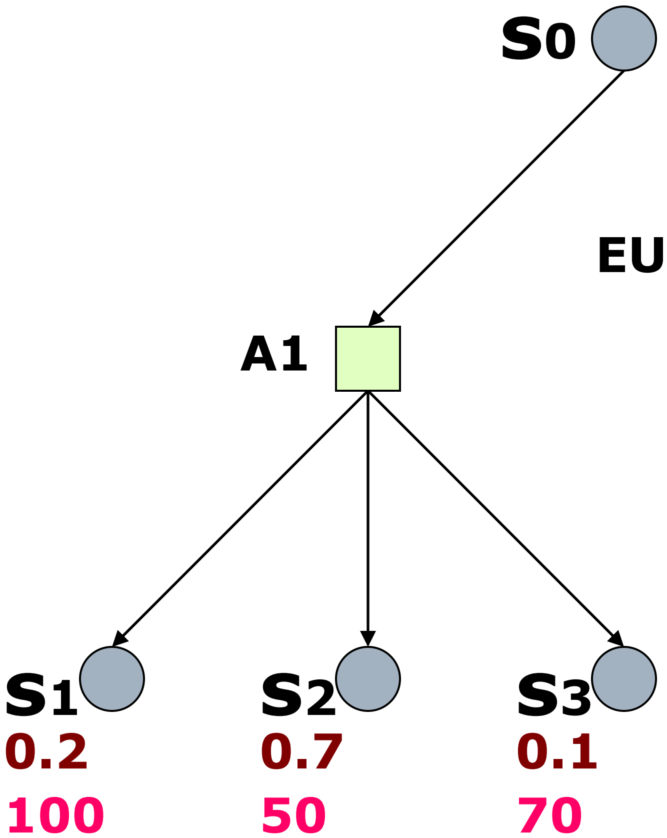
DECISION NETWORKS (CONT)
NEURAL NETWORKS

Expected Utility

- Random variable X with n values $x_1,\ldots,x_n$ and distribution $(p_1,\ldots,p_n)$
  E.g.: X is the state reached after doing an action A under uncertainty
- Function U of X
  E.g., U is the utility of a state
- The expected utility of A is

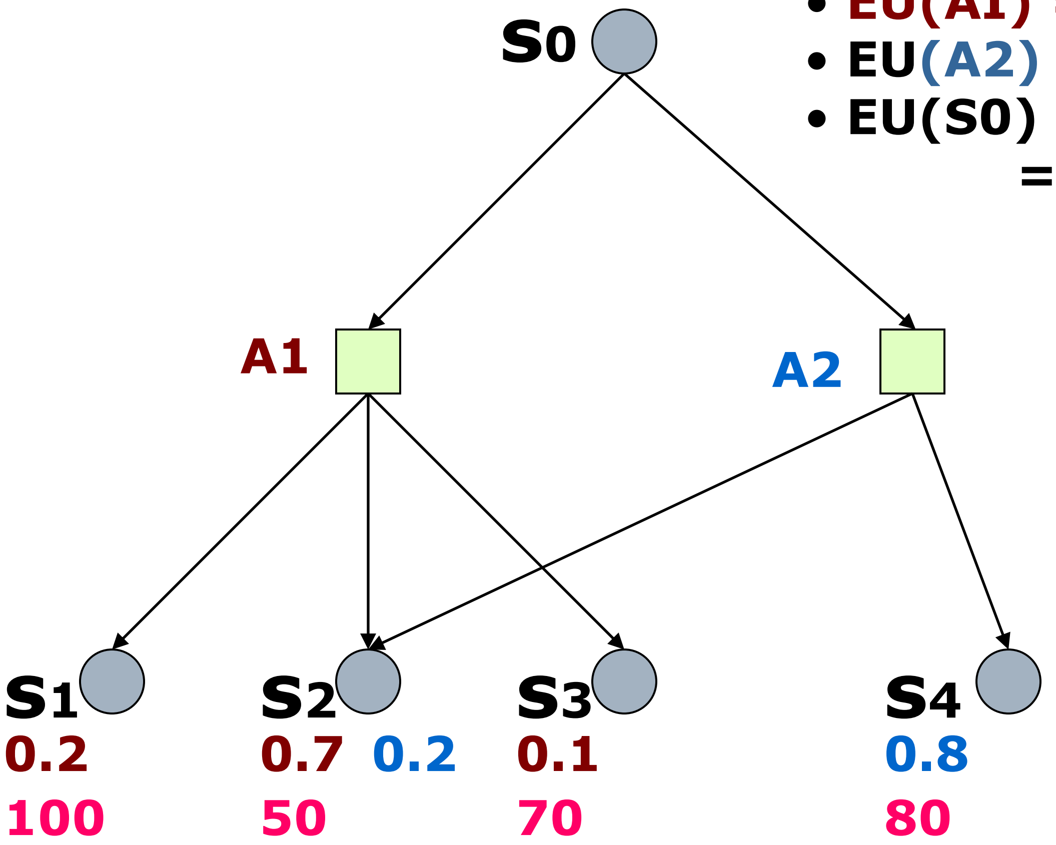
$$EU[A] = \sum_{i=1,\ldots,n} p(x_i|A)U(x_i)$$

One State/One Action Example



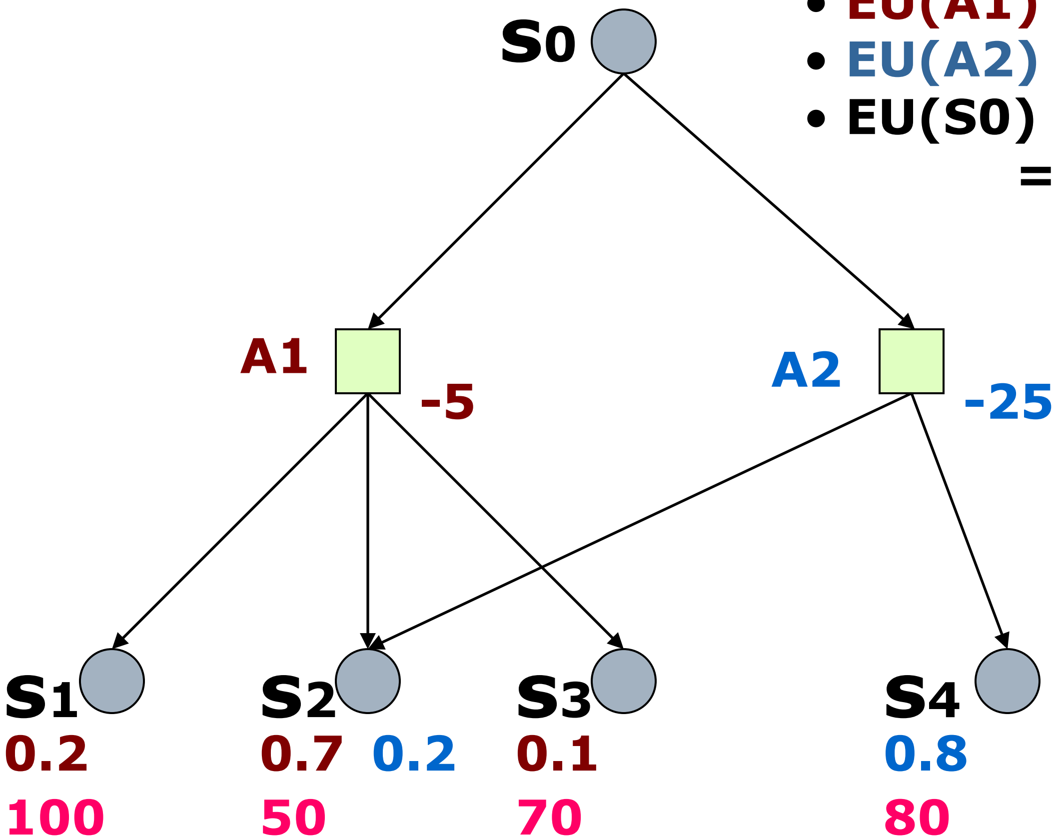
$$
\begin{aligned}
EU(A1) &= 100 \times 0.2 + 50 \times 0.7 + 70 \times 0.1 \\
&= 20 + 35 + 7 \\
&= 62
\end{aligned}
$$

One State/Two Actions Example



- EU(AI) = 62
- EU(A2) = 74
- EU(S0) = max{EU(A1),EU(A2)}
  = 74

Introducing Action Costs



- EU(A1) = 62 – 5 = 57
- EU(A2) = 74 – 25 = 49
- EU(S0) = max{EU(A1),EU(A2)}
  = 57

MEU Principle

- rational agent should choose the action that maximizes agent's expected utility
- this is the basis of the field of decision theory
- normative criterion for rational choice of action

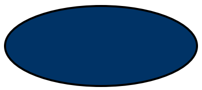
We looked at

- Decision Theoretic Planning
  - Simple decision making (ch. 16)
  - Sequential decision making (ch. 17)

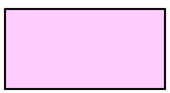
Decision Networks

- Extend BNs to handle actions and utilities
- Also called Influence diagrams
- Make use of BN inference
- Can do Value of Information calculations

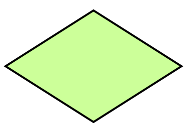
Decision Networks cont.



- Chance nodes: random variables, as in BNs



- Decision nodes: actions that decision maker can take



- Utility/value nodes: the utility of the outcome state.

Making a Rational Decision

- At a decision node
- Given a combination of values of evidence variables, and each possible action given this evidence
  - Compute the EU of each action you can decide to do
  - Decide to do the action with the maximum EU
- Policy: choice of action (not necessarily the best) for each possible combination of values of evidence variables

Policy

Policy is a mapping from states to actions.
Given a policy, one may calculate the expected utility from series of actions produced by policy.
The goal: Find an **optimal policy**, one that would produce maximal expected utility.

- Decision node D_i
  - Can take values in domain dom(D_i)
  - Has set of parents P_i that take values in domain dom(P_i)
- Policy π is a set of mappings π_i of dom(P_i) to dom(D_i)
  - π_i associates a decision to each state the parents of D_i can be in
  - π associates a series of decisions to each state the network can be in

Value of a Policy

- Expected utility if decisions are taken according to the policy
- $EU(\pi) = \sum_x P(x)\, U(x,\pi(x))$
  - $EU(\pi_{bp}) = \sum_{\$,s} P(\$,S)\, U(\$,S,\pi_{bp}(\$,S))$
- Optimal policy π^* is the one with the highest expected utility
  - $EU(\pi^*) \geq EU(\pi)$ for all policies π

Value of Information (VOI)

- suppose agent's current knowledge is E. The value of the current best action α is

$$EU(\alpha \mid E) = \max_A \sum_i U(Result_i(A))P(Result_i(A) \mid E, Do(A))$$

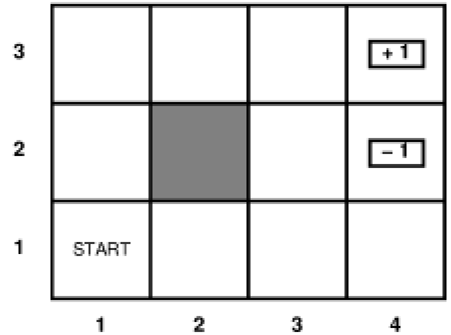
- the value of the new best action (after new evidence E′ is obtained):

$$EU(\alpha' \mid E, E') = \max_A \sum_i U(Result_i(A))P(Result_i(A) \mid E, E', Do(A))$$

- the value of information for E′ is:

$$VOI(E') = \sum_k P(e_k \mid E)EU(\alpha_{ek} \mid e_k, E) - EU(\alpha \mid E)$$

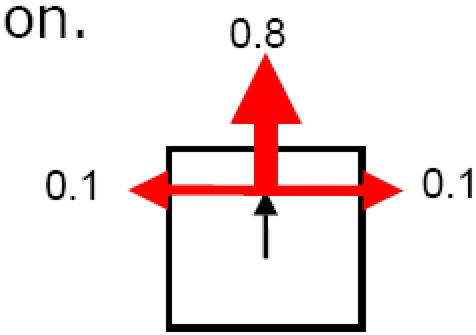
Sequential Decision Problems (1)



- Beginning in the start state the agent must choose an action at each time step.
- The interaction with the environment terminates if the agent reaches one of the goal states (4, 3) (reward of +1) or (4,2) (reward –1). Each other location has a reward of -.04.
- In each location the available actions are Up, Down, Left, Right.

Sequential Decision Problems (2)

- Deterministic version: All actions always lead to the next square in the selected direction, except that moving into a wall results in no change in position.
- Stochastic version: Each action achieves the intended effect with probability 0.8, but the rest of the time, the agent moves at right angles to the intended direction.



Markov Decision Problem (MDP)

Given a set of states in an accessible, stochastic environment, an MDP is defined by

- Initial state S_0
- Transition Model T(s,a,s')
- Reward function R(s)

Transition model: T(s,a,s') is the probability that state s' is reached, if action a is executed in state s.

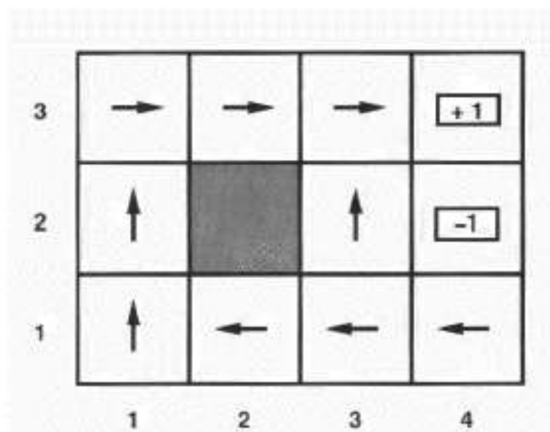
Policy: Complete mapping π that specifies for each state s which action $\pi(s)$ to take.

Wanted: The optimal policy π^* that maximizes the expected utility.

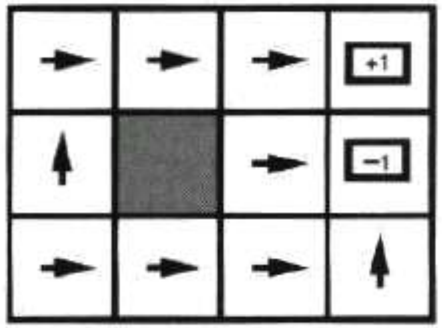
Optimal Policies (1)

- Given the optimal policy, the agent uses its current percept that tells it its current state.
- It then executes the action $\pi^*(s)$.
- We obtain a simple reflex agent that is computed from the information used for a utility-based agent.

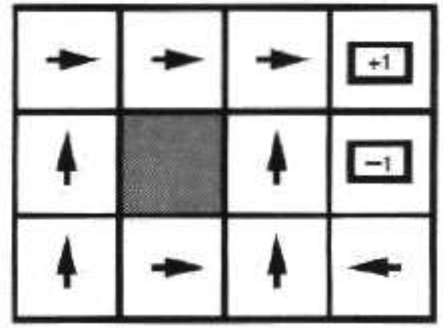
Optimal policy for our MDP:



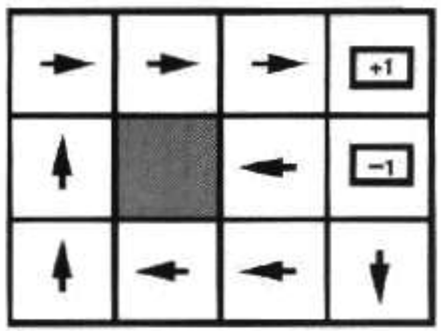
Optimal Policies (2)



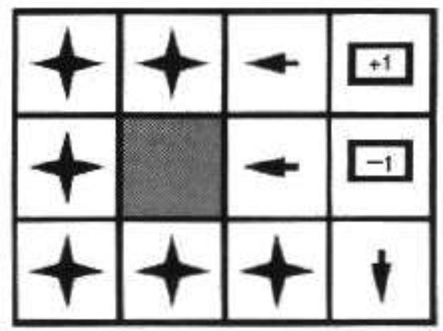
$R(s) \leq -1.6248$



$-0.4278 < R(s) < -0.085$



$-0.0221 < R(s) < 0$



$0 < R(s)$

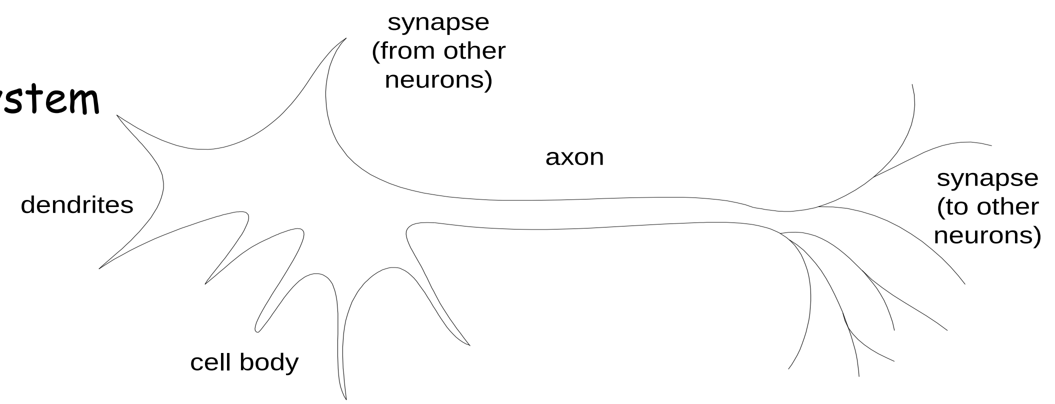
How to compute optimal policies?

NEURAL NETWORKS

Neural Networks

- A technique to mimic the human brain (how it learns, retains info, etc).

- **Neuron**
  - Major cells of the nervous system
  - Simple computing element.



• Dendrites receive impulses from other neurons via synapses.

• Synapses can be excitatory or inhibitory, and respectively raise or lower the electric potential of the target neuron.
•Electrical potential = ion concentration inside the nucleus vs. outside.

Once electric potential exceeds a threshold, the cell "fires" and sends a signal (action potential) down the axon to synapses and from there to other neurons.

Why Try To Mimic The Brain?

- Brain is highly parallel and can learn by changing/adjusting the connections between neurons.
- Failure of a single neuron is unlikely to affect overall operation (robust).
- Information is encoded in:
  - <u>Connection pattern</u> of neurons,
  - <u>Amplification</u> of signals in dendrites,
  - <u>Activation function</u> and <u>threshold values</u> controlling the firing of a cell.

Why Try To Mimic The Brain?

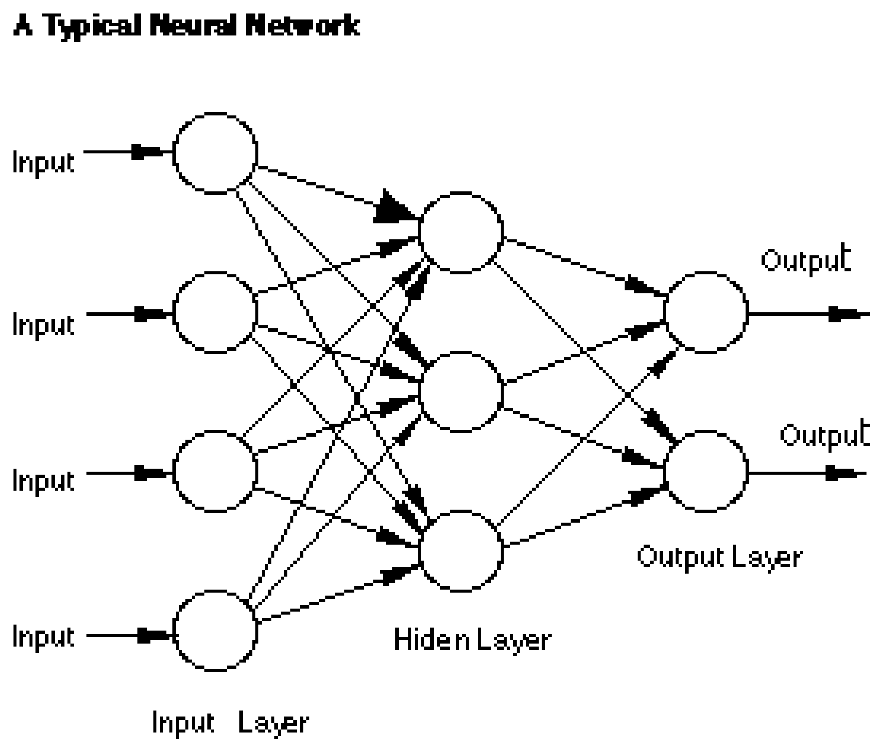
- A neuron generates a response (action potential) when its inputs raise the electrical potential above a threshold.
- Since the inputs are amplified (<u>weighted</u>) by dendrites, this is equivalent to detecting an <u>input pattern</u>.
- More neurons and connections between neurons means more complex patterns can be learned and detected.

Artificial Neural Networks

- An "artificial neural network" consists of a large, densely inter-connected arrangements of simple computing elements
- Each computing element is designed to mimic a neuron.
- There are different types/categories of Neural Networks.
- The type/category depends on:
  - Topology of computing elements;
  - Characteristics of computing elements;
  - How the network learns.
- Different networks are appropriate for different applications.

Illustration of an Artificial Neural Network

- Example of a multi-layer, feed-forward neural network:



Types of Neural Nets

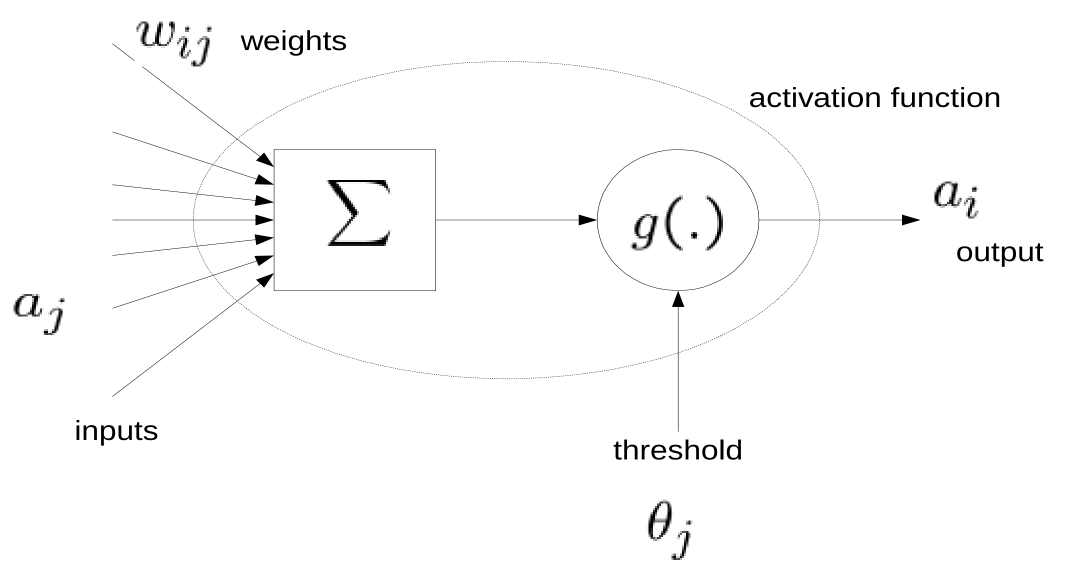
CLASS	EXAMPLES	CHARACTERISTICS
Feed-forward NN (no feedback)	Single-layer perceptron (SLP)	❑ Simplest NN ❑ Uses supervised learning. ❑ Can handle only linearly separable problems.
	Multi-layer perceptron (MLP)	☐ Can handle non-linear problems. ☐ Sometimes converge to local minima. ☐ Prone to overfitting.
	Radial Basis Function (RBF) Networks	☐ Alternative to MLP. ☐ Do not converge to local minima. ☐ Prone to overfitting.
	Self-organizing maps (SOM)	☐ Uses an unsupervised learning method. ☐ Good for producing visualizations of data (groups similar objects).

Types of Neural Nets

CLASS	EXAMPLES	CHARACTERISTICS
Recurrent NN (involves feedback)	Hopfield NN	□ Fully-connected network. □ Knowledge is encoded, not learned. □ Uses unsupervised learning. □ Like associative memory.
Stochastic NN (involves randomness)	Boltzmann Machine	□ Seen as the stochastic counterpart of Hopfield. □ Based on simulated annealing. □ Practical when connectivity is constrained (not fully-connected).
Modular NN (collection of smaller networks)	Committee of Machines (CoM)	□ Collection of networks that vote on a particular example. □ Gives better result than individual networks.

Simple Computing Element (1 of 2)

- Referred to as a neuron (since that's what we mimic).



- Note similarity with a physical neuron (corresponding structure in parentheses):
  - Inputs (synapses) are weighted (dendrites) and summed (nucleus).
  - A threshold, θ, and activation function, g, transform the sum (electric potential) into an output (action potential).

Simple Computing Element (2 of 2)

- We see two computations occurring inside a neuron:

  1. Linear weighted summation of incoming values.

  2. Non-linear activation function for calculation of output.

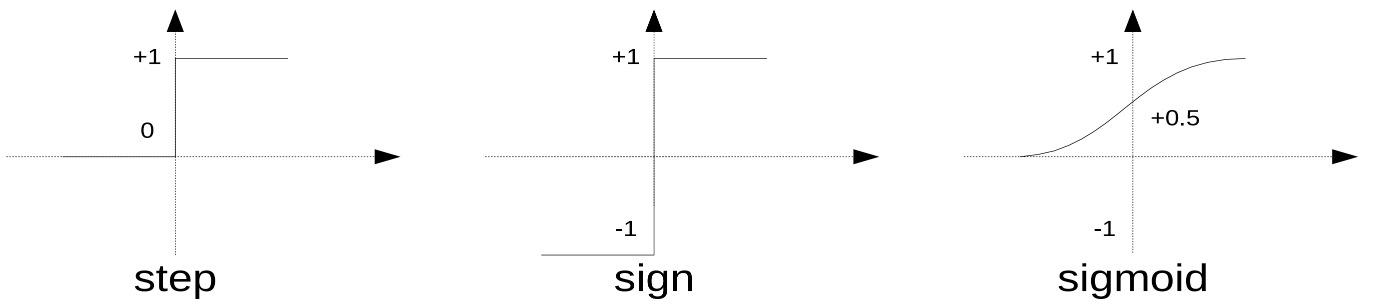
$$a_i = g_i(\sum_j w_{ij} a_j - \theta_i)$$

- The threshold θ_i can be replaced by an extra input fixed to -1 whose weight is then the threshold value
  - This is just an alternative arrangement to account for the threshold.

Activation Functions

- Common activation functions are the step/sign function and sigmoid function (shown below).

- Functions map the sum (after thresholding) to the range [0,1] or [-1,1]
  - We want the neuron active (output close to 1) when suitable inputs are present, and inactive (close to 0) when wrong inputs are present.
  - May alternatively want to penalize the wrong inputs (-1 output).



$$sig(x) = \frac{1}{1 + \exp(-x)}$$

Sigmoid Activation Function

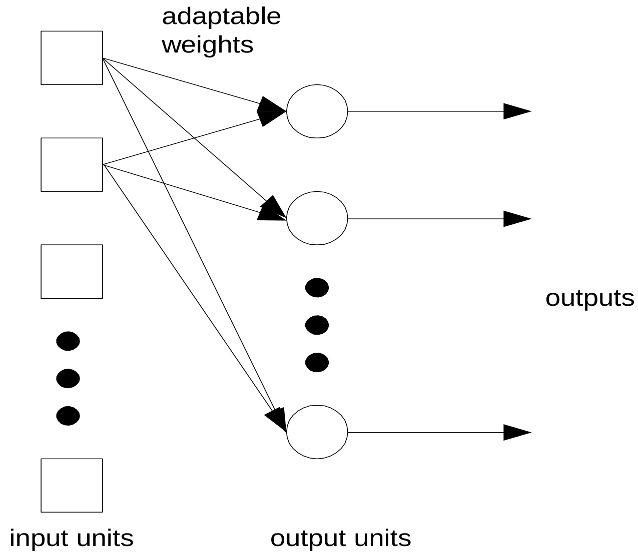
- The sigmoid activation function is nice because it is differentiable.
- Also, the derivative is simple to compute:

$$\begin{aligned}
\frac{d}{dx} sig(x) &= \frac{d}{dx}(1+\exp(-x))^{-1} \\
&= exp(-x)(1+\exp(-x))^{-2} \\
&= (1+\exp(-x))^{-1}\left(\frac{exp(-x)+1-1}{1+\exp(-x)}\right) \\
&= sig(x)(1-sig(x))
\end{aligned}$$

- So, its derivative can be computed from the function itself; this is convenient when training back-propagation networks (more later).

Simple Perceptron Network

- A simple network with one layer of neurons connected to a number of inputs is called a perceptron:



- Example of a “feed-forward” network. Inputs applied, work their way through the network via weighting and activation functions, result in outputs.

Supervised Learning

- Simple perceptron is a network which uses **supervised learning**.
  - For a given input $\mathbf{I}^{\mu}$ we know the desired output ζ^{μ}.
  - We compute the output $\mathbf{O}^{\mu}$ using the weights and activation function.
  - We compare $\mathbf{O}^{\mu}$ with ζ^{μ} and **adjust weights and thresholds** to **minimize** the **differences** (error) in the desired output vs. actual output.

Simple Perceptron Network Capabilities

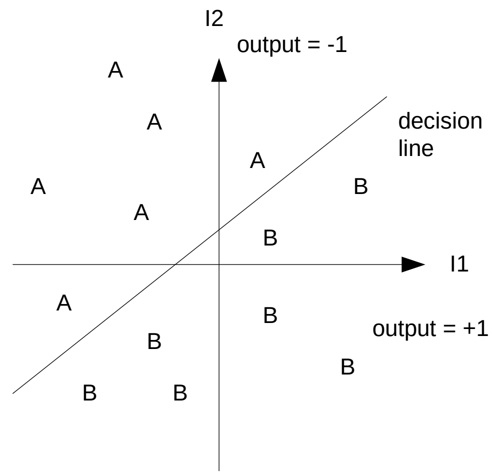
- **Question:** What can a simple perceptron network do?
- Consider a single output and assume the activation function is the sign fn.

$$O_i = \mathrm{sign}(\sum_j w_{ij} I_j)$$

- If sum ¸ 0, then output is +1. If sum < 0, then output is -1.
- **The output divides the input into 2 categories (+1 and -1) by drawing a "decision boundary" (a hyper-plane).**

Visualization

- Consider a two-input problem space, and apply a simple perceptron:



- Output switches between +1 and -1 occur when input summation is zero:

$$\sum_j w_{ij} I_j = w_{i0}(-1) + w_{i1} I_1 + w_{i2} I_2 = 0 \qquad I_2 = -\frac{w_{i1}}{w_{i2}} I_1 - \frac{w_{i0}}{w_{i2}}$$

- So training (learning) is accomplished by **modifying the weights (and threshold)** of the network:
  - Changing the weights alters the **decision line**, and thereby facilitates getting the desired output for a given input.

Training Algorithm (Rosenblatt, 1959)

- Steps:

  1. Initialize weights and thresholds to small random numbers.

  2. Present a new input $\mathbf{I}^{\mu}$ and desired output ζ^{μ} to the network.

  3. Calculate the current (actual) output $\mathbf{O}^{\mu}$ from the network.

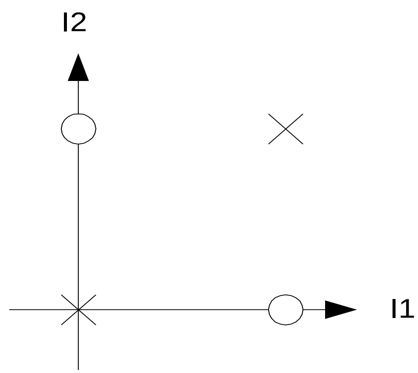
  4. Adapt weights $\mathbf{w}_i$ Ã $\mathbf{w}_i + \eta\ (\ \zeta^{\mu} - \mathbf{O}^{\mu})\ \mathbf{I}^{\mu}$ where η **2 (0,1)** is the **learning rate**.

  5. Repeat for many desired input-output pairs until no error.

- NOTE: weights will cease to change once output becomes correct since $(\ \zeta^{\mu} - \mathbf{O}^{\mu})$ will be zero.

The XOR Problem

- Single layer networks have a big problem! What happens if no decision boundary?
- Consider XOR problem:

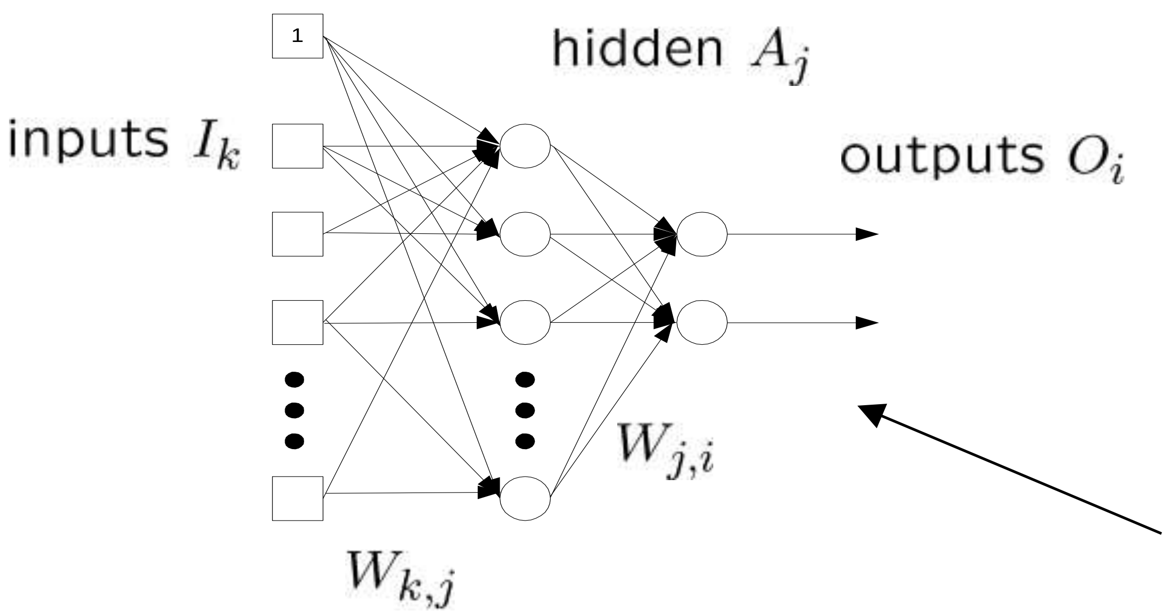


I1	I2	O
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

- NO SINGLE decision line that separates the two categories. Problem is not **linearly separable** which is required for single layer networks.
- So,… how good can a neural network be if it can't even solve the XOR problem?

Multi-Layer Neural Networks (1 of 2)

- Overcomes the limitation of simple single layer networks; consists of several layers of **hidden nodes** between input and output nodes.
- Originally ignored since no training methods had been developed.



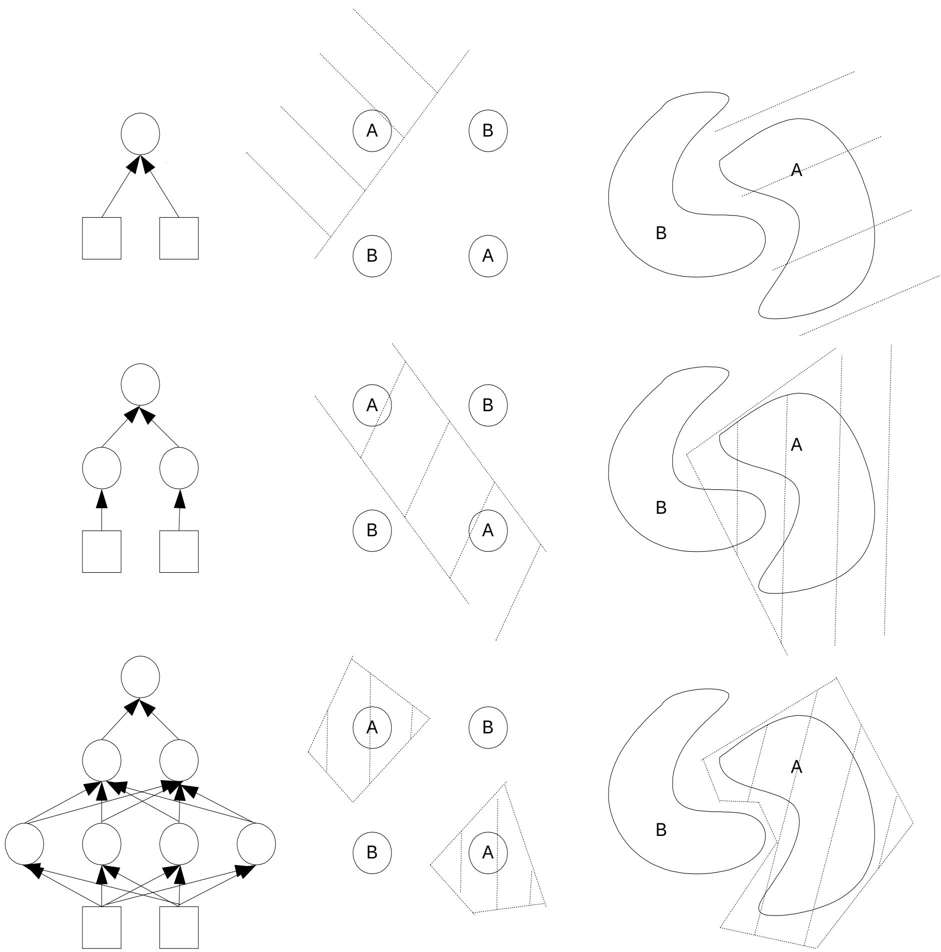
Example; 1 layer of hidden nodes

Issues in Multi-Layer Networks

- How many layers of hidden nodes are needed?
- How many nodes in each hidden layer?
- How do we do training?

Hidden Layers and Geometry

- Hidden layers determine the geometric regions which can be categorized:



- Regions shown with step or sign activation function.
- Sigmoid would smooth out the sharp corners.

Training Multi-Layer Networks (Back Propagation)

- Training is easiest with **sigmoid function** so we will consider that; need to be able to take a **derivative**.
- **Back-propagation** is similar to training a simple perceptron network:
  - Present input and compute resulting output.
  - If resulting output and desired output agree, do nothing. Otherwise adjust weights to reduce the error. (What do we call this type of learning?)
- The novelty of back-propagation is that weights are updated by assessing and **dividing** the **error** in the output **among all the weights** in the network.
  - Why is this important?
- **We can derive a means of reducing output error and use the sigmoid function which has a derivative.**
  - Essentially **gradient descent**. (How will this affect learning?)

Back-Propagation Derived (1 of 5)

- Use the sigmoid function:

$$sig(x) = \frac{1}{1 + \exp(-x)}$$

- Assume we present input I, get actual output O and our target output is T.
- Measure the error in the output as:

$$E = \frac{1}{2} \sum_i (T_i - O_i)^2$$

- Notice that although E is written as a function if T and O, it is really a function of the weights (and thresholds) in the network.
- **To minimize the error E, we take its partial derivative wrt. each weight in the network; this becomes a descent direction for minimization.**
- **The rest is just math…**

Back Propagation Derived (2 of 5)

- The output of each hidden node is:

$$A_j = sig(\sum_k W_{k,j} I_k)$$

- The output of each output node is:

$$O_i = sig(\sum_j W_{j,i} A_j)$$

Back Propagation Derived (3 of 5)

- Take partial derivative of E wrt. weights from hidden nodes to output nodes:

$$
\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(T_i - O_i)\frac{\partial}{\partial W_{j,i}} O_i \\
&= -(T_i - O_i)\frac{\partial}{\partial W_{j,i}} sig\left(\sum_j W_{j,i} A_j\right) \\
&= -(T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) A_j \\
&= -A_j \Delta_i
\end{aligned}
$$

- Where g(.) is the derivative of the sigmoid function and:

$$\Delta_i = (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right)$$

Back Propagation Derived (4 of 5)

- Take partial derivative of E wrt. weights from input nodes to hidden nodes:

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (T_i - O_i) \frac{\partial}{\partial W_{k,j}} O_i \\
&= -\sum_i (T_i - O_i) \frac{\partial}{\partial W_{k,j}} sig\left(\sum_j W_{j,i} A_j\right) \\
&= -\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i} \frac{\partial}{\partial W_{k,j}} A_j \\
&= -\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i} \frac{\partial}{\partial W_{k,j}} sig\left(\sum_k W_{k,i} I_k\right) \\
&= -\left[\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i}\right] g\left(\sum_k W_{k,j} I_k\right) I_k \\
&= -\left[\left(\sum_i W_{j,i} \Delta_i\right) g\left(\sum_k W_{k,i} I_k\right)\right] I_k \\
&= -\Delta_j I_k
\end{aligned}
$$

- Where g(.) is the derivative of the sigmoid function and:

$$\Delta_j = g(\textstyle\sum_k W_{k,i} I_k) \sum_i W_{j,i} \Delta_i.$$

Back-Propagation Derived (5 of 5)

- The previously derived partial derivatives give a descent direction.

The network is trained in the following manner:

- Initial weights are determined randomly.
- An input is applied to the network and output is determined.
- Weights from hidden to output nodes are updated:

$$W_{j,i} = W_{j,i} + \alpha \times A_j \times \Delta_i$$

- Weights from input to hidden nodes are updated:

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

  - Here, α **2 (0,1)** is the **learning rate.**

Additional Details On Back-Propagation Training

- The input-output pairs used for training are the **training set**.
- Idea is to train (update weights) using a **good**, **trusted** and **representative** sample of inputs.
  - When new (unknown) inputs are applied that were not used in training, the network should/will give the correct input.
  - This is because similar inputs share similar characteristics.
- **Terminology**: Given a set of I/O pairs, presenting each I/O pair to the network and updating weights is referred to as **one epoch** of training.
- **Training sets** should be applied to the network multiple times until the network weights stop changing.; i.e., we should do **multiple epochs** and the network becomes "better" after each epoch.

Problems With Back-Propagation Training

- This training method does have some problems.
  - **Efficiency** – slow to learn and need to do many epochs of training to get good network performance.
  - **Locality** – training based on gradient descent which means we might converge to a local minimum of E which will give bad answers for inputs not in the training set.

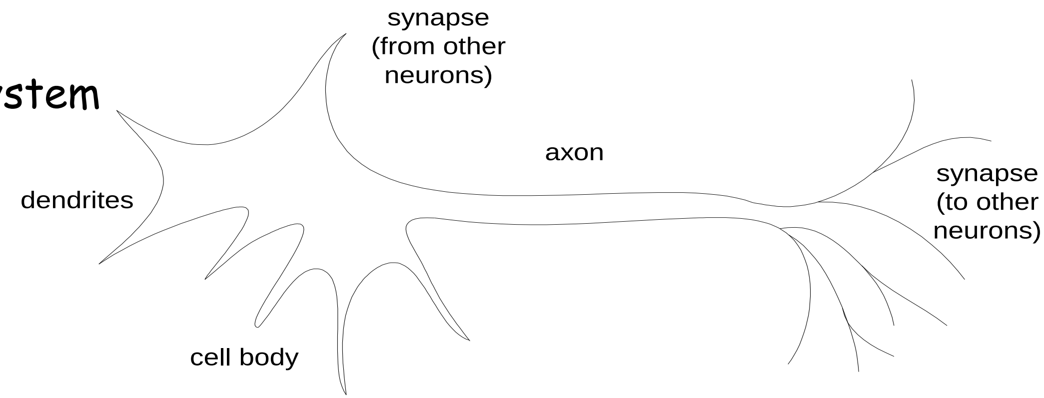
Example Applications

- NETtalk (1987):
  - Network trained to pronounce English text.
  - Sliding window of text; 7 characters in stream presented to network.
  - 80 hidden nodes, 26 output nodes (encoded phonemes).
  - Trained on 1024 words from side-by-side English/phoneme source.
  - 10 training epochs gave intelligible speech; 50 epochs gave 95% accuracy.
  - Performance on actual data as 78%.
- Character Recognition (1989):
  - Read zip codes on hand addressed envelopes; 16x16 pixel array as input
  - 3 hidden layers (768, 192, 30 units) – 2 layers for feature detection).
  - Many edges ignored (only 9760 weights in network).
  - Trained on 7300 digits; tested on 2000 additional digits.
  - 1% error on training set. 5% on test data.
- Backgammon (Neurogammon, 1989):
  - Program beat other programs but lost to a human.

Neural Networks

- A technique to mimic the human brain (how it learns, retains info, etc).

- **Neuron**
  - Major cells of the nervous system
  - Simple computing element.



• Dendrites receive impulses from other neurons via synapses.

• Synapses can be excitatory or inhibitory, and respectively raise or lower the electric potential of the target neuron.
•Electrical potential = ion concentration inside the nucleus vs. outside.

Once electric potential exceeds a threshold, the cell “fires” and sends a signal (action potential) down the axon to synapses and from there to other neurons.

Why Try To Mimic The Brain?

- Brain is highly parallel and can learn by changing/adjusting the connections between neurons.

- Failure of a single neuron is unlikely to affect overall operation (robust).

- Information is encoded in:
  - <u>Connection pattern</u> of neurons,
  - <u>Amplification</u> of signals in dendrites,
  - <u>Activation function</u> and <u>threshold values</u> controlling the firing of a cell.

Why Try To Mimic The Brain?

- A neuron generates a response (action potential) when its inputs raise the electrical potential above a threshold.

- Since the inputs are amplified (<u>weighted</u>) by dendrites, this is equivalent to detecting an <u>input pattern</u>.

- More neurons and connections between neurons means more complex patterns can be learned and detected.

Artificial Neural Networks

- An "artificial neural network" consists of a large, densely inter-connected arrangements of simple computing elements

- Each computing element is designed to mimic a neuron.

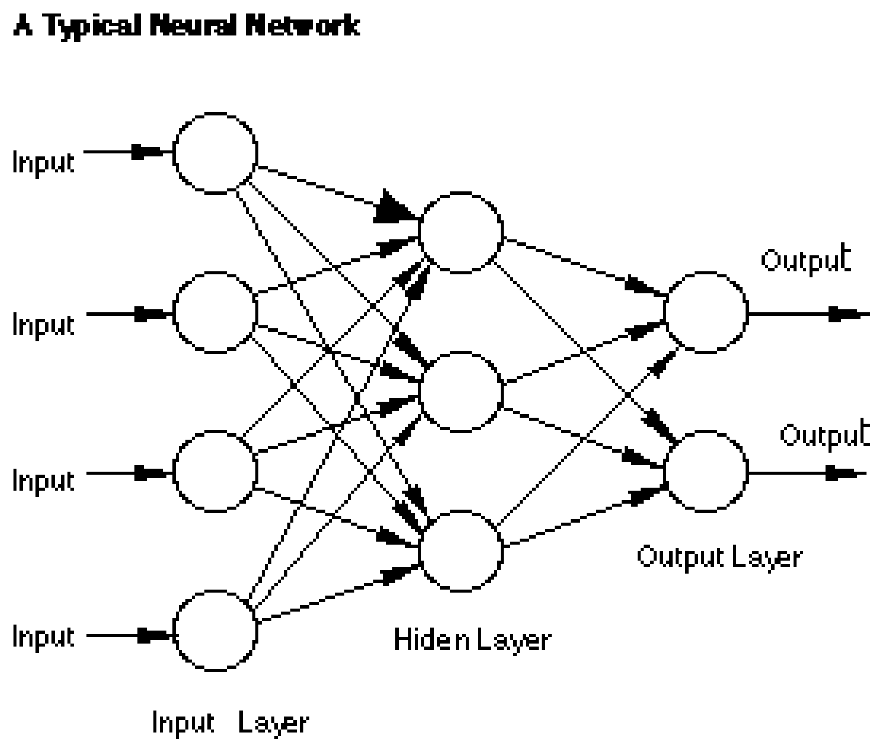
- There are different types/categories of Neural Networks.

- The type/category depends on:
  - Topology of computing elements;
  - Characteristics of computing elements;
  - How the network learns.

- Different networks are appropriate for different applications.

Illustration of an Artificial Neural Network

- Example of a multi-layer, feed-forward neural network:



Types of Neural Nets

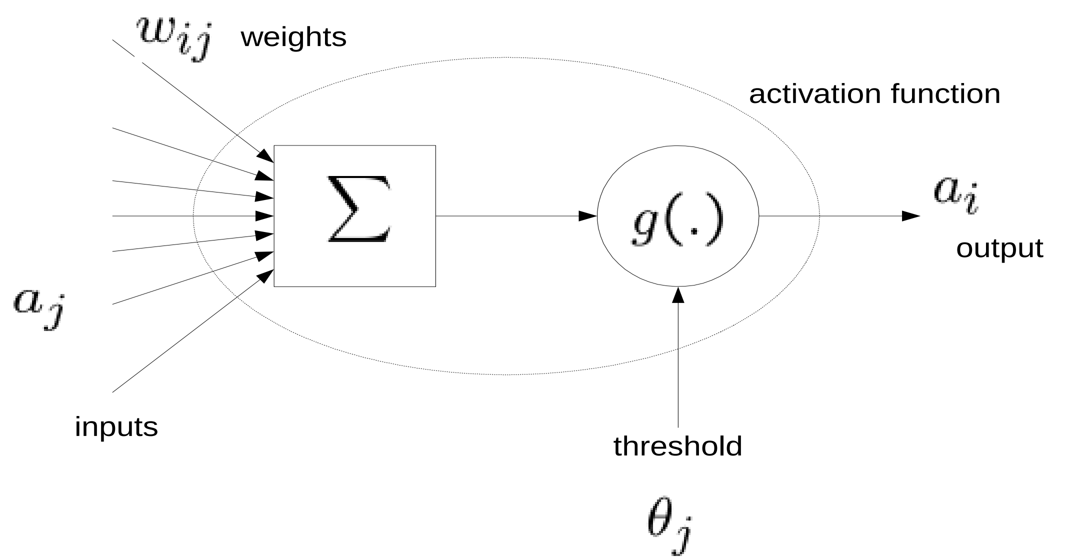
CLASS	EXAMPLES	CHARACTERISTICS
Feed-forward NN (no feedback)	Single-layer perceptron (SLP)	❑ Simplest NN ❑ Uses supervised learning. ❑ Can handle only linearly separable problems.
	Multi-layer perceptron (MLP)	☐ Can handle non-linear problems. ☐ Sometimes converge to local minima. ☐ Prone to overfitting.
	Radial Basis Function (RBF) Networks	☐ Alternative to MLP. ☐ Do not converge to local minima. ☐ Prone to overfitting.
	Self-organizing maps (SOM)	☐ Uses an unsupervised learning method. ☐ Good for producing visualizations of data (groups similar objects).

Types of Neural Nets

CLASS	EXAMPLES	CHARACTERISTICS
Recurrent NN (involves feedback)	Hopfield NN	□ Fully-connected network. □ Knowledge is encoded, not learned. □ Uses unsupervised learning. □ Like associative memory.
Stochastic NN (involves randomness)	Boltzmann Machine	□ Seen as the stochastic counterpart of Hopfield. □ Based on simulated annealing. □ Practical when connectivity is constrained (not fully-connected).
Modular NN (collection of smaller networks)	Committee of Machines (CoM)	□ Collection of networks that vote on a particular example. □ Gives better result than individual networks.

Simple Computing Element (1 of 2)

- Referred to as a neuron (since that's what we mimic).



- Note similarity with a physical neuron (corresponding structure in parentheses):
  - Inputs (synapses) are weighted (dendrites) and summed (nucleus).
  - A threshold, θ, and activation function, g, transform the sum (electric potential) into an output (action potential).

Simple Computing Element (2 of 2)

- We see two computations occurring inside a neuron:

  1. Linear weighted summation of incoming values.

  2. Non-linear activation function for calculation of output.

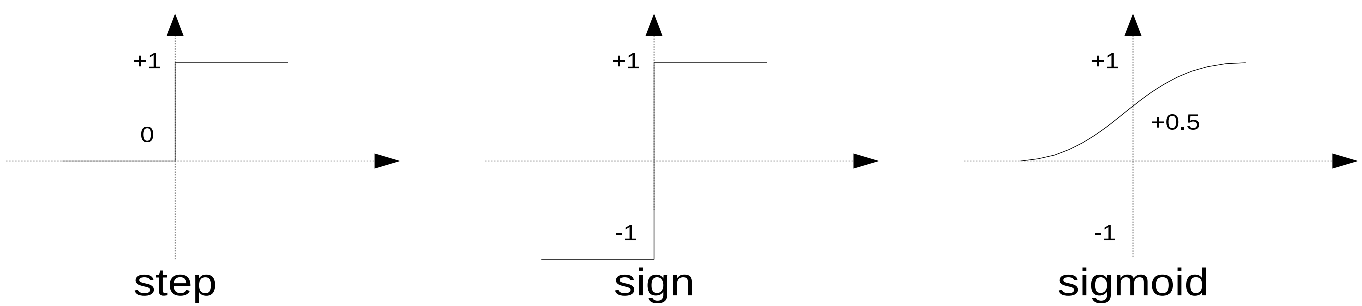
$$a_i = g_i(\sum_j w_{ij} a_j - \theta_i)$$

- The threshold θ_i can be replaced by an extra input fixed to -1 whose weight is then the threshold value
  - This is just an alternative arrangement to account for the threshold.

Activation Functions

- Common activation functions are the step/sign function and sigmoid function (shown below).

- Functions map the sum (after thresholding) to the range [0,1] or [-1,1]
  - We want the neuron active (output close to 1) when suitable inputs are present, and inactive (close to 0) when wrong inputs are present.
  - May alternatively want to penalize the wrong inputs (-1 output).



$$sig(x) = \frac{1}{1 + \exp(-x)}$$

Sigmoid Activation Function

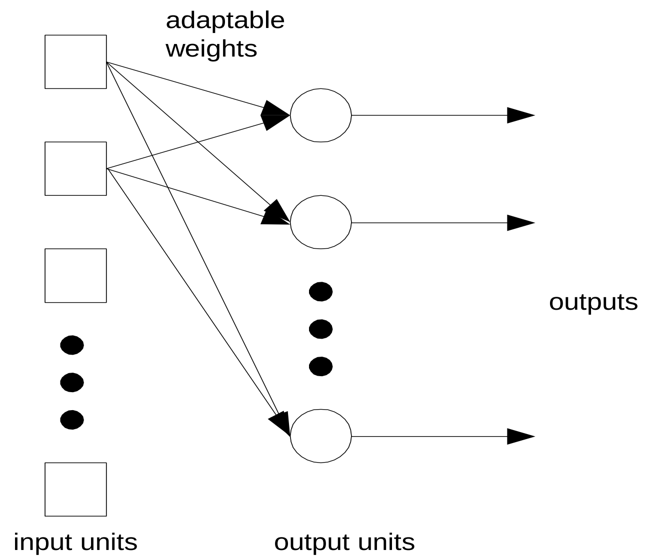
- The sigmoid activation function is nice because it is differentiable.
- Also, the derivative is simple to compute:

$$
\begin{aligned}
\frac{d}{dx} sig(x) &= \frac{d}{dx}(1 + \exp(-x))^{-1} \\
&= exp(-x)(1 + \exp(-x))^{-2} \\
&= (1 + \exp(-x))^{-1}\left(\frac{exp(-x)+1-1}{1+\exp(-x)}\right) \\
&= sig(x)(1 - sig(x))
\end{aligned}
$$

- So, its derivative can be computed from the function itself; this is convenient when training back-propagation networks (more later).

Simple Perceptron Network

- A simple network with one layer of neurons connected to a number of inputs is called a perceptron:



- Example of a “feed-forward” network. Inputs applied, work their way through the network via weighting and activation functions, result in outputs.

Supervised Learning

- Simple perceptron is a network which uses **supervised learning**.
  - For a given input $\mathbf{I}^\mu$ we know the desired output ζ^μ.
  - We compute the output $\mathbf{O}^\mu$ using the weights and activation function.
  - We compare $\mathbf{O}^\mu$ with ζ^μ and **adjust weights and thresholds** to **minimize** the **differences** (error) in the desired output vs. actual output.

Simple Perceptron Network Capabilities

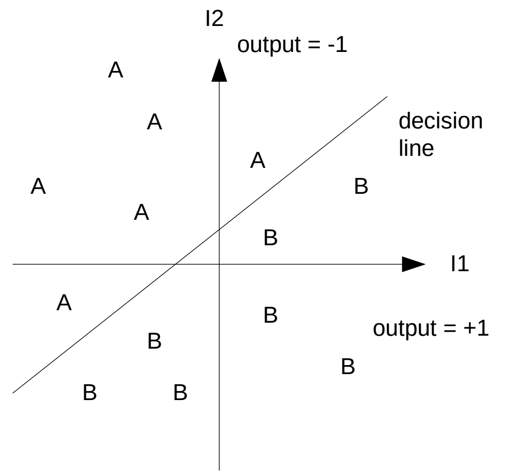
- **Question:** What can a simple perceptron network do?
- Consider a single output and assume the activation function is the sign fn.

$$O_i = \text{sign}(\sum_j w_{ij} I_j)$$

- If sum ¸ 0, then output is +1. If sum < 0, then output is -1.
- **The output divides the input into 2 categories (+1 and -1) by drawing a "decision boundary" (a hyper-plane).**

Visualization

- Consider a two-input problem space, and apply a simple perceptron:



- Output switches between +1 and -1 occur when input summation is zero:

$$\sum_j w_{ij} I_j = w_{i0}(-1) + w_{i1} I_1 + w_{i2} I_2 = 0 \qquad I_2 = -\frac{w_{i1}}{w_{i2}} I_1 - \frac{w_{i0}}{w_{i2}}$$

- So training (learning) is accomplished by **modifying the weights (and threshold)** of the network:
  - Changing the weights alters the **decision line**, and thereby facilitates getting the desired output for a given input.

Training Algorithm (Rosenblatt, 1959)

- Steps:

  1. Initialize weights and thresholds to small random numbers.

  2. Present a new input $\mathbf{I}^\mu$ and desired output ζ^μ to the network.

  3. Calculate the current (actual) output $\mathbf{O}^\mu$ from the network.

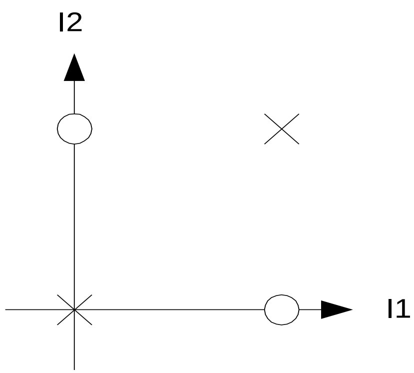
  4. Adapt weights $\mathbf{w}_i$ Ã $\mathbf{w}_i + \eta\ (\zeta^\mu - \mathbf{O}^\mu)\ \mathbf{I}^\mu$ where η **2 (0,1)** is the **learning rate**.

  5. Repeat for many desired input-output pairs until no error.

- NOTE: weights will cease to change once output becomes correct since $(\zeta^\mu - \mathbf{O}^\mu)$ will be zero.

The XOR Problem

- Single layer networks have a big problem! What happens if no decision boundary?
- Consider XOR problem:



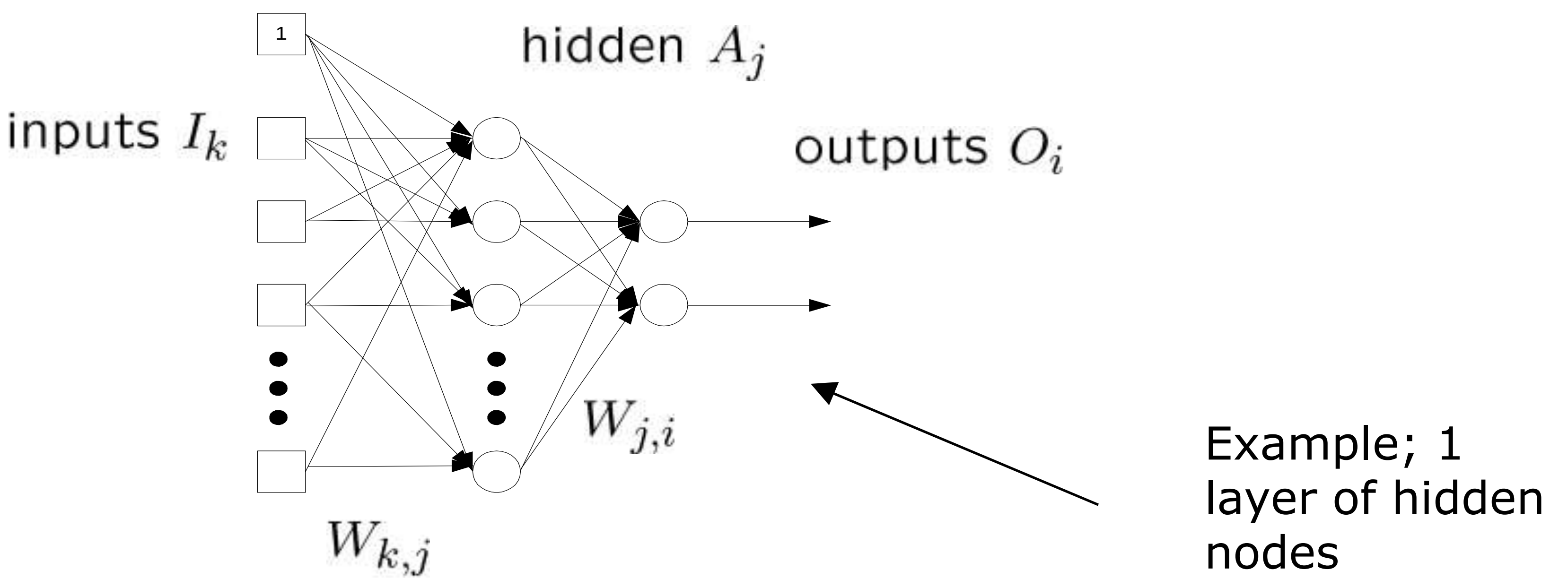
I1	I2	O
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

- NO SINGLE decision line that separates the two categories. Problem is not **linearly separable** which is required for single layer networks.
- So,… how good can a neural network be if it can't even solve the XOR problem?

Multi-Layer Neural Networks (1 of 2)

- Overcomes the limitation of simple single layer networks; consists of several layers of **hidden nodes** between input and output nodes.

- Originally ignored since no training methods had been developed.

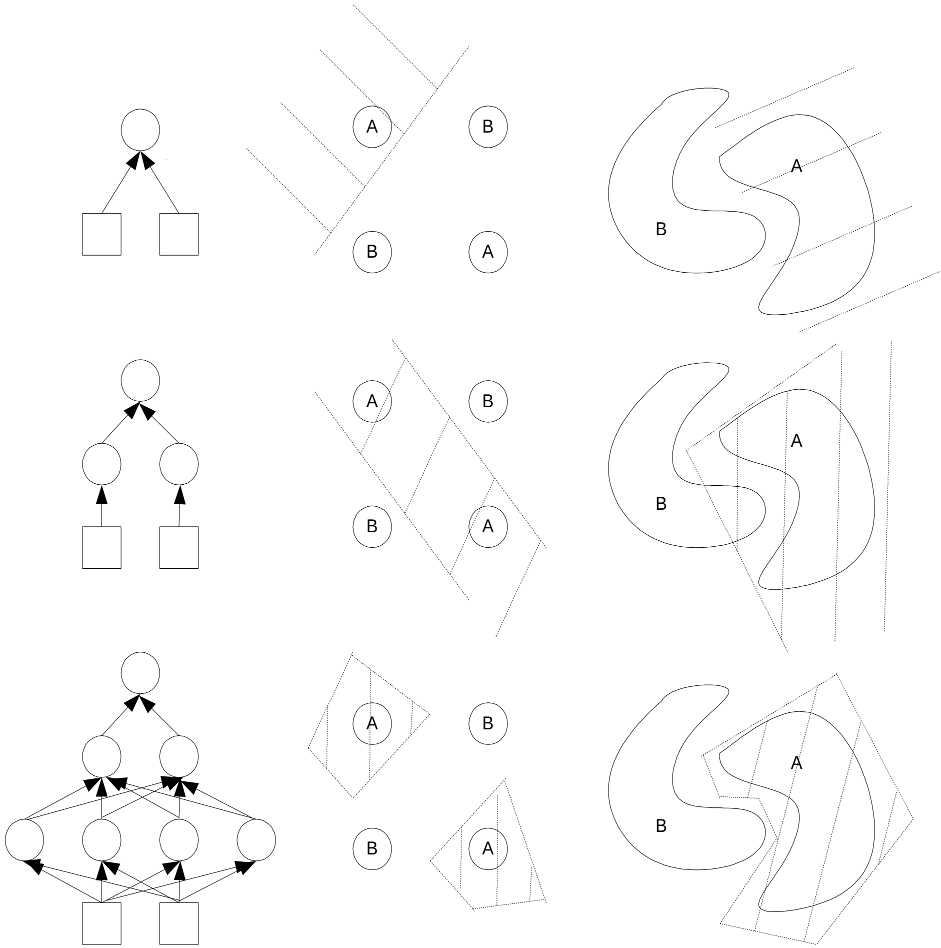


Issues in Multi-Layer Networks

- How many layers of hidden nodes are needed?
- How many nodes in each hidden layer?
- How do we do training?

Hidden Layers and Geometry

- Hidden layers determine the geometric regions which can be categorized:



- Regions shown with step or sign activation function.
- Sigmoid would smooth out the sharp corners.

Training Multi-Layer Networks (Back Propagation)

- Training is easiest with **sigmoid function** so we will consider that; need to be able to take a **derivative**.
- **Back-propagation** is similar to training a simple perceptron network:
  - Present input and compute resulting output.
  - If resulting output and desired output agree, do nothing. Otherwise adjust weights to reduce the error. (What do we call this type of learning?)
- The novelty of back-propagation is that weights are updated by assessing and **dividing** the **error** in the output **among all the weights** in the network.
  - Why is this important?
- **We can derive a means of reducing output error and use the sigmoid function which has a derivative.**
  - Essentially **gradient descent**. (How will this affect learning?)

Back-Propagation Derived (1 of 5)

- Use the sigmoid function:

$$sig(x) = \frac{1}{1 + \exp(-x)}$$

- Assume we present input I, get actual output O and our target output is T.
- Measure the error in the output as:

$$E = \frac{1}{2} \sum_i (T_i - O_i)^2$$

- Notice that although E is written as a function if T and O, it is really a function of the weights (and thresholds) in the network.
- **To minimize the error E, we take its partial derivative wrt. each weight in the network; this becomes a descent direction for minimization.**
- **The rest is just math…**

Back Propagation Derived (2 of 5)

- The output of each hidden node is:

$$A_j = sig(\sum_k W_{k,j} I_k)$$

- The output of each output node is:

$$O_i = sig(\sum_j W_{j,i} A_j)$$

Back Propagation Derived (3 of 5)

- Take partial derivative of E wrt. weights from hidden nodes to output nodes:

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(T_i - O_i)\frac{\partial}{\partial W_{j,i}} O_i \\
&= -(T_i - O_i)\frac{\partial}{\partial W_{j,i}} sig\left(\sum_j W_{j,i}A_j\right) \\
&= -(T_i - O_i) g\left(\sum_j W_{j,i}A_j\right) A_j \\
&= -A_j \Delta_i
\end{aligned}$$

- Where g(.) is the derivative of the sigmoid function and:

$$\Delta_i = (T_i - O_i) g\left(\sum_j W_{j,i}A_j\right)$$

Back Propagation Derived (4 of 5)

- Take partial derivative of E wrt. weights from input nodes to hidden nodes:

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (T_i - O_i) \frac{\partial}{\partial W_{k,j}} O_i \\
&= -\sum_i (T_i - O_i) \frac{\partial}{\partial W_{k,j}} sig\left(\sum_j W_{j,i} A_j\right) \\
&= -\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i} \frac{\partial}{\partial W_{k,j}} A_j \\
&= -\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i} \frac{\partial}{\partial W_{k,j}} sig\left(\sum_k W_{k,i} I_k\right) \\
&= -\left[\sum_i (T_i - O_i) g\left(\sum_j W_{j,i} A_j\right) W_{j,i}\right] g\left(\sum_k W_{k,j} I_k\right) I_k \\
&= -\left[\left(\sum_i W_{j,i} \Delta_i\right) g\left(\sum_k W_{k,i} I_k\right)\right] I_k \\
&= -\Delta_j I_k
\end{aligned}
$$

- Where g(.) is the derivative of the sigmoid function and:

$$\Delta_j = g\left(\sum_k W_{k,i} I_k\right) \sum_i W_{j,i} \Delta_i .$$

Back-Propagation Derived (5 of 5)

- The previously derived partial derivatives give a descent direction.

The network is trained in the following manner:

- Initial weights are determined randomly.
- An input is applied to the network and output is determined.
- Weights from hidden to output nodes are updated:

$$W_{j,i} = W_{j,i} + \alpha \times A_j \times \Delta_i$$

- Weights from input to hidden nodes are updated:

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

  - Here, α **2 (0,1)** is the **learning rate.**

Additional Details On Back-Propagation Training

- The input-output pairs used for training are the **training set.**
- Idea is to train (update weights) using a **good**, **trusted** and **representative** sample of inputs.
  - When new (unknown) inputs are applied that were not used in training, the network should/will give the correct input.
  - This is because similar inputs share similar characteristics.
- **Terminology**: Given a set of I/O pairs, presenting each I/O pair to the network and updating weights is referred to as **one epoch** of training.
- **Training sets** should be applied to the network multiple times until the network weights stop changing.; i.e., we should do **multiple epochs** and the network becomes "better" after each epoch.

Problems With Back-Propagation Training

- This training method does have some problems.
  - **Efficiency** – slow to learn and need to do many epochs of training to get good network performance.
  - **Locality** – training based on gradient descent which means we might converge to a local minimum of E which will give bad answers for inputs not in the training set.

Example Applications

- NETtalk (1987):
  - Network trained to pronounce English text.
  - Sliding window of text; 7 characters in stream presented to network.
  - 80 hidden nodes, 26 output nodes (encoded phonemes).
  - Trained on 1024 words from side-by-side English/phoneme source.
  - 10 training epochs gave intelligible speech; 50 epochs gave 95% accuracy.
  - Performance on actual data as 78%.
- Character Recognition (1989):
  - Read zip codes on hand addressed envelopes; 16x16 pixel array as input
  - 3 hidden layers (768, 192, 30 units) – 2 layers for feature detection).
  - Many edges ignored (only 9760 weights in network).
  - Trained on 7300 digits; tested on 2000 additional digits.
  - 1% error on training set. 5% on test data.
- Backgammon (Neurogammon, 1989):
  - Program beat other programs but lost to a human.