear programming like perceptron training rule does not scale to Multilayer neural networks.
- the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks

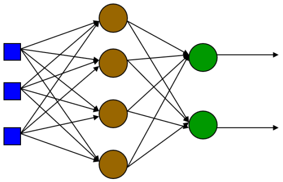
Single Layer Perceptron



- Single perceptrons can only express linear decision surfaces. In Where as multi-layer neural networks learned by the back-propagation algorithm are capable of expressing a rich variety of nonlinear decision surfaces

Multi layer feed-forward

3-4-2 Network



Input layer of neurons

Output layer of neurons

Hidden layer

Multi-layer feed-forward networks

- Perceptrons are rather weak as computing models since they can only learn linearly-separable functions.
- Thus, we now focus on multi-layer, feed forward networks of non-linear sigmoid units: i.e.,

$$\mathbf{g(x)} = 1/1 + e^{-x}$$

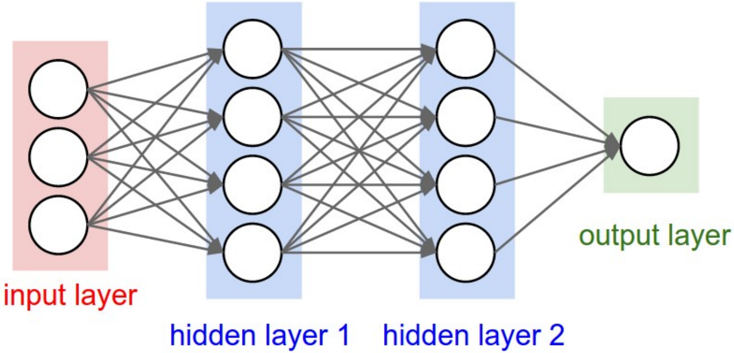
Multi-layer feed-forward networks

Multi-layer, feed forward networks extend perceptrons i.e., 1-layer networks into n-layer by:

- Partition units into layers 0 to L such that;
- lowermost layer number, layer 0 indicates the input units
- topmost layer numbered L contains the output units.
- layers numbered 1 to L are the hidden layers.
- Connectivity means bottom-up connections only, with no cycles, hence the name "feed-forward" nets
- Input layers transmit input values to hidden layer nodes hence do not perform any computation.

Note: layer number indicates the distance of a node from the input nodes.

Multilayer feed forward network

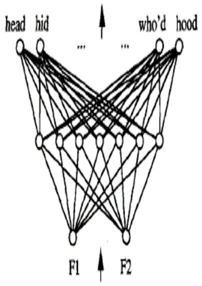


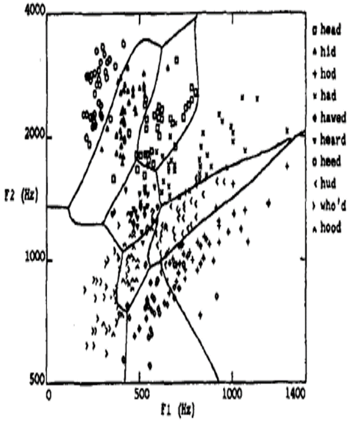
Hidden Units

- Hidden units are nodes that are situated between the input nodes and the output nodes.
- Given too many hidden units, a neural net will simply memorize the input patterns.
- Given too few hidden units, the network may not be able to represent all the necessary representations.

Multi-layer feed-forward networks

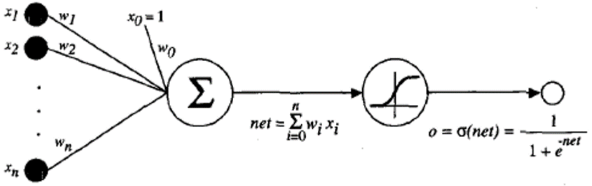
- The kind of multilayer networks learned by the **BACKPROPACATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces.
- Back-propagation learning is a gradient descent search through the parameter space to minimize the sum-of-squares error.
    - Most common algorithm for learning algorithms in multilayer networks.
- It is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces.
- For example, in following figure. the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., “hid”, “had”, “head”, “hood”, etc.).





The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

Multilayer Neworks


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

- Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of sigmoid unit, however, the threshold output is a continuous function of its input. The sigmoid function $\sigma(x)$ is also called the **logistic function**.

The sigmoid unit computes its output O as

σ is often called the sigmoid function or, alternatively, the logistic function.

- Its output ranges between 0 and 1, increasing monotonically with its input.
- The sigmoid unit computes its output o as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

The sigmoid function has the useful property that its derivative is easily expressed in terms of its output

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

Back-propagation Learning

- The BP algorithm learns the weights for a multi-layer network, given a network with a fixed set of units and interconnections. It employs a gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- Inputs:
  - Network topology: includes all units & their connections
  - Some termination criteria
  - Learning Rate (constant of proportionality of gradient descent search)
  - Initial parameter values
  - A set of classified training data
- Considering networks with multiple output units rather than single units, redefine E to sum the errors over all of the network output units.
- Error is calculated as

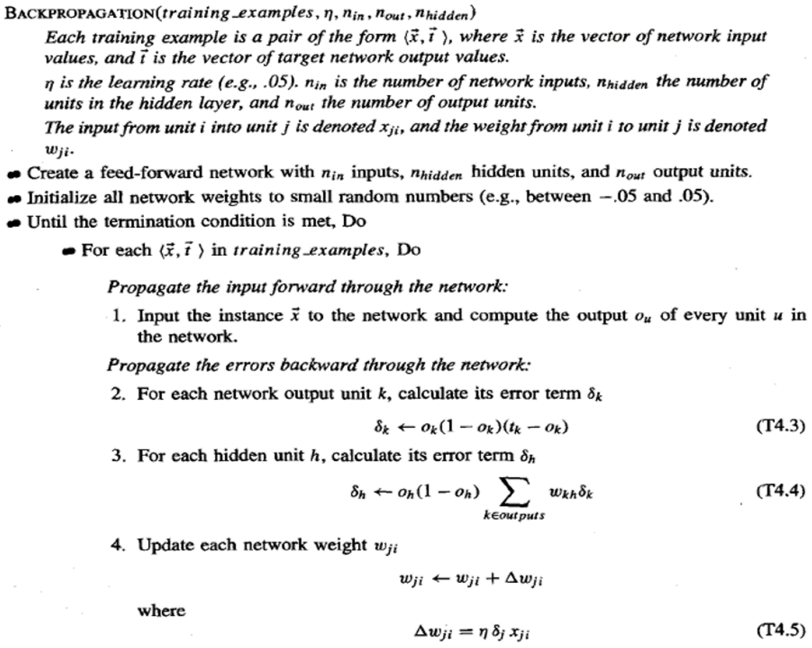
$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd}-o_{kd})^2$$

Learning in backprop

- Learning in backprop is similar to learning with perceptrons, i.e.,
    - Example inputs are fed to the network.
        - If the network computes an output vector that matches the target, nothing is done.
        - If there is a difference between output and target (i.e., an error), then the weights are adjusted to reduce this error.
        - The key is to assess the blame for the error and divide it among the contributing weights.
    - The error term (T - O) is known for the units in the output layer. To adjust the weights between the hidden and the output layer, the gradient descent rule can be applied as done for perceptrons.
    - To adjust weights between the input and hidden layer some way of estimating the errors made by the hidden units in needed.

Gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error

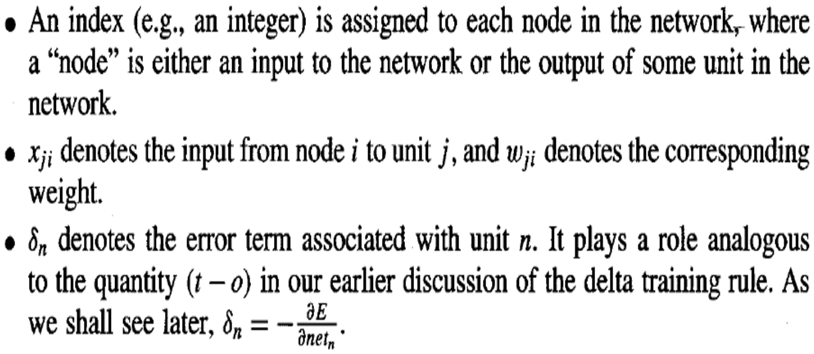
- One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface.
- Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error.
- Despite this obstacle, in practice BACKPROPAGATION been found to produce excellent results in many real-world applications.



```
BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)
    Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input
    values, and $\vec{t}$ is the vector of target network output values.
    $\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of
    units in the hidden layer, and $n_{out}$ the number of output units.
    The input from unit $i$ into unit $j$ is denoted $x_{ji}$, and the weight from unit $i$ to unit $j$ is denoted
    $w_{ji}$.
• Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
• Initialize all network weights to small random numbers (e.g., between −.05 and .05).
• Until the termination condition is met, Do
    • For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do
        Propagate the input forward through the network:
        1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in
           the network.
        Propagate the errors backward through the network:
        2. For each network output unit $k$, calculate its error term $\delta_k$
               $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$                                  (T4.3)
        3. For each hidden unit $h$, calculate its error term $\delta_h$
               $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$          (T4.4)
        4. Update each network weight $w_{ji}$
               $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
           where
               $\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$                                    (T4.5)
```

- The algorithm is applied to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer.
- This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION.

- The notation used are



- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Working of algorithm

- The algorithm starts by constructing a network with the desired no of hidden input and output units.
- Initializing all network weights to small random values.
- The main loop of the algorithm then repeatedly iterates over the training examples.
- For each training example, it applies the network to the example , calculates the error of the network output for this example , then update all the weights in the network.
- The gradient descent weight-update rule ($\Delta w_{ji} = \eta \delta_j x_{ji}$) ,updates each weight in proportion to the learning rate η,the input value x_{ji} to which the weight is applied, and the error in the output of the unit.

- The exact form of δ_j follows from the derivation of the weight-tuning rule.
- δ_k is computed for each network output unit k as :
$$\delta_k = O_k(1 - O_k)(t_k - O_k)$$
The factor $O_k(1 - O_k)$ is derivative of sigmoid squashing function.
- δ_h value for each hidden unit h is calculated as
$$\delta_h = O_h(1 - O_h) \sum_{k \epsilon outputs} W - kh\delta_k$$
- Training examples provide target values tk only for network output,no target values are directly available to indicate the error of hidden unit's values.

- Error term for hidden unit h is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of the δ_k's by W_{kh}, the weight from hidden unit h to output unit k.
- This weight characterizes the degree to which from hidden unit h is ' responsible for' the error in output unit k.
- The algorithm updates weights incrementally , following the presentation of each training example.
- The obtain the true gradient of E, one would sum the $\delta_j x_{ij}$ values over all training examples before altering weight values.

- The weigh-update loop in BackPropagation may be iterated thousands of times in a typical application.
- The termination conditions are
  - i) after fixed number of iterations through the loop or,
  - ii) once the error on the training examples falls below some threshold or,
  - iii) once the error on a separate validation set of examples meet some criterion.
- The choice of termination plays important role, as few no of iterations can fail to reduce error and too many can lead to overfitting the training data.

ADDING MOMENTUM

- We can alter the weigh-update rule in $\Delta w_{ji} = \eta\delta_j x_{ji}$ in the algorithm by making the weight update on the n^{th} iteration depend partially on the update that occurred during (n-1)th iteration, as follows :

$$\Delta w_{ji}(n) = \eta\, \delta_j\, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- Where $0 \leq \alpha \leq 1$ is constant, called momemtum.
- $\Delta w_{ji}(n)$ is the weight update performed during the nth iteration.
- On RHS of the equation :
  - First term is just weight-update rule of equation $\Delta w_{ji} = \eta\delta_j x_{ji}$ in Backpropagation algorithm.
  - Second term is new and called as momentum term.

LEARNING IN ARBITRARY ACYCLIC NETWORKS

- Above mentioned algorithm is applied to two layer networks, it easily generalizes to feed forward networks of arbitrary depth. We need to update it.
- The weight update rule : $\Delta w_{ji} = \eta \delta_j x_{ji}$ is used ,with small changes in computing of δ values.
- δ_r value for a unit 'r' in layer 'm' is computed from δ values at next deeper layer m+1 according to :

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr}\, \delta_s$$

- This is similar to error term for hidden unit in algorithm. That means, this step may be repeated for any number of hidden layers in network.
- We can generalize the algorithm to any directed acyclic graph,regardless of whether the network units are arranged in uniform layers or not.

- If network units are not uniformly arranged, we calculate error for any internal unit as

$$\delta_r = o_r\,(1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\,\delta_s$$

- Downstream(r) is the set of units immediately downstream from units r in the network: i.e. whose inputs include the output of unit 'r'.

Derivation of the BACKPROPAGATION Rule

- We will derive the **weight-tuning rule** of Back-propagation algorithm.
- First we will derive the stochastic gradient descent rule implemented by the algorithm.
- The stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d w.r.t this single example.

- For each training example d every weight w_{ji} is updated by adding to Δw_{ji}.

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \qquad \text{-(1.1)}$$

where E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- Here, outputs is the set of units in the network, t_k is the target value of unit k for training example d

Subscripts and variables used in notation of stochastic gradient descent rule:

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum w_{ji}x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j, = the target output for unit j
- σ = the sigmoid function
- outputs = the set of units in the final layer of the network
- Downstream(j) = the set of units whose immediate inputs include the output of unit j

Downstream(j) = the set of units whose immediate inputs include the 0utput of unit j

- In order to implement the stochastic gradient rule , we will derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$.
- The weight w_{ji} can influence the rest of network only through net_j.
- Using chain rule :

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji}$$

-(1.2)

- Now we have to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

Case 1: Training Rule for Output Unit Weights

We consider two cases in turn: the case where unit j is an output unit for the network, and the case where j is an internal unit.

- Case 1: Training Rule for Output Unit Weights: Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

- (1)

To begin, consider just the first term in Equation 1

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

- The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when k = j. We therefore drop the summation over output units and simply set k = j.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned}$$

-(2)

- Next consider the second term in Equation 1. Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(l - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned}$$

-(3)

- Substituting expressions (2) and (3) into (1), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j)$$

- and combining this with Equations (1.1) and (1.2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j)x_{ji}$$

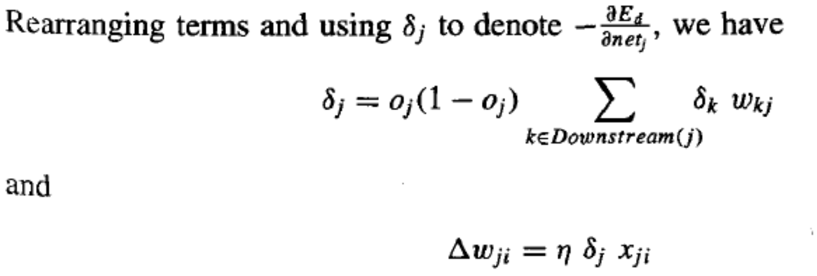
- Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm. Furthermore, now S_k in Equation (T4.3) is equal to the quantity $-\frac{\partial E_d}{\partial net_k}$

Case 2: Training Rule for Hidden Unit Weights

- In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d.
- For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network.
- We denote this set of units by Downstream(j). Notice that net_j can influence the network outputs only through the units in Downstream(j).

Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned}$$



Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \; w_{kj}$$

and

$$\Delta w_{ji} = \eta \; \delta_j \; x_{ji}$$

- Notice Equation- error term for output unit is just a special case of this rule, in which Downstream(j) = outputs.

ADVANCED TOPICS IN ARTIFICIAL NEURAL NETWORKS

Alternative Error Functions

- Gradient descent can be performed for any function E that is differentiable with respect to the parameterized hypothesis space.
- While the basic BACKPROPAGATION algorithm defines E in terms of the sum of squared errors of the network, other definitions have been suggested in order to incorporate other constraints into the weight-tuning rule.
- For each new definition of E a new weight-tuning rule for gradient descent must be derived.

Examples of alternative definitions of E include

- *Adding a penalty term for weight magnitude*.
    - We can add a term to E that increases with the magnitude of the weight vector.
    - This causes the gradient descent search to seek weight vectors with small magnitudes, thereby reducing the risk of overfitting.
    - One way to do this is to redefine E as

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

which yields a weight update rule identical to the BACKPROPAGATION rule, except that each weight is multiplied by the constant $(1 - 2\gamma\eta)$ upon each iteration.

- *Adding a term for errors in the slope, or derivative of the target function.*
    - In some cases, training information may be available regarding desired derivatives of the target function, as well as desired values.
    - For example, Simard et al. (1992) describe an application to character recognition in which certain training derivatives are used to constrain the network to learn character recognition functions that are invariant of translation within the image.
    - Mitchell and Thrun (1993) describe methods for calculating training derivatives based on the learner's prior knowledge.

- In both of these systems, the error function is modified to add a term measuring the discrepancy between these training derivatives and the actual derivatives of the learned network.
- One example of such an error function is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Here x_d^j denotes the value of the jth input unit for training example d. Thus, $\frac{\partial t_{kd}}{\partial x_j^d}$ is the training derivative describing how the target output value t_{kd} should vary with a change in the input x_d^j. Similarly, $\frac{\partial o_{kd}}{\partial x_d}$ denotes the corresponding derivative of the actual learned network. The constant μ determines the relative weight placed on fitting the training values versus the training derivatives.

- *Minimizing the cross entropy of the network with respect to the target values*
    - Consider learning a probabilistic function, such as predicting whether a loan applicant will pay back a loan based on attributes such as the applicant's age and bank balance.
    - Although the training examples exhibit only boolean target values, the underlying target function might be best modeled by outputting the probability that the given applicant will repay the loan, rather than attempting to output the actual 1 and 0 value for each input instance.
    - Given such situations in which we wish for the network to output probability estimates, it can be shown that the best (i.e., maximum likelihood) probability estimates are given by the network that minimizes the cross entropy, defined as

$$-\sum_{d\in D} t_d \log o_d + (1-t_d)\log(1-o_d)$$

- *Altering the effective error function by weight sharing*.
  - By weight sharing, or "tying together" weights associated with different units or inputs.
  - The idea here is that different network weights are forced to take on identical values, usually to enforce some constraint known in advance to the human designer.
  - For example, an application of neural networks to speech recognition, in which the network inputs are the speech frequency components at different times within a 144 millisecond time window.
  - One assumption that can be made in this application is that the frequency components that identify a specific sound (e.g., "eee") should be independent of the exact time that the sound occurs within the 144 millisecond window.

- *Altering the effective error function by weight sharing.(cont.)*
    - To enforce this constraint, the various units that receive input from different portions of the time window are forced to share weights.
    - The net effect is to constrain the space of potential hypotheses, thereby reducing the risk of overfitting and improving the chances for accurately generalizing to unseen situations.
    - Such weight sharing is typically implemented by first updating each of the shared weights separately within each unit that uses the weight, then replacing each instance of the shared weight by the mean of their values.
    - The result of this procedure is that shared weights effectively adapt to a different error function than do the unshared weights.

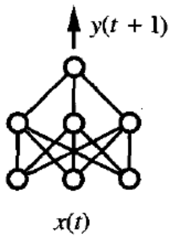
Alternative Error Minimization Procedures

- Gradient descent is not always the most efficient.
- It is not uncommon for BACKPROPAGATION to require tens of thousands of iterations through the weight update loop when training complex networks.
- For this reason, a number of alternative weight optimization algorithms have been proposed and explored.
- Other possibilities, weight update method as involving two decisions:
  - choosing a direction in which to alter the current weight vector
  - choosing a distance to move

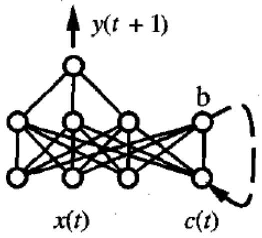
- In BACKPROPAGATION, the direction is chosen by taking the negative of the gradient, and the distance is determined by the learning rate constant η.
- One optimization method, known as line search, involves a different approach to choosing the distance for the weight update.
- A second method, that builds on the idea of line search, is called the conjugate gradient method.

Recurrent Networks

- Recurrent networks are artificial neural networks that apply to **time series data** and that use outputs of network units at time t as the input to other units at time t + 1.
- They support a form of directed cycles in the network.
- Example:
  - Consider the time series prediction task of predicting the next day's stock market average y(t + 1) based on the current day's economic indicators x(t).
  - Given a time series of such data, one obvious approach is to train a feedforward network to predict y(t + 1) as its output, based on the input values x(t).
  - One such network is shown in Fig(a).

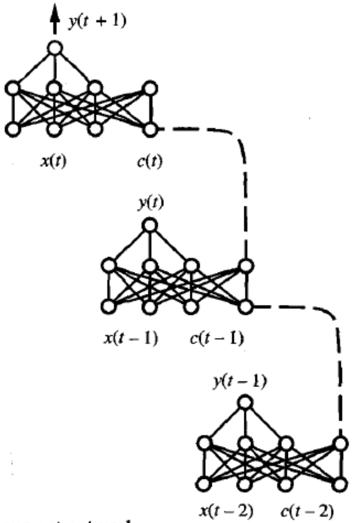


(*a*) Feedforward network



(*b*) Recurrent network

- One limitation of such a network is that the prediction of y(t + 1) depends only on x(t) and cannot capture possible dependencies of y (t + 1) on earlier values of x.
- Remedy to this difficulty is making both x(t) and x(t - 1) inputs to the feedforward network.
- If we wish the network to consider an arbitrary window of time in the past when predicting y(t + l), then a different solution is required.
- In Fig(b), we added a new unit b to the hidden layer, and new input unit c(t). The value of c(t) is defined as the value of unit b at time t - 1; that is, the input value c(t) to the network at one time step is simply copied from the value of unit b on the previous time step.



(c) Recurrent network unfolded in time

- Notice this implements a recurrence relation, in which b represents information about the history of network inputs. Because b depends on both x(t) and on c(t), it is possible for b to summarize information from earlier values of x that are arbitrarily distant in time.
- Many other network topologies also can be used to represent recurrence relations. For example, we could have inserted several layers of units between the input and unit b, and we could have added several context in parallel where we added the single units b and c.
- Figure (b) can be trained using a simple variant of BACKPROPAGATION.
- To understand how, consider Figure (c), which shows the data flow of the recurrent network "unfolded in time. Here we have made several copies of the recurrent network, replacing the feedback loop by connections between the various copies. Notice that this large unfolded network contains no cycles. Therefore, the weights in the unfolded network can be trained directly using BACKPROPAGATION.

Neural Networks: Advantages

- Distributed representations
- Simple computations
- Robust with respect to noisy data
- Robust with respect to node failure
- Empirically shown to work well for many problem domains
- Parallel processing

Neural Networks: Disadvantages

- Training is slow
- Interpretability is hard
- Network topology layouts ad hoc
- Can be hard to debug
- May converge to a local, not global, minimum of error
- Not known how to model higher-level cognitive mechanisms
- May be hard to describe a problem in terms of features with
- numerical values

Applications

- **Classification**:
  - Image recognition
  - Speech recognition
  - Diagnostic
  - Fraud detection
  - Face recognition ..
- **Regression**:
  - Forecasting (prediction on base of past history)
  - Forecasting e.g., predicting behavior of stock market
- **Pattern association**:
  - Retrieve an image from corrupted one
- **Clustering**:
  - clients profiles
  - disease subtypes

Applications

- Pronunciation: NETtalk program (Sejnowski & Rosenberg 1987) is a neural network that learns to pronounce written text: maps characters strings into phonemes (basic sound elements) for learning speech from text
- Handwritten character recognition:a network designed to read zip codes on hand-addressed envelops
- ALVINN (Pomerleau) is a neural network used to control vehicles steering direction so as to follow road by staying in the middle of its lane