Prof. Rajitha U.N

Unit-4 Artificial Neural Networks

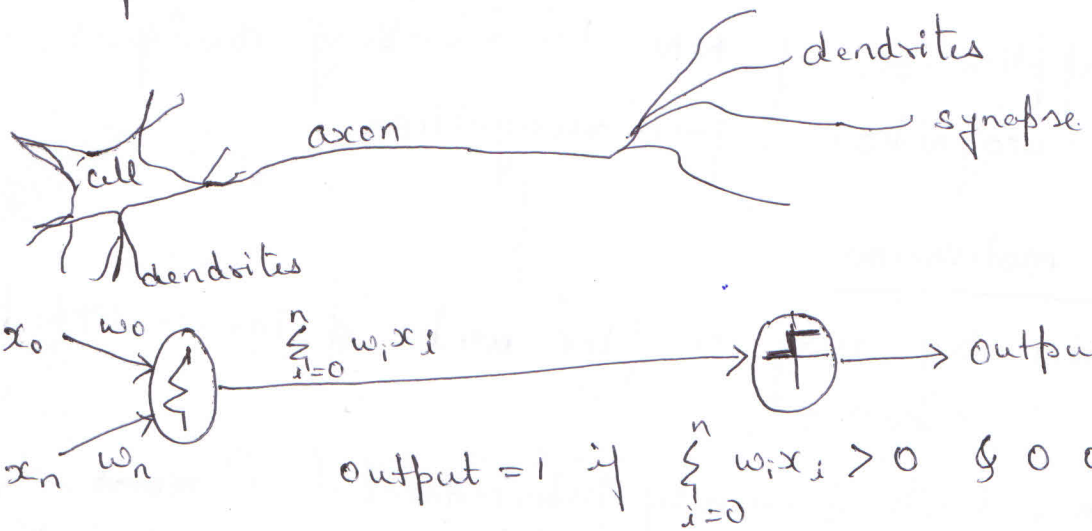
Introduction

* NN provides robust approach ~~to~~ in approximating real-valued, discrete-valued and vector-valued target functions
* Major applications of NN handwriting analysis, speech recognition, face recognition.

Biological motivation

* Inspired by very complex webs of interconnected neurons.
* ANN are built of densely interconnected ~~neurons~~ simple units, where each unit takes a number of real-valued inputs and produces a single real-valued output.
* Comparison can be associated with neurobiology (human brain)
* Human brain is a network of densely connected neurons with a switching speed of 10^{-3} seconds, which is quite slow compared to computer switching speeds of 10^{-10} sec. Yet humans are able to make surprisingly complex decisions ☺.
* One motivation for ANN system is based on highly parallel processing spanned over neurons of ~~an~~ a human brain.

* Though there is analogy between the biological neural network & ANN, they are inconsistent with each other. For Example, in ANN a single unit's value is a single constant value, whereas biological neurons output a complex time series of spikes.



Characters of Neural Network

→ Architecture
→ Method of determining weights
→ The activation unit.

When to consider Neural Network

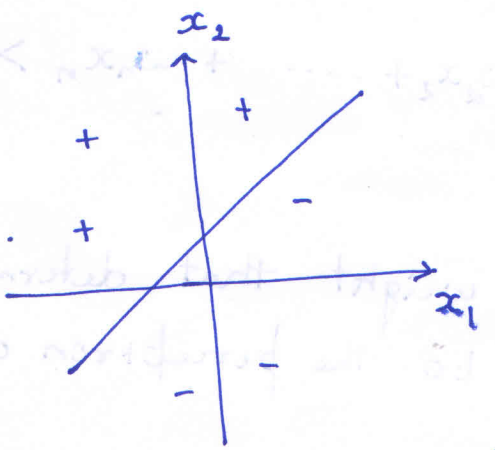
→ I/p is high dimensional (discrete or real valued)
→ Possibly noisy data
→ Form of target function is unknown

Representational power of Perceptrons

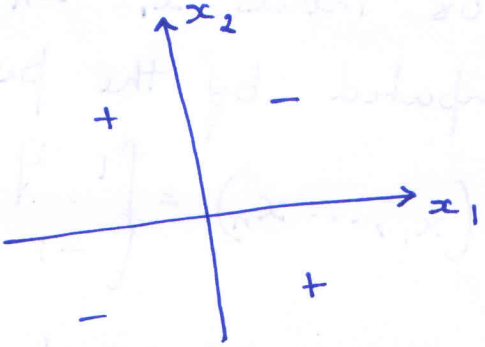
* Perceptron can be viewed as representing a hyperplane decision surface in n-dimensional space of instances.

* The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs −1 for instances lying on the other side.
  The equation for this decision is $\vec{w} \cdot \vec{x} = 0$

* Some set of examples cannot be separated by any hyperplane, the examples that can be seperated are called linearly separable set of examples.



Decision surface that classifies correctly



Set of training examples that is not linearly separable

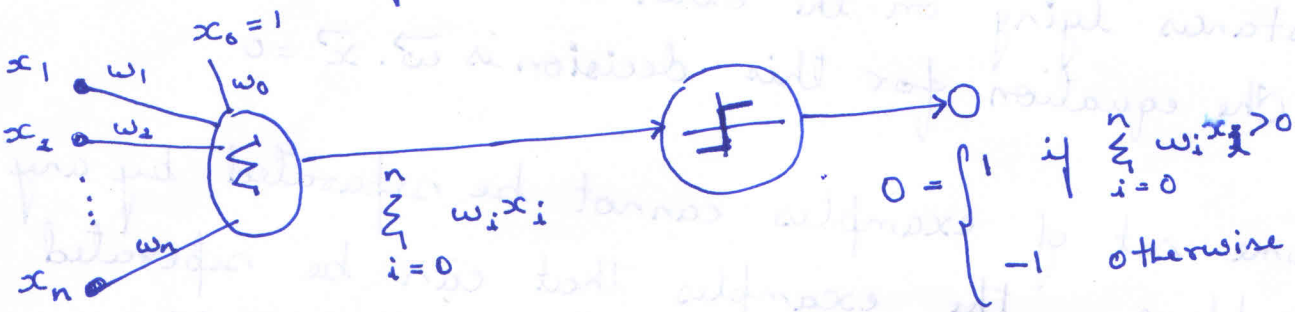
Note: x_1 & x_2 are perceptron inputs.

Psg. Rajitha U.N

Perceptrons

* In Artificial Neural Network system, a unit called Perceptron is as shown in the fig below.

* A perceptron takes a real valued inputs, calculates linear combination of these inputs. Outputs a 1 if the result is greater than some threshold and -1 otherwise.



* For inputs $x_1 \cdots x_n$, the output $o(x_1 \cdots x_n)$ computed by the perceptron is

$$o(x_1, \cdots x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

$w_i \rightarrow$ real valued constant or weight that determines the contribution of input x_i to the perceptron o/p.

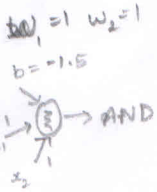
* Learning a perceptron involves choosing values for the weights $w_0 \cdots w_n$. Hence the hypotheses space H considered in perceptron is the set of all possible real-valued weight vectors.

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

Note: Perceptron is a single layer network with i/ps, wts & activation unit

A single perceptron can be used to represent many boolean functions For example, if we assume boolean values of 1 (true) and -1 (false)

AND implementation

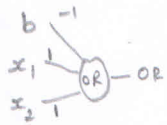


Looking back at the logic table for the A^B, we can see that we only want the neuron to output a 1 when both inputs are activated.

To do this, we want the sum of both inputs to be greater than the threshold, but each input alone must be lower than the threshold.Let the threshold value be 1 (simple and convenient!).So, the weights are chosen according to the constraints.

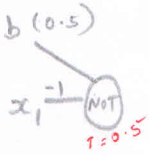
Choosing weights as 0.6 and 0.6? With these weights, individual activation of either input A or B will not exceed the threshold, while the sum of the two will be 1.2, which exceeds the threshold and causes the neuron to activate.

OR implementation



In this case, we want the output to be 1 when either or both of the inputs, A and B, are active, but 0 when both of the inputs are 0. If we make each synapse greater than the threshold, then it'll fire whenever there is any activity in either or both of A and B.

NOT implementation



We have to change a 1 to a 0 - this is easy, just make sure that the input doesn't exceed the threshold. However, we also have to change a 0 to a 1 - how can we do this? The answer is to think of our decision neuron as a tonic neuron - one whose natural state is active. To make this, all we do is set the threshold lower than 0, so even when it receives no input, it still exceeds the threshold. In fact, we have to set the synapse to a negative number (I've used -1) and the threshold to some number between that and 0 (I've used -0.5).

AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to rn = 1 and the AND function to m = n. Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold wo accordingly.

Note:Perceptron can represent all of the primitive boolean (McCulloch-Pitts)functions AND, OR, and NOR. Unfortunately, however, some boolean functionscannot be represented by a single perceptron, such as the XOR functionwhose value is 1 if and only inputs are unequal (Ex: XOR function)

The logic was developed as a model of how our mind works

Note: Difference between Perceptrons and neurons

*Perceptrons come first in 1950s, and it uses a brittle activation function to do classification, so if w*x is greater than some value it predicts positive, otherwise negative.*

Neurons uses a softer activation function by introducing a sigmoid function, a tanh function or other activation functions to pass on values to other neurons in the network.

There are two popular weight update rules.

1) The perceptron rule, and

2)Delta rule

THE PERCEPTRON TRAINING RULE

Learning a perceptron means finding the right values for W(weights). The hypothesis space of a perceptron is the space of all weight vectors. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct output for each of the given training examples.When the training examples are linearly separable perceptron training rule can be applied

The perceptron learning algorithm can be stated as below.

1. Assign random values to the weight vector
2. Apply the weight update ruleto every training example
3. Are all training examples correctly classified?

If yes, Quit else, Go back to Step 2.

The perceptron rule(to learn an acceptable weight vector)

1. Begin with random weights,

2.Then iteratively apply the perceptron to each training example,

3.Modifying the perceptron weights (w_i) whenever it misclassifies an example.

4. Repeat the process, iterating through the training examples as many times as needed untilthe perceptron classifies all training examples correctly.

According to the rule,

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

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

t - target output for the current training example,

o - is the output generatedby the perceptron

η- is a positive constant called the learning rate

Note: Perceptron learning procedure can converge if the training examples are linearly separable and sufficiently small value of η is used.

THE DELTA TRAINING RULE(GRADIENT DESCENT):

*When the training examples are not linearly separable (convergence fails in learning procedures like perceptron learning rule), Delta rule is used. Delta rule converges toward a best-fit approximation to the target concept.

*The key idea behind the delta rule is to use gradient descent to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

Note: Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

* The delta training rule is best understood by considering the task of training an unthresholded perceptron i.e.a linear unit for which the output o is given by

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

* To derive a weight learning rule for linear units, a measure for the training error of a hypothesis is formulated which is given by

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \quad \text{---} \textit{Eq.1}$$

D - set of training examples,
t_d- target output for training example d,
O_d- output of the linear unit for training example d

* In generalGradient descent search determines

i)A weight vector that minimizes E by starting with an arbitrary initial weight vector,

ii) Then repeatedly modifying it in small steps.

iii) At each step, the weight vector is altered in the direction that produces the steepest descent (computing the derivative of E) along the error surface.

iv)This process continues until the global minimum error is reached.

Steepest descent along the error surface 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 as

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \quad \text{-----}\textit{Eq.2}$$

The gradient specifies the direction that produces the steepest increase in E, hence

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta \vec{w} = -\eta \nabla E(\vec{w}) \quad \text{------ } Eq.3$$

η is the learning rate, which indicates step size
Negative sign indicates the movement of the weight vector in the direction that decreases E

*In Component form equation can be written as

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad \text{----- } Eq.4$$

*According to equation 3, iteratively updating weights is done by obtaining partial derivatives of E of equation 1 i.e.

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \quad \text{------ } Eq.\ 5$$

*Substituting equation 5 in equation 4 yields the weight update rule for gradient descent i.e.

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

Gradient- Descent algorithm (summarized)

Gradient- Descent algorithm (summarized)

Each training example is a pair of the form $(\vec{x}, t)$ where where $\vec{x}$ is the vector of input values, andt is the target output value , η is the learning rate

- Initialize each w_i, to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero
  - For each $(\vec{x}, t)$ 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

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

- For each linear unit weight w_i, Do

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

*If η is too large, the gradient descent search runs the riskof overstepping the minimum in the error surface rather than settling into it. Forthis reason, one common modification to the algorithm is to gradually reduce thevalue of η as the number of gradient descent steps grows

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent can be applied whenever

(1) The hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and

(2) The error can be differentiated with respect tothese hypothesis parameters

Drawbacks of gradient descent

The key practical difficulties in applying gradient descent are

(1) Converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and

(2) If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

Incremental gradient descent, or alternatively **stochastic gradientdescent** overcomes the drawbacks of gradient descent algorithm.

Difference between Gradient descent and stochastic descent

Gradient descent	**Stochastic descent**
Computes weight updates after summing over all the training examples	Computes weight updates incrementally for each individual example
Weight update rule $\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$	Weight update rule $\Delta w_i = \eta (t - o)\, x_i$
requires more computation per weight update step	Comparatively requires less computation per weight update step
Handles the situation when there are multiple local minima	In case of multiple local minima, sometimes avoid falling into these local minima

Difference between perceptron rule and delta rule

Perceptron rule	Delta rule
Handles only the linearly separable trainingexamples	Handles both linearly separable and linearly non separable
Converges after a finite numberof iterations to a hypothesis that perfectly classifies the training data	Converges only asymptoticallyin unbounded timetoward the minimum error hypothesis
Updates weights based on the error in the thresholded perceptron output	Updates weights based on the error in the unthresholdedlinear combination of inputs

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

*Single perceptron can only express linear decisionsurfacewhereas multilayer networks learned by the backpropagationalgorithm are capable of expressing a rich variety of nonlinear decisionsurfaces. (refer fig 4.5 on text TM)

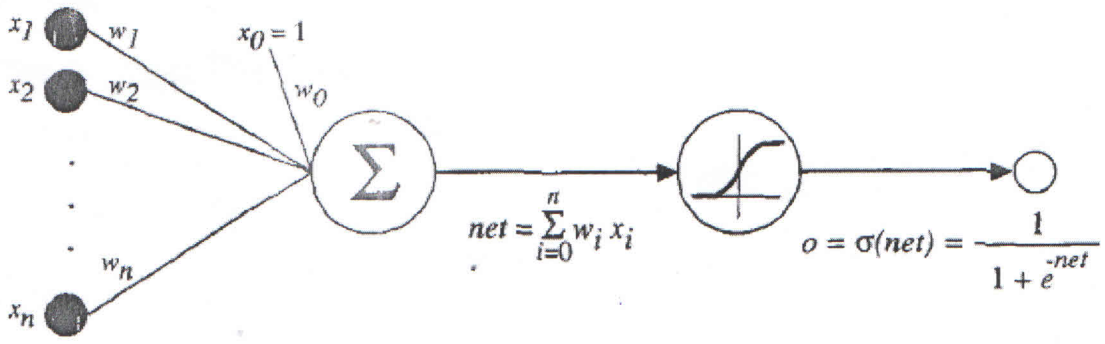
A Differentiable Threshold Unit

In case of delta training rule an unthresholded perceptron is used; that is, a linear unit for which the output o is given by

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

Multiple layers of cascaded linear units produce only linear functions and representing highly nonlinear functions is not met. Choosing perceptron to represent nonlinear function makes it unsuitable because of its undifferentiable nature due to discontinuous thresholds.
The solution is the sigmoid unit-a unit based on a smoothed, differentiable thresholdfunction, producing output which is nonlinear and differentiable function of its input

The sigmoid function is shown in the figure (1) below.
Like the perceptron, the sigmoidunit first computes a linear combination of its inputs,then applies a threshold tothe result.
In the case of the sigmoid unit, the threshold output is acontinuous function of its input(unlike perceptron, which is discontinuous)



the sigmoid unit computes itsoutput o as

$$o = \sigma(\vec{w}.\vec{x}) \qquad \text{where,} \sigma(y) = 1/1+e^{-y}$$

σ = sigmoid function / logistic function/squashing function which ranges between 0 and 1

The BACKPROPAGATION Algorithm

* The backpropagationalgorithm learns the weights for a multilayer network, for a network with a fixed set of units and interconnections.

* It employs gradientdescent to attempt to minimize the squared error between the network outputvalues and the target values for these outputs.

* Multilayer networks considers multiple output units, hence Error E is considered for all the units given b the equation

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

D - set of training examples,
outputs- set of output units in the network
t_{kd} -targetoutput of kth output unit of training example d
o_{kd} -output of kth output unit of training example d

* Challenge faced by back propagation is to search a large hypothesisspace defined by allpossible weight values for all the units in the network

Back propagation algorithm

Back propagation (training_examples, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of $(\vec{x}, \vec{t})$

$\vec{x}$ — input vector for the network

$\vec{t}$ — target vector

η — learning rate

n_{in} — no. of inputs, n_{out} — no. of o/p

n_{hidden} — no. of hidden layer.

→ Create a feed forward network with n_{in}, n_{hidden} & n_{out}

→ Initialize all network weights to small value (-0.5 to 0.5)

→ 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 and compute o_u of every unit u in the network.

Propagate the errors backward through the network.

2. For each network output unit k, calculate the error term δ_k

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

3. For each hidden unit h, calculate its error δ_h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

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

* Multilayer networks can have multiple local minima and the gradient descent algorithm is guaranteed only to converge toward some local minimam & not necessarily global minimum.
  this is the drawback of the back propagation algorithm

* The weight-update loop in backpropagation algorithm may be iterated thousands of times.
  A variety of methods can be chossen for the termination of the iteration viz
  - → halt after a fixed no. of iterations
  - → halt after the error falls below threshold.
  - → halt when validation set meets some criterion.

* The choice of termination is important as too less iterations can fail to reduce error or too many iterations can lead to overfitting the training data.

Adding momentum

BACKPROPAGATION algorithm learns weights for a multilayer network.

The weight update is given by the equation

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \quad \text{——— (1)}$$

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad \text{——— (2)}$$

The variation in eq (2) as shown below will yield the momentum is

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \underbrace{\alpha \Delta w_{ji}(n-1)}_{\text{momentum term}} \quad \text{——— (3)}$$

$0 \leq \alpha < 1$ is called momentum.

Therefore equation 3 with momentum term as in equation 3 is the adding of momentum to backpropagation algorithm.

Note! "Learning in arbitarary acyclic networks" → Self study from Tom Mitchell.

Derivation of the Backpropagation rule :-

The stochastic gradient descent of the Backpropagation algorithm iterates through the training examples descending the gradient of the error E_d.

In this course the weight w_{ji} is updated adding Δw_{ji} i.e.

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{——} (1)$$

$E_d \rightarrow$ Error on the training example d summed over all output units in the network.

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

Applying chain rule

$$net_j = \sum_i w_{ji} x_{ji}$$

(weighted sum of i/ps for unit j)

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

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \quad \text{——} (3)$$

For equation (3), 2 cases are considered

→ Case 1, where unit j is an output unit

→ Case 2, where unit j is an internal unit

Case 1: Training rule for Output Unit Weights.

w_{ji} can influence the network through net_j

net_j can influence the network through o_j.

$$\therefore \frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \quad \text{——} (4)$$

considering the first term of equation (4)

we have $$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2 \quad \text{——} (5)$$

In equation 5,

$\sum_{k} \frac{\partial}{\partial o_j} (t_k - o_k)^2$ will be zero for all output units k except when $k = j$ therefore ignoring the summation, we have,

$$(5) \Rightarrow \frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} (t_j - o_j)^2$$

$$= \frac{1}{2} 2 (t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j}$$

$$= -(t_j - o_j) \quad \text{---------- (6)}$$

Now, considering the second term of equation (4)

w.k.T $o_j = \sigma(net_j)$

$$\therefore \frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

$$= o_j (1 - o_j) \quad \text{---------- (7)}$$

Substituting equations (6) & (7) values in equation (4)

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j) o_j (1 - o_j) \quad \text{---------- (8)}$$

considering equation 8 ~~for~~ & combining (1) & (2)

we have, $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

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

Case 2: Training rule for hidden unit weights

(self study from TOM MITCHELL)

Prof. Ranjitha U.N

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.
Examples : tasks such as learning to control a mobile ,robot, learning to optimize operations in factories, and learning to play board games.

Algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment

Note: Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems

***Introduction**:*
*Reinforcement learning is concerned with how agents(robot for example) can learn successful control policies by experimenting in their environment.

*The goals of the agent can be defined by a reward function that assigns a numerical value to each distinct action the agent may take from each distinct state.

Example – Robert docking to battery for charging
Actions - assigning a positive reward (e.g., +loo) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

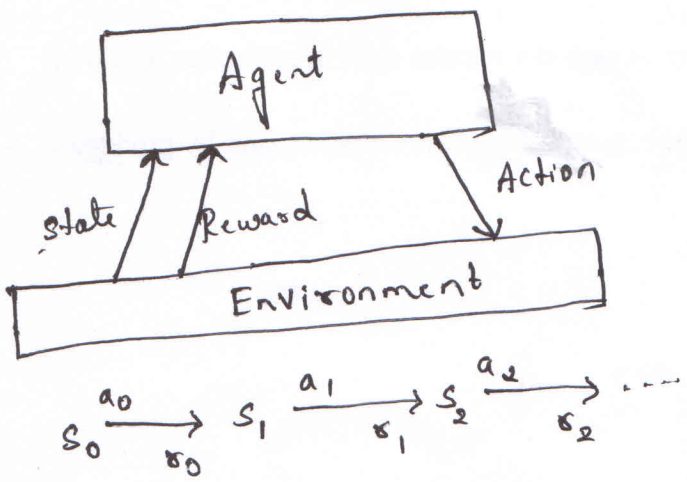
* The control policy (actions taken to achieve goal)is that starts from any initial state, chooses actions that maximize the reward accumulated over time by the agent (shown in below example)

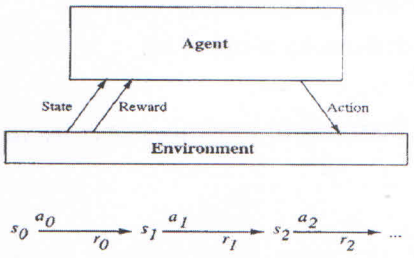
*the problem of learning a control policy to maximize cumulative reward can be applied to examples like
1. manufacturing considering goods produced and cost involved
2. scheduling problems such as choosing which taxis to send for passengers

The goal in the below figure is to maximize

Goal: Learn to choose actions that maximize

$r_0+\gamma r_1+\gamma^2 r_2+\ldots$, where $0 \leq \gamma < 1$





Reinforcement learning problem differs from other function approximation tasks in several important respects
(Features of reinforcement learning)

1.**Delayed reward** : Generally optimal action a = ∏ (s) where, s- given current state , ∏- target function. Each training example would be a pair of the form ***(s, ∏(s)).***
In reinforcement learning, training information is not available in this form , The agent, therefore, faces the problem of temporal credit assignment: determining which of the actions in its sequence are to be credited with producing the eventual rewards

2. **Exploration**: choice of experimentation strategy that produces most effective learning. The learner faces a tradeoff in choosing whether to favor exploration of unknown states and actions (to gather new information), or exploitation of states and actions that it has already learned will yield high reward (to work with gathered information, to yield high reward)

3. **Partially observable states:** many practical situations sensors provide only partial information. In such cases, it may be necessary for the agent to consider its previous observations.

4. **Life-long learning.** raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors

The learning task:
*The formulation of the learning task is based on Markov decision processes.

*In a Markov decision process (MDP) the agent can perceive
*A set ***S*** of distinct states of its environment and
*has a set *A* of actions that it can perform.
*At each discrete time step *t*, the agent senses the current state s_t, chooses a current action a_t and performs it.
*The environment responds by giving the agent a reward $r_t = r(s_t, a)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a)$.

*Here $\delta(s_t, a)$ and $r(s_t, a)$ depend only on the current state and action, and not on earlier states or actions.

*The task of the agent is to learn a ***policy,*** $\prod : S \rightarrow A$,
Precise policy is obtained by possible cumulative reward. Indicated by $V^{\pi}(s_t)$

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

*For the learning task , we need learning policy $\prod$,that maximizes $V^{\pi}(s)$
We will call such a policy an optimal policy and denote it by

$$\pi^* \equiv \underset{\pi}{\text{argmax}}\, V^{\pi}(s), (\forall s)$$

Q LEARNING:

When the available training is not in the form of $\langle s, a \rangle$, learning an optimal policy π^* is difficult. In such situation, it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

$$\pi^*(s) = \underset{a}{\text{argmax}} \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

∴ the agent can acquire the optimal policy by learning V^*, provided it has perfect knowledge of the immediate reward function r and the state transition function δ.

In case where either δ or r is unknown, learning V^* is of no use to find optimal actions.

In such situations evaluation function Q is used

The Q function:

The value of Q is the reward received immediately upon executing action a from state s, plus the value of following the optimal policy thereafter.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \quad \text{—①}$$

$Q(s,a)$ is the quantity that gets maximized w.r.t optimal policy equation.

$$\text{ie } \pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \quad \text{—②}$$

Eq ② implies that optimal policy can be found, even without the knowledge of r and δ. which inturn implies that the agent can choose the optimal action without ever conducting a search to consider what state results from the action.

[Refer fig 13.2 of Tom Mitchell.]

An algorithm for learning Q

* Learning the Q function corresponds to learning the optimal policy.
* Given only a sequence of immediate rewards r, need to estimate training values for Q.
  which is accomplished through iterative approximation.
* The ~~relation~~ relationship between Q & V^* (max cummulative reward)
  is given by $V^*(s) = \max_{a'} Q(s,a') \quad \text{—①}$

But w.k.T

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a)) \quad -(2)$$

Rewriting (2) by making use of (1), we have

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a') \quad -(3)$$

* In the description of the algorithm

$\hat{Q}$ → learner's estimate of actual Q function, where learner represents $\hat{Q}$ by a large table with a separate entry for each state action pair.

The agent repeatedly observes its current state s, chooses some action a, executes this action then observes the resulting reward $= r(s,a)$ and the new state $s' = \delta(s,a)$

Then updates the table entry for $\hat{Q}(s,a)$ following each such transition according to the rule:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

Q algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to '0'

Observe the current state s

Do forever:

→ Select an action a and execute it

→ Receive immediate reward r

→ Observe the new state s'

→ Update the table entry for $\hat{Q}(s,a)$ as follows.

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

Section 13.3.3, 13.3.4, 13.3.5, 13.3.6 } → Self study

(TOM MITCHELL)