# Learning

Course: CS40022
Instructor: Dr. Pallab Dasgupta

Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

# Paradigms of learning

- Supervised Learning
  - Both inputs and outputs are given
  - The outputs are typically provided by a friendly *teacher.*
- Reinforcement Learning
  - The agent receives some evaluation of its actions (such as a fine for stealing bananas), but is not told the correct action (such as how to buy bananas).

# Paradigms of learning

- Unsupervised Learning
  - The agent can learn relationships among its percepts, and the trend with time

# Decision Trees

- A decision tree takes as input an object or situation described by a set of properties, and outputs a yes/no “decision”.

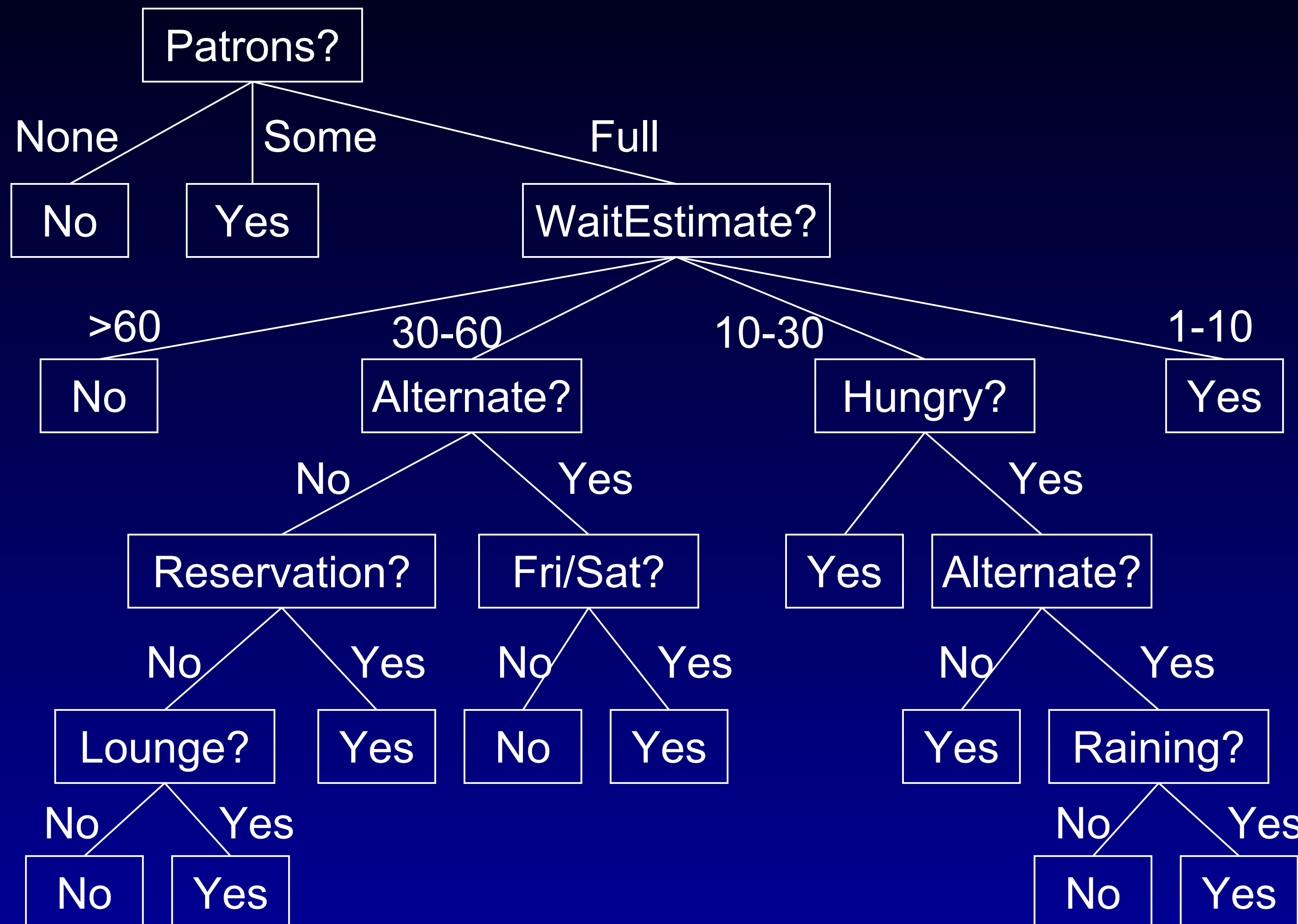
Decision: *Whether to wait for a table at a restaurant.*

1. *Alternate*: whether there is a suitable alternative restaurant
2. *Lounge*: whether the restaurant has a lounge for waiting customers
3. *Fri/Sat*: true on Fridays and Saturdays

# Decision Trees

4. *Hungry*: whether we are hungry
5. *Patrons*: how many people are in it (None, Some, Full)
6. *Price*: the restaurant's rating (★, ★★, ★★★)
7. *Raining*: whether it is raining outside
8. *Reservation*: whether we made a reservation
9. *Type*: the kind of restaurant (Indian, Chinese, Thai, Fastfood)
10. *WaitEstimate*: 0-10 mins, 10-30, 30-60, >60.

# Sample Decision Tree



# Decision Tree Learning Algorithm

**Function** ***DTreeL***( *samples, attributes, default* )

  if *samples* is empty then return *default*

  else if all *samples* have the same classification then

    return the classification

  else if *attributes* is empty then

    return Majority-Value( *samples* )

  else

    *best* ← ChooseAttrib( *attributes, samples*)

    *tree* ← a new dtree with root test *best*

# DTreeL Algorithm (contd..)

for each value $v_i$ of *best* do

*samples*$_i$ ← { members of *samples* with *best* = $v_i$ }

*subtree* ← DTreeL( *samples*$_i$, *attributes* – *best*, Majority-Value( *samples* ))

add a branch to *tree* with label $v_i$ and subtree *subtree*

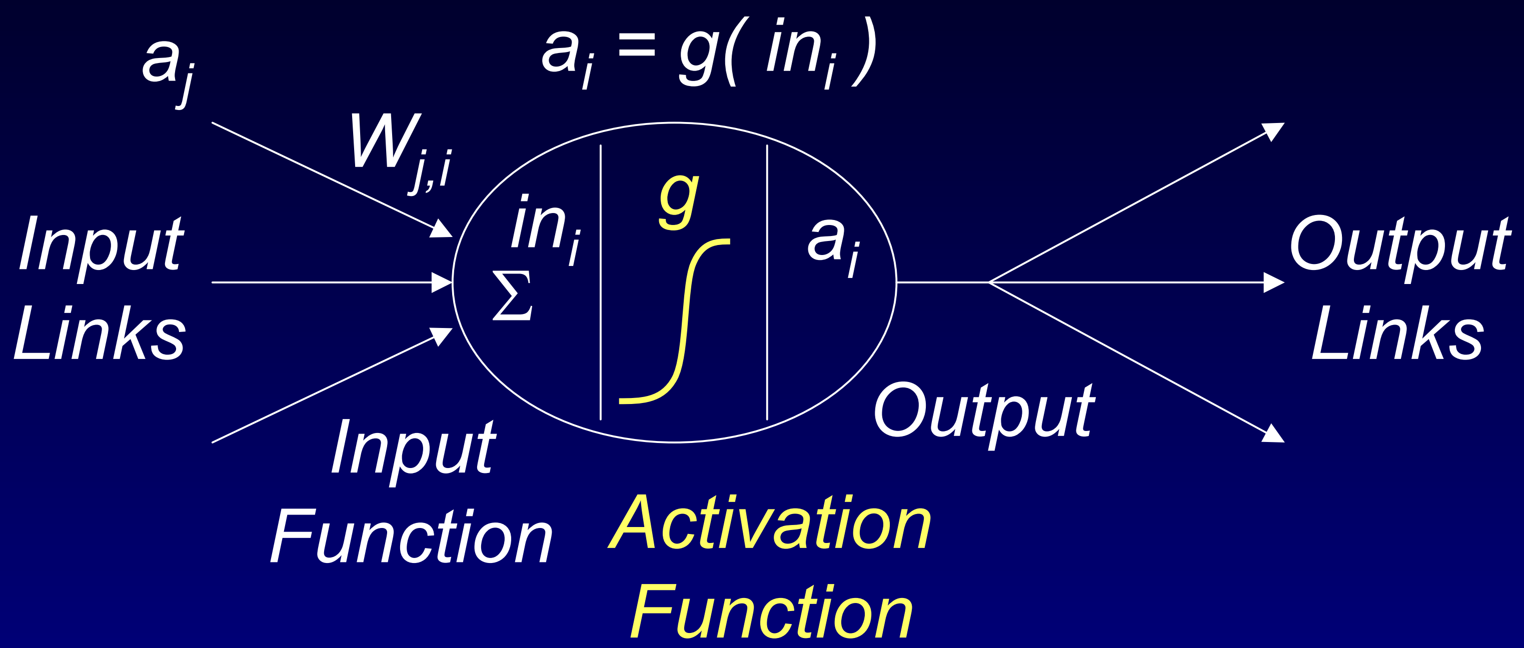
end

return *tree*

# Neural Networks

- A neural network consists of a set of nodes (neurons/units) connected by links
  - Each link has a numeric weight
- Each unit has:
  - a set of input links from other units,
  - a set of output links to other units,
  - a current activation level, and
  - an activation function to compute the activation level in the next time step.

# Basic definitions



# Basic definitions

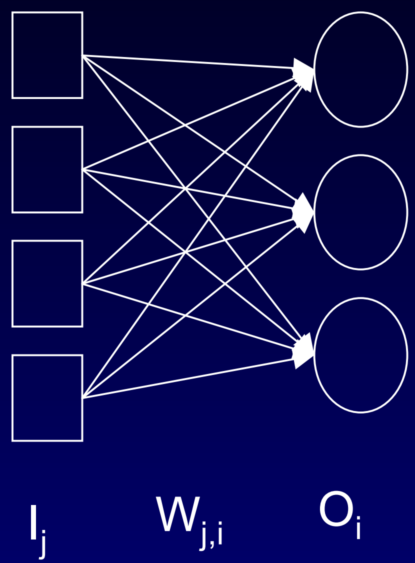
- The total weighted input is the sum of the input activations times their respective weights:

$$in_i = \sum_j W_{j,i} a_j$$

- In each step, we compute:

$$a_i \leftarrow g(in_i) = g\left( \sum_j W_{j,i} a_j \right)$$

# Learning in Single Layer Networks




- If the output for a output unit is O, and the correct output should be T, then the error is given by:

  $$Err = T - O$$

- The weight adjustment rule is:

  $$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

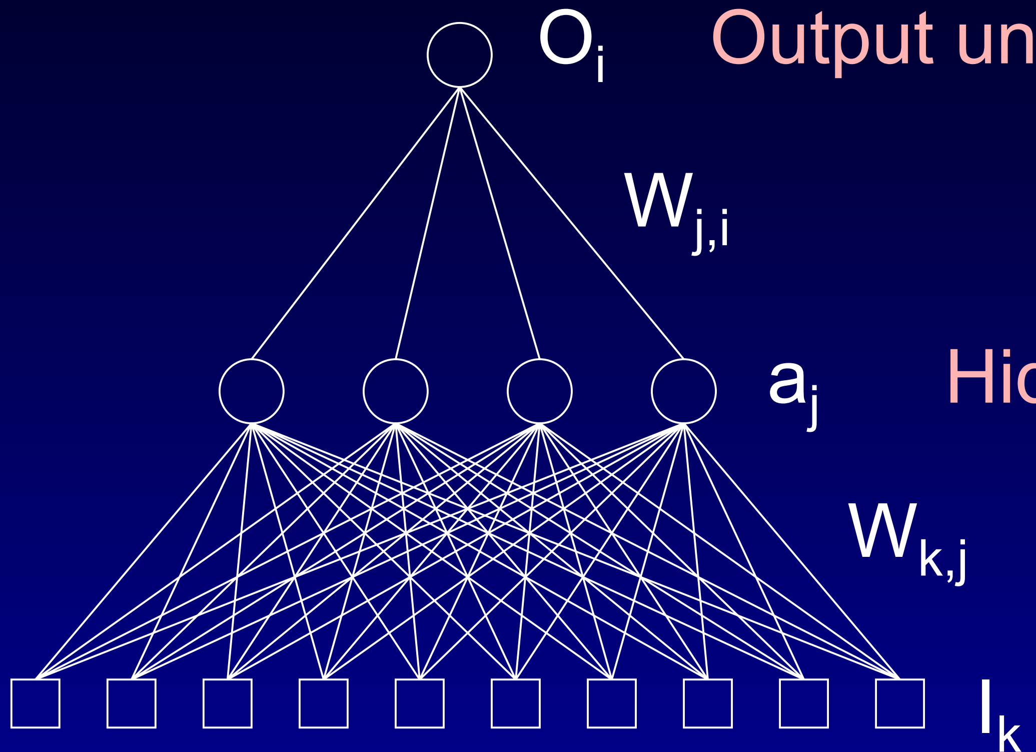
  where $\alpha$ is a constant called the learning rate

# Learning in Single Layer Networks

- This method is unable to learn all types of functions
  - can learn only linearly separable functions

- Used in competitive learning

# Two-layer Feed-Forward Network



# Back-Propagation Learning

- $Err_i = (T_i - O_i)$ is the error at the output node
- The weight update rule for the link from unit j to output unit i is:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(in_i)$$

where g' is the derivative of the activation function g.

# Back-Propagation Learning

- Let $\Delta_i$ = $Err_i$ $g'(in_i)$. Then we have:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

# Error Back-Propagation

- For updating the connections between the inputs and the hidden units, we need to define a quantity analogous to the error term for the output nodes
    - Hidden node j is responsible for some fraction of the error $\Delta_i$ each of the output nodes to which it connects

# Error Back-Propagation

- So, the $\Delta_i$ values are divided according to the strength of the connection between the hidden node and the output node, and propagated back to provide the $\Delta_j$ values for the hidden layer

$$\Delta_j = g'(in_j)\sum_i W_{j,i}\Delta_i$$

# Error Back-Propagation

- The weight update rule for weights between the inputs and the hidden layer is:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j$$

# The theory

- The total error is given by:

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2 \quad = \quad \frac{1}{2}\sum_i \left( T_i - g\left( \sum_j W_{j,i} a_j \right)\right)^2$$

$$= \frac{1}{2}\sum_i \left( T_i - g\left( \sum_j W_{j,i} \left( \sum_k W_{k,j} I_k \right)\right)\right)^2$$

# The theory

- Only one of the terms in the summation over i and j depends on a particular $W_{j,i}$, so all other terms are treated as constants with respect to $W_{j,i}$

$$\frac{\partial E}{\partial W_{j,i}} = -a_j (T_i - O_i) g'\left(\sum_j W_{j,i} a_j\right) = -a_j (T_i - O_i) g'(in_i) = -a_j \Delta_i$$

Similarly:
$$\frac{\partial E}{\partial W_{k,j}} = -I_k \Delta_j$$