Introduction to Artificial Neural Networks

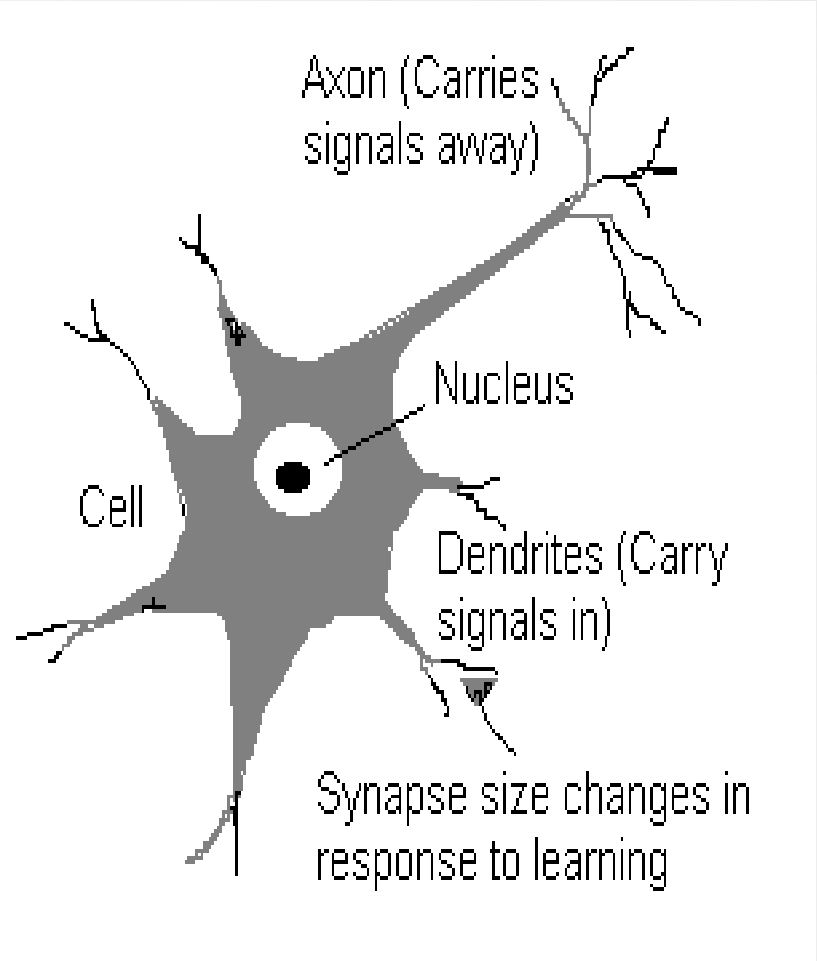
Ahmed Guessoum
Natural Language Processing and Machine Learning Research Group
Laboratory for Research in Artificial Intelligence
Université des Sciences et de la Technologie Houari Boumediene

Lecture Outline

- **The Perceptron**
- **Multi-Layer Networks**
  - Nonlinear transfer functions
  - Multi-layer networks of nonlinear units (sigmoid, hyperbolic tangent)
- **Backpropagation of Error**
  - The backpropagation algorithm
  - Training issues
    - Convergence
    - Overfitting
- **Hidden-Layer Representations**
- **Examples: Face Recognition and Text-to-Speech**
- **Backpropagation and Faster Training**
- **Some Open problems**

In the beginning was … the Neuron!

- A neuron (nervous system cell): many-inputs / one-output unit
- output can be *excited* or *not excited*
- incoming signals from other neurons determine if the neuron shall excite ("fire")
- The output depends on the attenuations occuring in the *synapses:* parts where a neuron communicates with another



The Synapse Concept

- The synapse resistance to the incoming signal can be changed during a "learning" process [Hebb, 1949]

Hebb's Rule:

If an input of a neuron is repeatedly and persistently causing the neuron to fire, then a metabolic change happens in the synapse of that particular input to reduce its resistance

Connectionist (Neural Network) Models

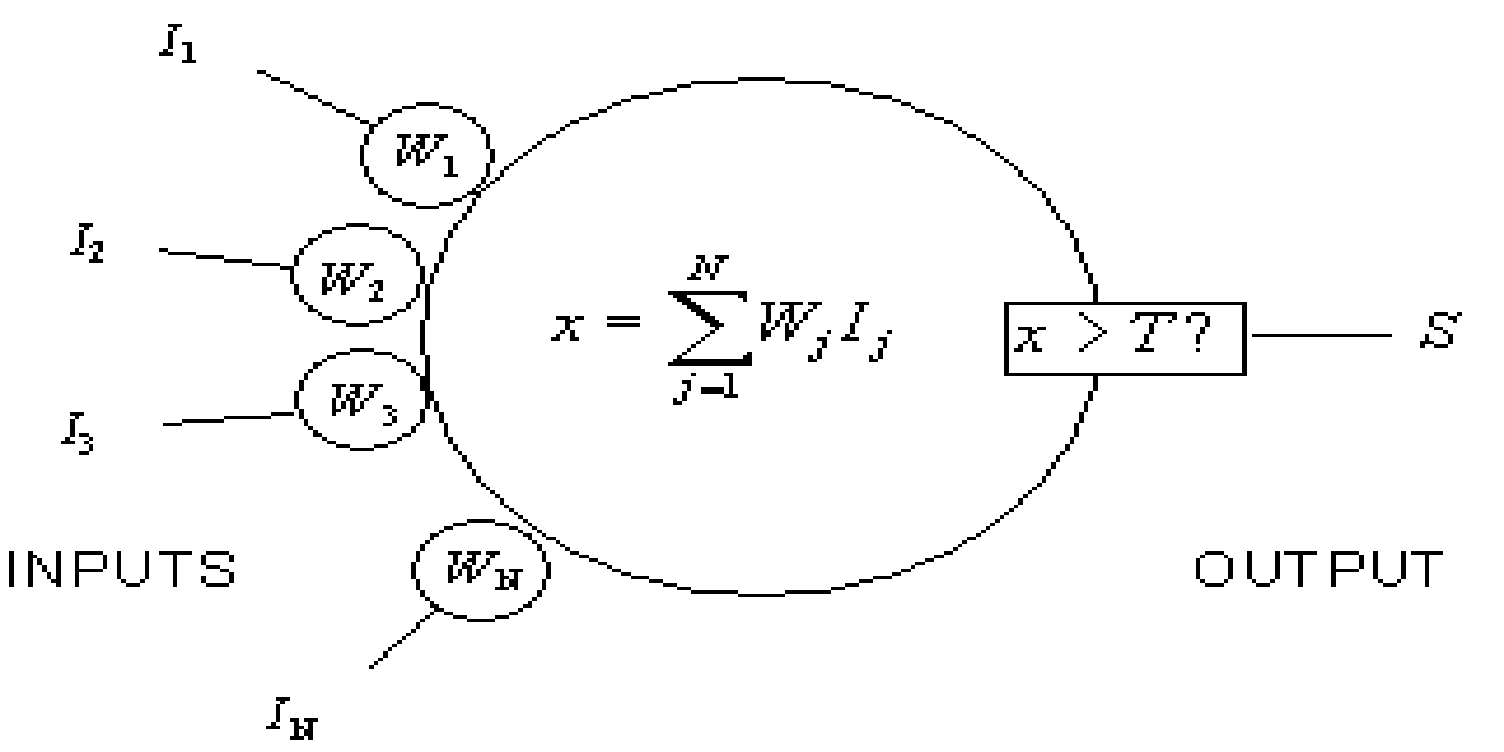
- **Human Brain**
  - Number of neurons: ~100 billion (10^{11})
  - Connections per neuron: ~10-100 thousand (10^4 – 10^5)
  - Neuron switching time: ~ 0.001 (10^{-3}) second
  - Scene recognition time: ~0.1 second
  - 100 inference steps doesn't seem sufficient!

    ➔ Massively parallel computation

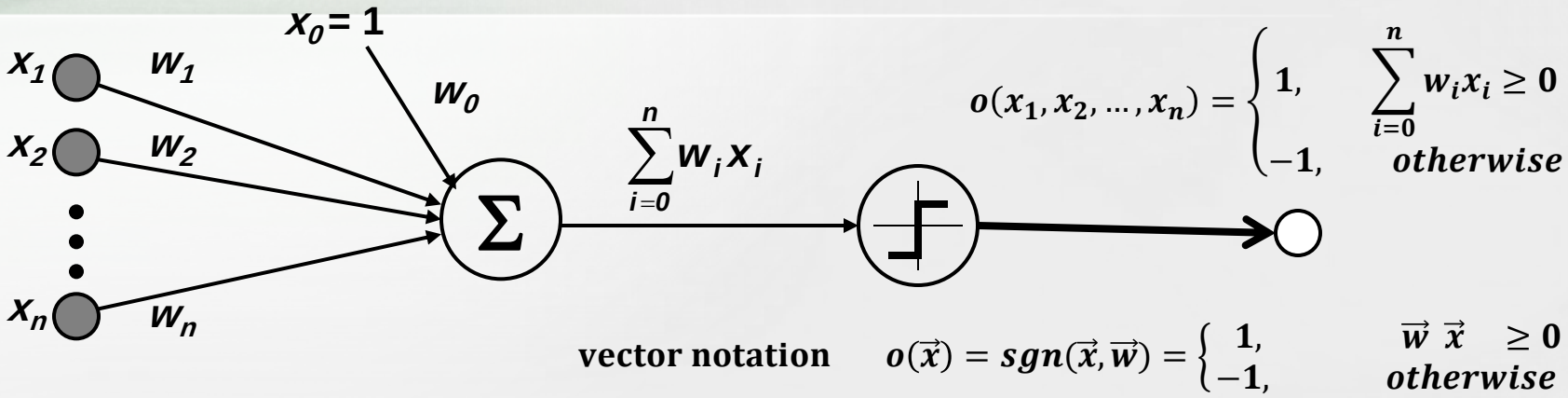
- (List of animals by number of neurons: https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons)

Mathematical Modelling

The neuron calculates a weighted **x** (or **net**) sum of inputs and compares it to a threshold **T**. If the sum is higher than the threshold, the output **S** is set to 1, otherwise to -1.



The Perceptron

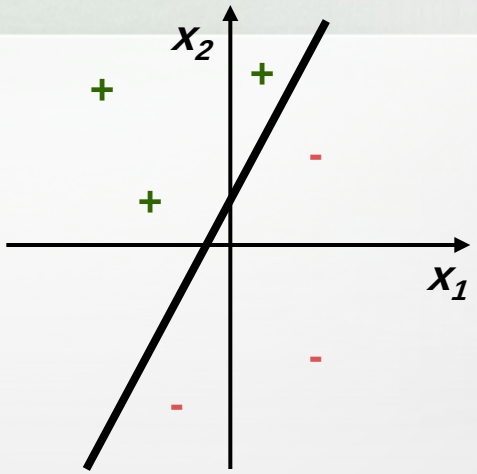


- **Perceptron: Single Neuron Model**
  - Linear Threshold Unit (LTU) or Linear Threshold Gate (LTG)
  - Net input to unit: defined as a linear combination $net(x) = \sum_{i=0}^{n} w_i x_i$
  - Output of unit: threshold (activation) function on net input (threshold $\theta = -w_0$)
- **Perceptron Networks**
  - Neuron is modeled using a unit connected by weighted links w_i to other units
  - Multi-Layer Perceptron (MLP)

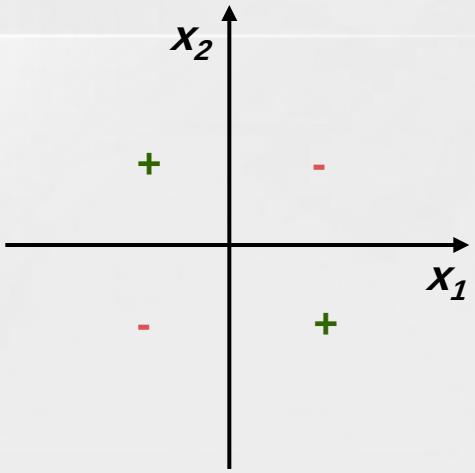
Connectionist (Neural Network) Models

- **Definitions of Artificial Neural Networks (ANNs)**
  - "… a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes." - DARPA (1988)
- **Properties of ANNs**
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically

Decision Surface of a Perceptron



Example A



Example B

- **Perceptron: Can Represent *Some* Useful Functions (And, Or, Nand, Nor)**
  - LTU emulation of logic gates (McCulloch and Pitts, 1943)
  - e.g., What weights represent $g(x_1, x_2) = AND(x_1, x_2)$? $OR(x_1, x_2)$? $NOT(x)$?

    ($w_0 + w_1 \cdot x_1 + w_2 \cdot x_2$ $w_0 = -0.8$ $w_1 = w_2 = 0.5$ $w_0 = -0.3$)
- **Some Functions are *Not* Representable**
  - e.g., not linearly separable
  - Solution: use networks of perceptrons (LTUs)

Learning Rules for Perceptrons

- **Learning Rule ≡ Training Rule**
  - Not specific to supervised learning
  - Idea: Gradual building/update of a model
- **Hebbian Learning Rule (Hebb, 1949)**
  - Idea: if two units are both active ("firing"), weights between them should increase
  - $w_{ij} = w_{ij} + r\, o_i\, o_j$

    where r is a learning rate constant
  - Supported by neuropsychological evidence

Learning Rules for Perceptrons

- **Perceptron Learning Rule (Rosenblatt, 1959)**
  - **Idea**: when a target output value is provided for a single neuron with fixed input, it can incrementally update weights to learn to produce the output
  - Assume binary (boolean-valued) input/output units; single LTU
  - $$w_i \leftarrow w_i + \Delta w_i$$

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

    where $t = c(x)$ is target output value, o is perceptron output, r is small learning rate constant (e.g., 0.1)
  - Convergence proven for D linearly separable and r small enough

Perceptron Learning Algorithm

- **Simple Gradient Descent Algorithm**
  - Applicable to concept learning, symbolic learning (with proper representation)
- **Algorithm *Train-Perceptron* ($D \equiv \{<x, t(x) \equiv c(x)>\}$)**
  - Initialize all weights w_i to random values
  - WHILE not all examples correctly predicted DO

    FOR each training example $x \in D$

    Compute current output $o(x)$

    FOR $i = 1$ to n

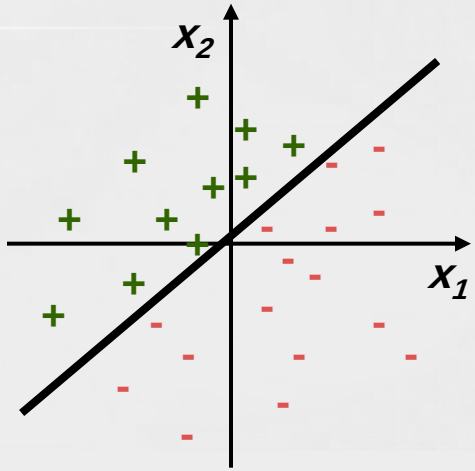
    $w_i \leftarrow w_i + r(t - o)x_i$ // perceptron learning rule

Perceptron Learning Algorithm

- **Perceptron Learnability**
  - Recall: can only learn $h \in H$ - i.e., linearly separable (LS) functions
  - Minsky and Papert, 1969: demonstrated representational limitations
    - e.g., parity (n-attribute XOR: $x_1 \oplus x_2 \oplus \ldots \oplus x_n$)
    - e.g., symmetry, connectedness in visual pattern recognition
    - Influential book *Perceptrons* discouraged ANN research for ~10 years
  - NB: "Can we transform learning problems into LS ones?"

Linear Separators

- **Functional Definition**
  - $f(x) = 1$ if $w_1x_1 + w_2x_2 + \ldots + w_nx_n \geq \theta$, 0 otherwise
  - θ: threshold value



Linearly Separable (LS) Data Set

- **Non Linearly Separable Functions**
  - Disjunctions: $c(x) = x_1' \vee x_2' \vee \ldots \vee x_m'$
  - m of n: $c(x)$ = at least 3 of $(x_1', x_2', \ldots, x_m')$
  - Exclusive OR (XOR): $c(x) = x_1 \oplus x_2$
  - General DNF: $c(x) = T_1 \vee T_2 \vee \ldots \vee T_m$; $T_i = l_1 \wedge l_2 \wedge \ldots \wedge l_k$
- **Change of Representation Problem**
  - Can we transform non-LS problems into LS ones?
  - Is this meaningful? Practical?
  - Does it represent a significant fraction of real-world problems?

Perceptron Convergence

- **Perceptron Convergence Theorem**
  - Claim: If there exists a set of weights that are consistent with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - Caveat 1: How long will this take?
  - Caveat 2: What happens if the data is *not* LS?
- **Perceptron Cycling Theorem**
  - Claim: If the training data is not LS the perceptron learning algorithm will eventually repeat the same set of weights and thereby enter an infinite loop
- **How to Provide More Robustness, Expressivity?**
  - Objective 1: develop algorithm that will find closest approximation
  - Objective 2: develop architecture to overcome representational limitation

Gradient Descent: Principle

- **Understanding Gradient Descent for Linear Units**
  - Consider simpler, unthresholded linear unit:

$$o(\vec{x}) = net(x) = \sum_{i=0}^{n} w_i x_i$$

  - Objective: find "best fit" to *D*
- **Approximation Algorithm**
  - Quantitative objective: minimize error over training data set *D*
  - Error function: sum squared error (SSE)

$$E(\vec{w}) = Error_D(\vec{w}) = \frac{1}{2} \sum_{x \in D} (t(x) - o(x))^2$$

- **How to Minimize?**
  - Simple optimization
  - Move in direction of steepest gradient in weight-error space
    - Computed by finding tangent
    - i.e. partial derivatives (of *E*) with respect to weights (w_i)

Gradient Descent: Derivation of Delta/LMS (Widrow-Hoff) Rule

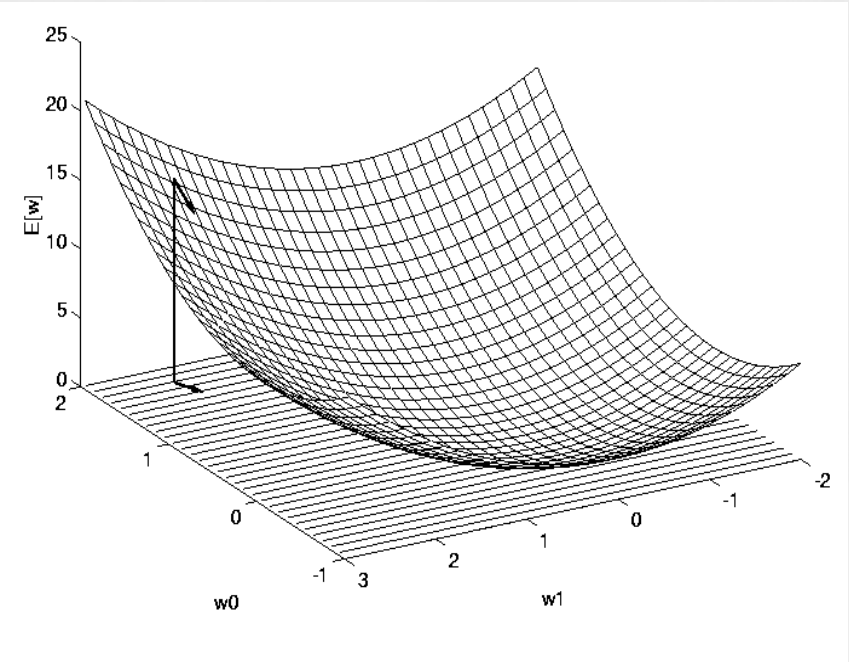
- **Definition: Gradient**

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

- **Modified Gradient Descent Training Rule**

$$\Delta \vec{W} = -r\, \nabla E(\vec{w})$$

$$\Delta w_i = -r\, \frac{\partial E}{\partial w_i}$$



$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left[\frac{1}{2}\sum_{x \in D}\big(t(x) - o(x)\big)^2\right] = \left[\frac{1}{2}\sum_{x \in D}\frac{\partial}{\partial w_i}\big(t(x) - o(x)\big)^2\right]$$

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_{x \in D}\left[2\big(t(x) - o(x)\big)\frac{\partial}{\partial w_i}\big(t(x) - o(x)\big)\right] = \sum_{x \in D}\left[\big(t(x) - o(x)\big)\frac{\partial}{\partial w_i}(t(x) - \vec{w}\ \vec{x})\right]$$

$$\frac{\partial E}{\partial w_i} = \sum_{x \in D}\left[\big(t(x) - o(x)\big)(-x_i)\right]$$

Gradient Descent: Algorithm using Delta/LMS Rule

- **Algorithm *Gradient-Descent* (*D*, *r*)**
  - Each training example is a pair of the form $<x, t(x)>$, where x: input vector; $t(x)$: target vector; r :learning rate
  - Initialize all weights w_i to (small) random values
  - UNTIL the termination condition is met, DO

    Initialize each Δw_i to zero

    FOR each instance $<x, t(x)>$ in D, DO

    Input x into the unit and compute output o

    FOR each linear unit weight w_i, DO

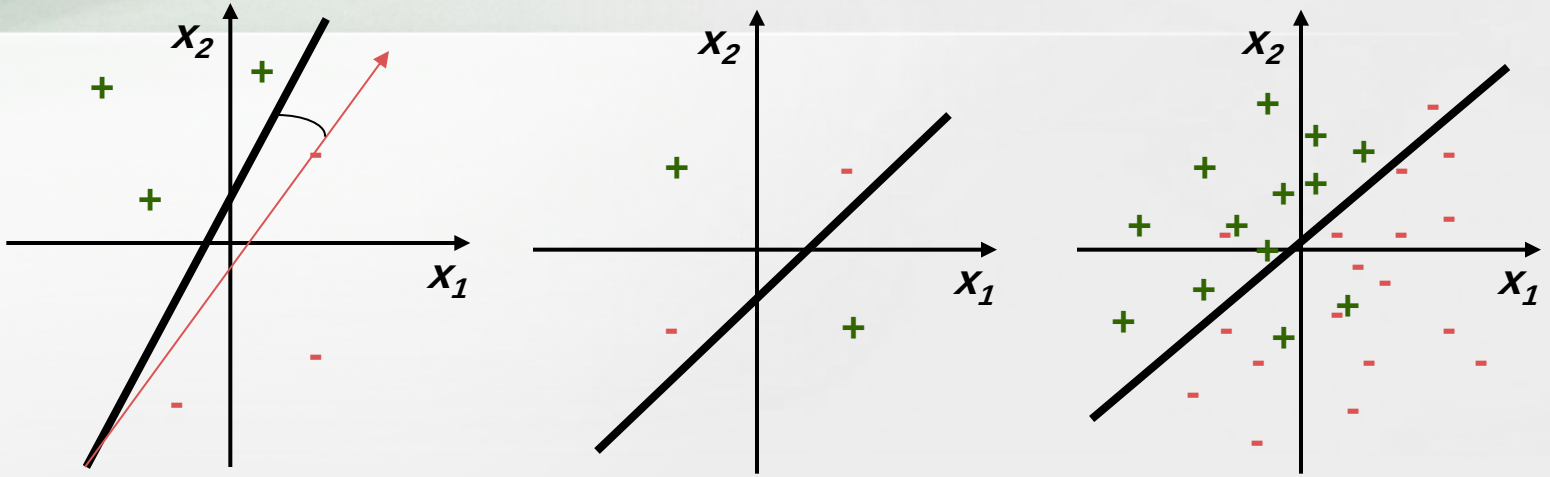
    $\Delta w_i \leftarrow \Delta w_i + r(t - o)x_i$

    $w_i \leftarrow w_i + \Delta w_i$
  - RETURN final w

Gradient Descent: Algorithm using Delta/LMS Rule

- **Mechanics of Delta Rule**
  - Gradient is based on a derivative
  - Significance: later, we will use nonlinear activation functions (*aka* transfer functions, squashing functions)

Gradient Descent: Perceptron Rule versus Delta/LMS Rule



Example A Example B Example C

- **LS Concepts: Can Achieve Perfect Classification**
  - Example A: perceptron training rule converges
- **Non-LS Concepts: Can Only Approximate**
  - Example B: not LS; delta rule converges, but can't do better than 3 correct
  - Example C: not LS; better results from delta rule

Incremental (Stochastic) Gradient Descent

- **Batch Mode Gradient Descent**
  - UNTIL the termination condition is met, DO
    1. Compute the gradient $\nabla E_D(\vec{w})$
    2. Update the weights $\vec{w} \leftarrow \vec{w} - r\,\nabla E_D(\vec{w})$
  - RETURN final *w*
- **Incremental (Online) Mode Gradient Descent**
  - UNTIL the termination condition is met, DO

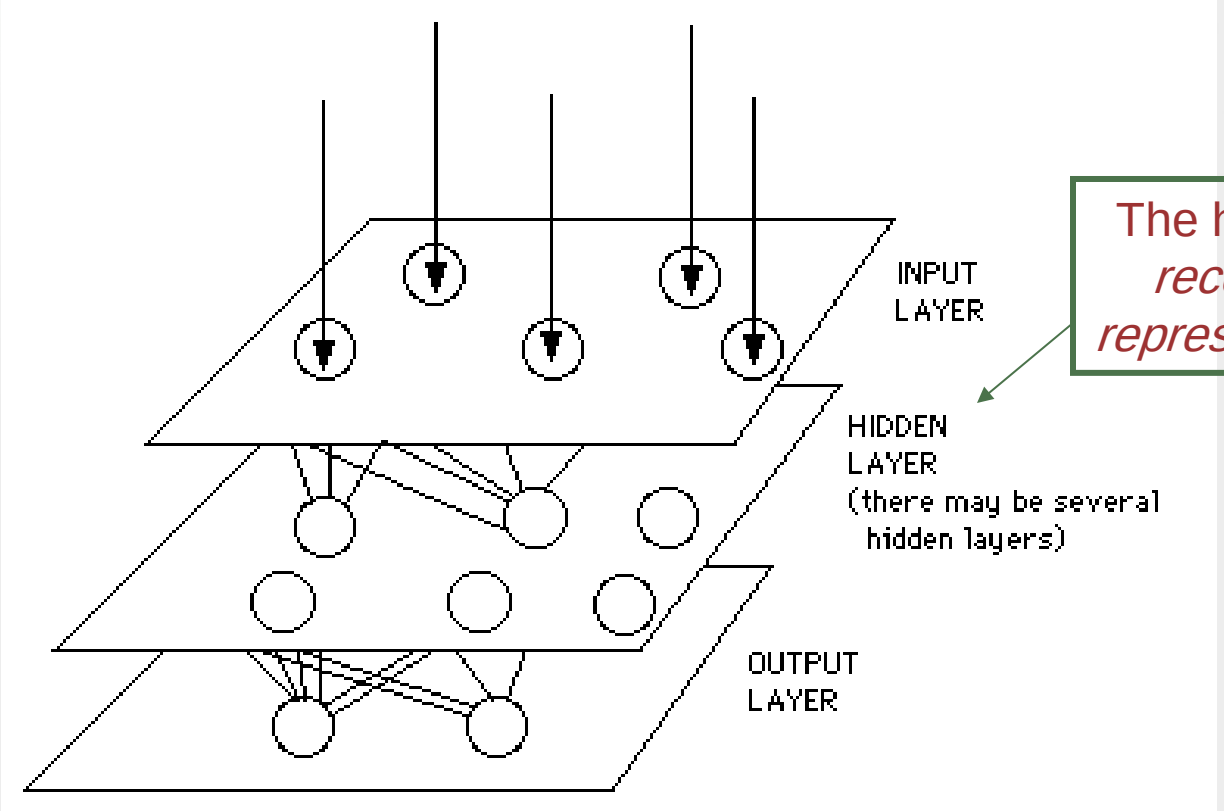
    FOR each <*x*, *t*(*x*)> in *D*, DO
    1. Compute the gradient $\nabla E_d(\vec{w})$
    2. Update the weights $\vec{w} \leftarrow \vec{w} - r\,\nabla E_d(\vec{w})$
  - RETURN final *w*
- **Emulating Batch Mode**
  - $$E_D[\vec{w}] \equiv \frac{1}{2}\left[\sum_{x \in D}(t(x) - o(x))^2\right], \quad E_d[\vec{w}] \equiv \frac{1}{2}(t(x) - o(x))^2$$
  - Incremental gradient descent can approximate batch gradient descent arbitrarily closely if *r* made small enough

GD vs Stochastic GD

- **Gradient Descent**
  - Converges to a weight vector with minimal error regardless whether D is Lin. Separable, given a sufficiently small learning rate is used.
  - Difficulties:
    - Convergence to local minimum can be slow
    - No guarantee to find global minimum
- **Stochastic Gradient Descent** intended to alleviate these difficulties
- Differences
  - In Standard GD, error summed over D before updating W
  - In Standard GD, more computation per weight update step (but larger step size per weight update)
  - Stochastic GD can sometimes avoid falling into local minima
- Both commonly used in Practice

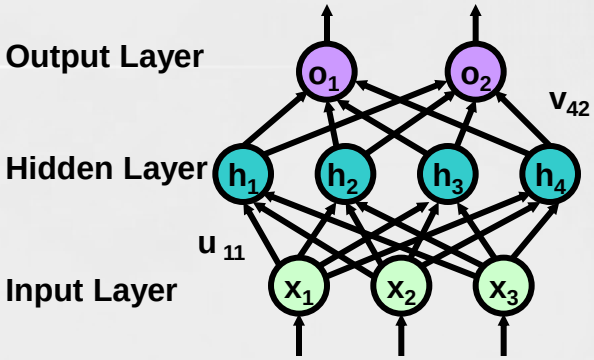
Artificial Neural Networks

Adaptive interaction between individual neurons
Power: collective behavior of interconnected neurons



The hidden layer learns to *recode* (or to *provide a representation* of) the inputs

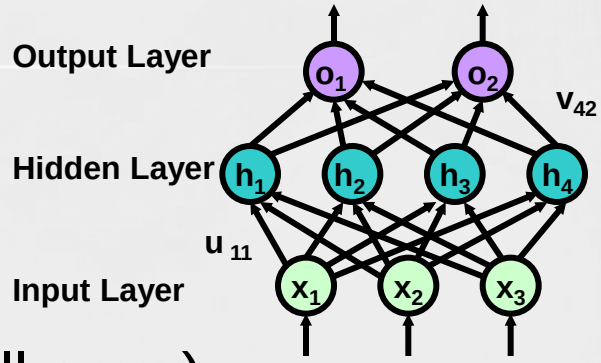
Multi-Layer Networks of Nonlinear Units



- **Nonlinear Units**
  - Recall: activation function *sgn* ($w \bullet x$)
  - Nonlinear activation function: generalization of *sgn*

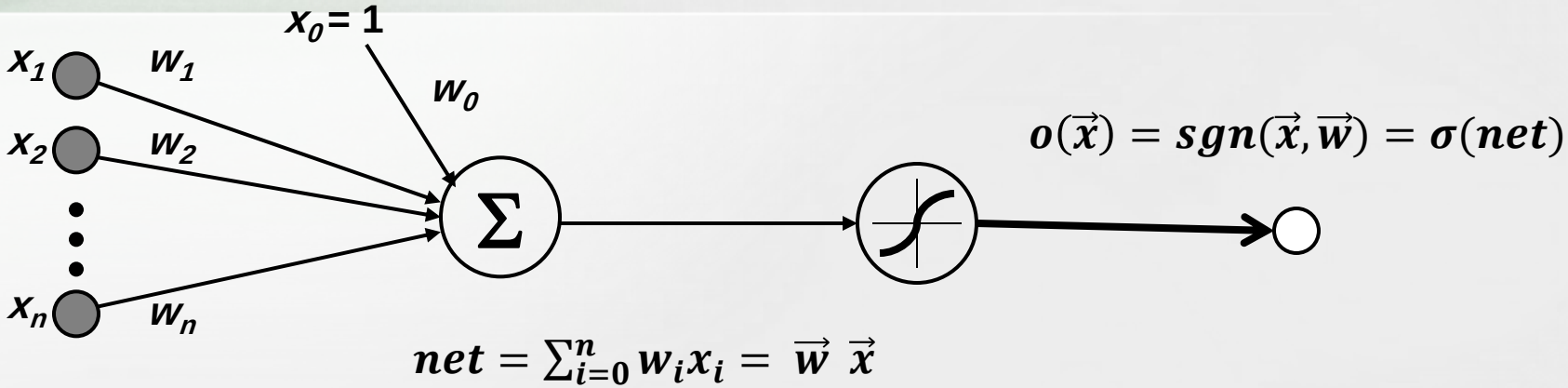
- **Multi-Layer Networks**
  - A specific type: Multi-Layer Perceptrons (MLPs)
  - Definition: a multi-layer feedforward network is composed of an input layer, one or more hidden layers, and an output layer
  - Only hidden and output layers contain perceptrons (threshold or nonlinear units)

Multi-Layer Networks of Nonlinear Units

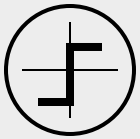


- **MLPs in Theory**
  - Network (of 2 or more layers) can represent any function (arbitrarily small error)
  - Training even 3-unit multi-layer ANNs is NP-hard (Blum and Rivest, 1992)
- **MLPs in Practice**
  - Finding or *designing* effective networks for arbitrary functions is difficult
  - Training is very computation-intensive even when structure is "known"

Nonlinear Activation Functions



- **Sigmoid Activation Function**



  - Linear threshold gate activation function: *sgn* (*w* • *x*)
  - Nonlinear activation (*aka* transfer, squashing) function: generalization of *sgn*
  - σ is the sigmoid function $\sigma(net) = \frac{1}{1+e^{-net}}$
  - Can derive gradient rules to train
    - One sigmoid unit
    - Multi-layer, feedforward networks of sigmoid units (using backpropagation)
- **Hyperbolic Tangent Activation Function** $\sigma(net) = \frac{\sinh(net)}{\cosh(net)} = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}}$

Error Gradient for a Sigmoid Unit

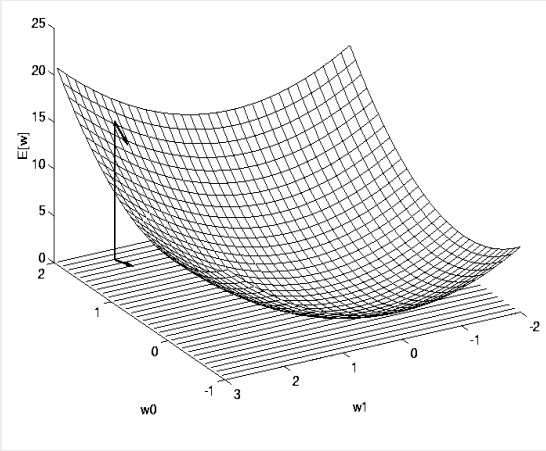
- **Recall: Gradient of Error Function** $\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$
- **Gradient of Sigmoid Activation Function**

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left[\frac{1}{2}\sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\left(t(\vec{x}) - o(\vec{x})\right)^2\right] = \left[\frac{1}{2}\sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\frac{\partial}{\partial w_i}\left(t(\vec{x}) - o(\vec{x})\right)^2\right] =$$

$$= \frac{1}{2}\sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\left[2\left(t(\vec{x}) - o(\vec{x})\right)\frac{\partial}{\partial w_i}\left(t(\vec{x}) - o(\vec{x})\right)\right]$$

$$= \sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\left[\left(t(\vec{x}) - o(\vec{x})\right)\left(-\frac{\partial}{\partial w_i}o(\vec{x})\right)\right]$$

$$= -\sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\left[\left(t(\vec{x}) - o(\vec{x})\right)\left(\frac{\partial o(\vec{x})}{\partial net(\vec{x})}\frac{\partial net(\vec{x})}{\partial w_i}\right)\right]$$



- **But we know:**

$$\frac{\partial o(\vec{x})}{\partial net(\vec{x})} = \frac{\partial \sigma(\overrightarrow{net})}{\partial net(\vec{x})} = \left(o(\vec{x})\right)\left(1 - o(\vec{x})\right) \qquad \frac{\partial net(\vec{x})}{\partial w_i} = \frac{\partial(\vec{w}\ \vec{x})}{\partial w_i} = x_i$$

So:

$$\frac{\partial E}{\partial w_i} = \sum_{\langle \vec{x}|t(\vec{x})\rangle \in D}\left[\left(t(\vec{x}) - o(\vec{x})\right)\left(o(\vec{x})\right)\left(1 - o(\vec{x})\right)x_i\right]$$

The Backpropagation Algorithm

- **Intuitive Idea: Distribute the *Blame* for Error to the Previous Layers**
- **Algorithm *Train-by-Backprop* (*D*, *r*)**
  - Each training example is a pair of the form $\langle x, t(x) \rangle$, where x: input vector; $t(x)$: target vector; r :learning rate
  - Initialize all weights w_i to (small) random values
  - UNTIL the termination condition is met, DO

    FOR each $\langle x, t(x) \rangle$ in D, DO

      Input the instance x to the unit and compute the output $o(x) = \sigma(net(x))$

      FOR each output unit k, DO (calculate its error)

$$\delta_k = o_k(x)(1 - o_k(x))(t_k(x) - o_k(x))$$

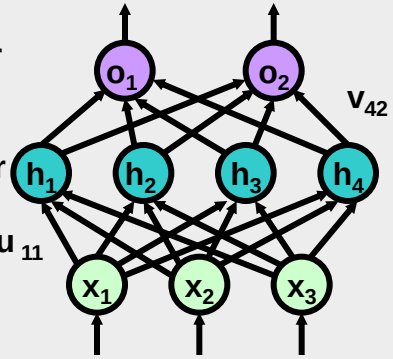
      FOR each hidden unit j, DO

$$\delta_j = h_j(x)(1 - h_j(x)) \sum_{k \in outputs} v_{j,k}\delta_k$$

      Update each $w = u_{i,j}$ $(a = h_j)$ or $w = v_{j,k}$ $(a = o_k)$

$$w_{start\text{-}layer,\ end\text{-}layer} \leftarrow w_{start\text{-}layer,\ end\text{-}layer} + \Delta w_{start\text{-}layer,\ end\text{-}layer}$$

$$\Delta w_{start\text{-}layer,\ end\text{-}layer} \leftarrow r\,\delta_{end\text{-}layer}\,a_{end\text{-}layer}$$



  - RETURN final u, v

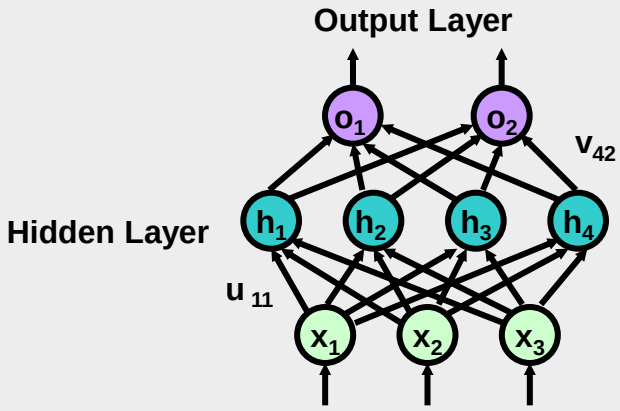
The Backpropagation Algorithm (1)

- **Algorithm *Train-by-Backprop* (*D*, *r*)**
  - *D* is a set of training examples of the form <*x*, *t*(*x*)>, where

    x: input vector;

    t(x): target vector;

    r :learning rate
  - Initialize all weights w_i to (small) random values
  - UNTIL the termination condition is met, DO



The Backpropagation Algorithm (cont.)

FOR each <x, $t(x)$> in D, DO

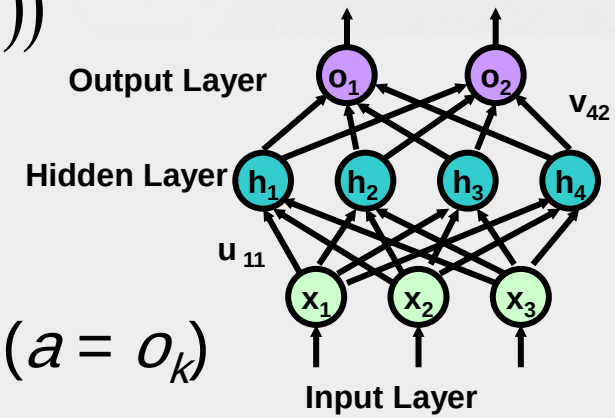
Input the instance x to the unit and compute the output $o(x)$ $= \sigma(net(x))$

FOR each output unit k, DO (calculate its error)

$$\delta_k = o_k(x)(1-o_k(x))(t_k(x)-o_k(x))$$

FOR each hidden unit j, DO

$$\delta_j = h_j(x)(1-h_j(x)) \sum_{k \in outputs} v_{j,k}\delta_k$$



Update each $w = u_{i,j}$ $(a = h_j)$ or $w = v_{j,k}$ $(a = o_k)$

$$w_{start\text{-}layer,\ end\text{-}layer} \leftarrow w_{start\text{-}layer,\ end\text{-}layer} + \Delta w_{start\text{-}layer,\ end\text{-}layer}$$

$$\Delta w_{start\text{-}layer,\ end\text{-}layer} \leftarrow r\,\delta_{end\text{-}layer}\,a_{end\text{-}layer}$$

– RETURN final u, v

Backpropagation and Local Optima

- **Gradient Descent in Backprop**
  - Performed over entire *network* weight vector
  - Easily generalized to arbitrary directed graphs
  - Property: Backpropagation on feedforward ANNs will find a *local* (not necessarily global) error minimum

Backpropagation and Local Optima

- **Backprop in Practice**
  - Local optimization often works well (can *run multiple times*)
  - A weight momentum α is often included:

$$\Delta w_{start\text{-}layer,end\text{-}layer}(n) = r\,\delta_{end\text{-}layer}\,a_{end\text{-}layer} + \alpha\Delta w_{start\text{-}layer,end\text{-}layer}(n-1)$$

  - Minimizes error over training examples

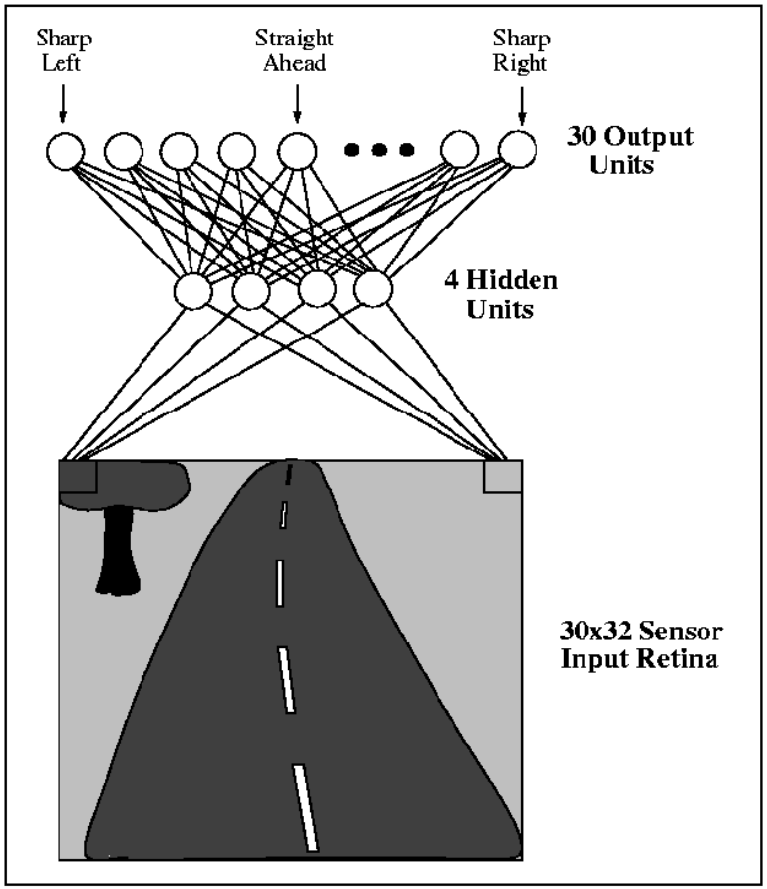
    Generalization to subsequent instances?
  - Training often *very* slow: thousands of iterations over *D* (epochs)
  - Inference (applying network after training) typically very fast
    - Classification
    - Control

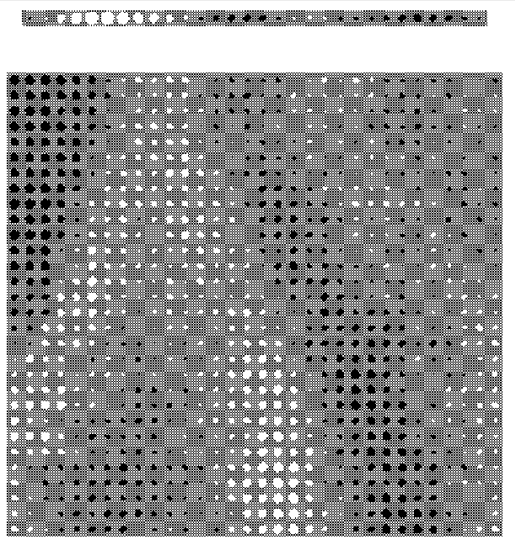
When to Consider Neural Networks

- **Input: High-Dimensional and Discrete or Real-Valued**
  - e.g., raw sensor input
  - Conversion of symbolic data to quantitative (numerical) representations possible
- **Output: Discrete or Real Vector-Valued**
  - e.g., low-level control policy for a robot actuator
  - Similar qualitative/quantitative (symbolic/numerical) conversions may apply
- **Data: Possibly Noisy**
- **Target Function: Unknown Form**
- **Result: Human Readability Less Important Than Performance**
  - Performance measured purely in terms of accuracy and efficiency
  - Readability: ability to explain inferences made using model; similar criteria
- **Examples**
  - Speech phoneme recognition
  - Image classification
  - Financial prediction

Autonomous Learning Vehicle in a Neural Net (ALVINN)

- **Pomerleau *et al***
  - http://www.cs.cmu.edu/afs/cs/project/alv/member/www/projects/ALVINN.html
  - Drives 70mph on highways





Hidden-to-Output Unit Weight Map (for one hidden unit)

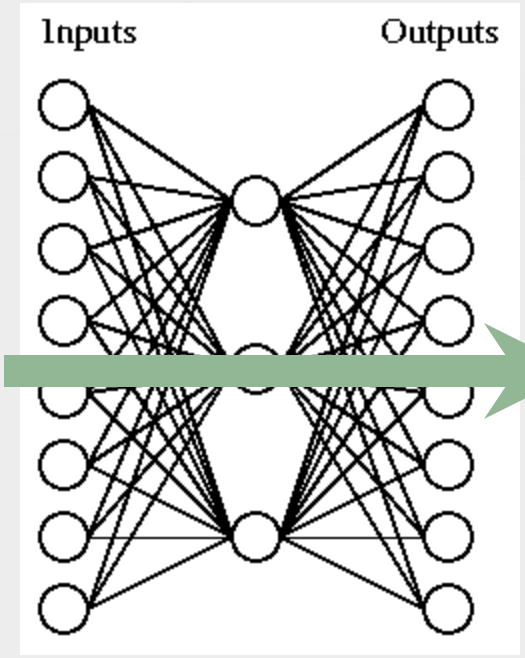
Input-to-Hidden Unit Weight Map (for one hidden unit)

Feedforward ANNs: Representational Power

- **Representational (i.e., Expressive) Power**
  - 2-layer feedforward ANN
    - Any Boolean function
    - Any bounded continuous function (*approximate with arbitrarily small error*) : 1 output (unthresholded linear units) + 1 hidden (sigmoid)
  - 3-layer feedforward ANN: any function (*approximate with arbitrarily small error*): output (linear units), 2 hidden (sigmoid)

Learning Hidden Layer Representations

- **Hidden Units and Feature Extraction**
  - Training procedure: hidden unit representations that minimize error E
  - Sometimes backprop will define new hidden features that are not explicit in the input representation *x*, but which capture properties of the input instances that are most relevant to learning the target function *t*(*x*)
  - Hidden units express *newly constructed features*
  - *Change of representation* to linearly separable *D'*



- **A Target Function (Sparse *aka* 1-of-C, Coding)**

Input									Hidden Values				Output							
1	0	0	0	0	0	0	0	→	0.89	0.04	0.08	→	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	→	0.01	0.11	0.88	→	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	→	0.01	0.97	0.27	→	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	→	0.99	0.97	0.71	→	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	→	0.03	0.05	0.02	→	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	→	0.22	0.99	0.99	→	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0	→	0.80	0.01	0.98	→	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	→	0.60	0.94	0.01	→	0	0	0	0	0	0	0	1

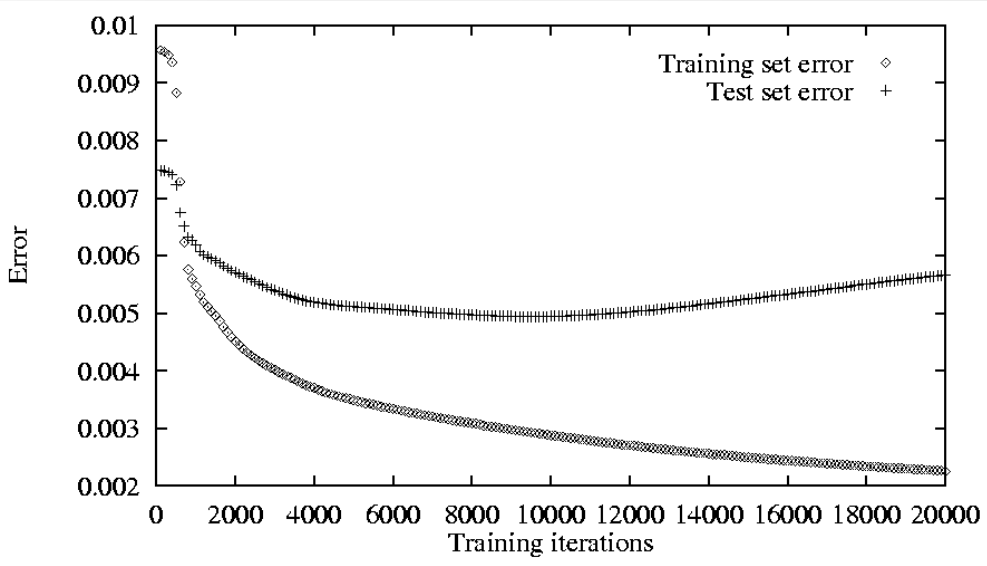
  - ANNs learn discover useful representations at the hidden layers

Convergence of Backpropagation

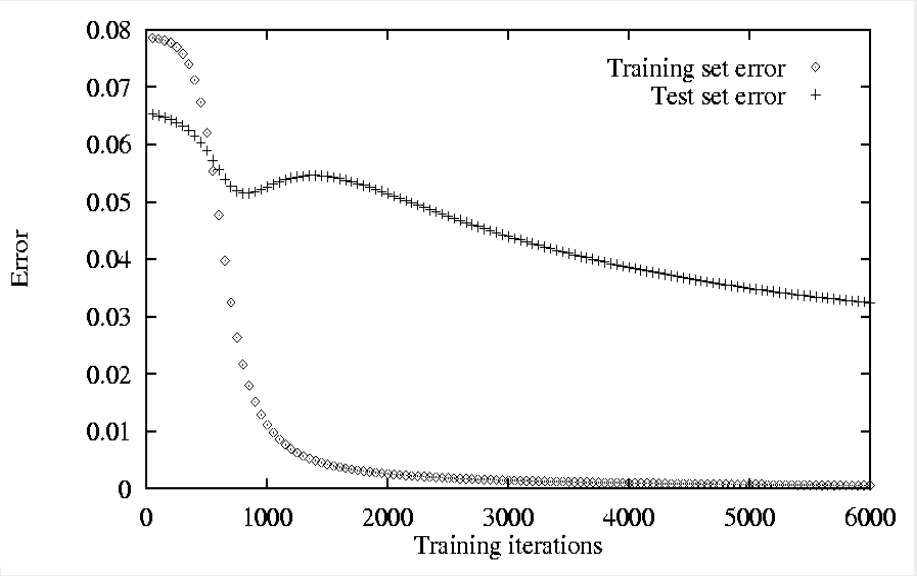
- **No Guarantee of Convergence to Global Optimum**
  - Compare: perceptron convergence (to best $h \in H$, *provided* $h \in H$; i.e., LS)
  - Gradient descent to some local error minimum (perhaps not global)
  - Possible improvements on Backprop (BP) (see later)
    - Momentum term (BP variant; slightly different weight update rule)
    - Stochastic gradient descent (BP variant)
    - Train multiple nets with different initial weights
  - Improvements on feedforward networks
    - Bayesian learning for ANNs
    - Other global optimization methods that integrate over multiple networks

Overtraining in ANNs

- **Recall: Definition of Overfitting**
  - *Performance of the model improves (converges) on* D_{train}, but worsens (or much worse) on D_{test}
- **Overtraining: A Type of Overfitting**
  - *Due to excessive iterations*



Error versus epochs (Example 1)



Error versus epochs (Example 2)

Overfitting in ANNs

- **Other Possible Causes of Overfitting**
  - Number of hidden units sometimes set in advance
  - Too many hidden units ("**overfitting**")
    - The network can memorise to a large extent the specifics of the training data
    - Analogy: fitting a quadratic polynomial with an approximator of degree >> 2
  - Too few hidden units ("**underfitting**")
    - ANNs with no growth
    - Analogy: underdetermined linear system of equations (more unknowns than equations)

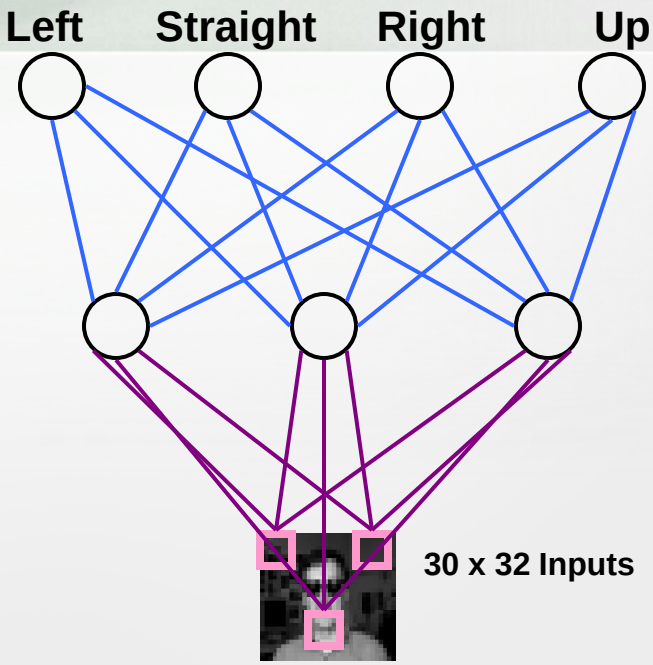
Overfitting in ANNs

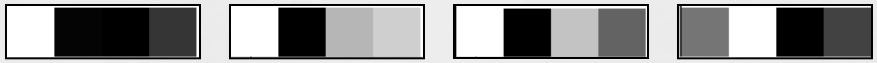
- **Solution Approaches**
  - Prevention: attribute subset selection
  - Avoidance
    - Hold out cross-validation (CV) set or split *k* ways (when to stop?)
    - Weight decay: decrease each weight by some factor on each epoch
  - Detection/recovery: random restarts, addition and deletion of units

Example: Neural Nets for Face Recognition
The Task (http://www.cs.cmu.edu/~tom/faces.html)

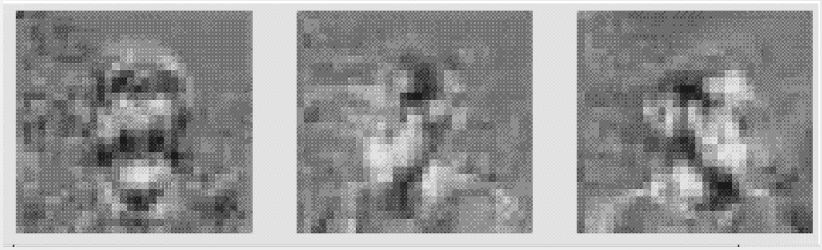
- **Learning Task**: Classifying Camera Images of faces of various people in various poses:
  - 20 people; 32 images per person (624 greyscale images, each with resolution 120 x 128, greyscale intensity 0 (black) to 255 (white))
  - varying expressions (happy, sad, angry, neutral)
  - Varying directions (looking left, right, straight ahead, up)
  - Wearing glasses or not
  - Variation in background behind the person
  - Clothing worn by the person
  - Position of face within image
- **Variety of target functions can be learned**
  - Id of person; direction; gender; wearing glasses; etc.
- **In this case**: learn direction in which person is facing

Neural Nets for Face Recognition

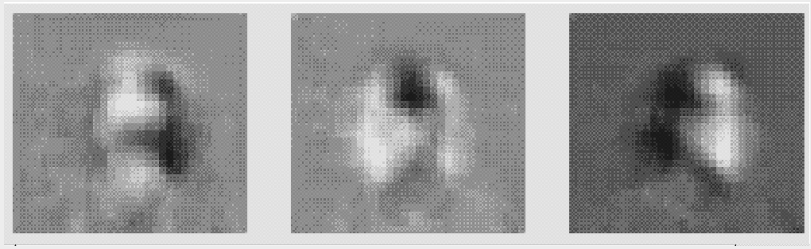




Output Layer Weights (including $w_0 = \theta$) after 1 Epoch



Hidden Layer Weights after 25 Epochs



Hidden Layer Weights after 1 Epoch

- **90% Accurate Learning Head Pose, Recognizing 1-of-20 Faces**

Neural Nets for Face Recognition: Design Choices

- **Input Encoding:**
  - How to encode an image?
    - Extract edges, regions of uniform intensity, other local features?
    - Problem: variable number of features ➔ variable # of input units
  - Choice: encode image as 30 x 32 pixel intensity values (summary/means of original 120 x 128) ➔ computational demands manageable
    - This is crucial in case of ALVINN (autonomous driving)
- **Output Encoding:**
  - ANN to output 1 of 4 values
    - Option1: single unit (values e.g. 0.2, 0.4, 0.6, 0.8)
    - Option2: 1-of-n output encoding (better option)
      - Note: Instead of 0 and 1 values, 0.1 and 0.9 are used (sigmoid units cannot output 0 and 1 given finite weights)

Neural Nets for Face Recognition: Design Choices

- **Network graph structure:** How many units to include and how to interconnect them
  - Commonly: 1 or 2 layers of sigmoid units (occasionally 3). If more, training becomes long!
  - How many hidden units? Fairly small preferable.
    - E.g. with 3 hidden units: 5 min. training; 90%
      with 30 hidden units: 1 hour training; barely better

- **Other Learning Algorithm Parameters:**
  - Learning rate: r = 0.3; momentum α = 0.3 (if too big, training fails to converge with acceptable error)
  - Full gradient descent used.

Neural Nets for Face Recognition: Design Choices

- Network weights
  - Initialised to small random values
- Input unit weights
  - Initialised to zero
- Number of training iterations
  - Data partitioned into training set and validation set
- Gradient descent used
  - Every 50 steps, network performance evaluated over the validation set
  - Final network: the one with highest accuracy over validation set
  - Final result (90%) measured over 3rd set of examples

Example: *NetTalk*

- **Sejnowski and Rosenberg, 1987**
- **Early Large-Scale Application of Backprop**
  - Learning to convert text to speech
    - Acquired model: *a mapping from letters to phonemes and stress marks*
    - Output passed to a speech synthesizer
  - Good performance after training on a vocabulary of ~1000 words
- **Very Sophisticated Input-Output Encoding**
  - Input: 7-letter window; determines the phoneme for the center letter and context on each side; distributed (i.e., sparse) representation: 200 bits
  - Output: units for articulatory modifiers (e.g., “voiced”), stress, closest phoneme; distributed representation
  - 40 hidden units; 10000 weights total
- **Experimental Results**
  - Vocabulary: trained on 1024 of 1463 (informal) and 1000 of 20000 (dictionary)
  - 78% on informal, ~60% on dictionary
- **http://www.boltz.cs.cmu.edu/benchmarks/nettalk.html**

Training (Learning) Functions

- During training: network weights and biases are iteratively adjusted to minimize its Error (/ Loss / performance) function
- Mean Squared Error: default error function for feedforward networks
- Different training algorithms for feedforward networks.
- All are based on the use of the gradient of the error function (to determine how to adjust the weights)
- Gradient determined using backpropagation
- Basic backpropagation training algorithm: weights are updated so as to move in the direction of the negative of the gradient

More on Backpropagation

- One iteration of backpropagation updates the weights as $w_{k+1} = w_k + \alpha_k g_k$
  where at iteration k
  - w_k : vector of weights and biases,
  - g_k : current gradient, and
  - α_k : learning rate.
- Two ways of implementing gradient descent :
  - **incremental mode**: gradient computed and weights updated after each input to the network.
  - **batch mode**: all the inputs are fed into the network before the weights are updated.

Variations of the Backpropagation Algorithm

Batch (Steepest) Gradient Descent training:

- Weights and biases updated in the direction of the negative gradient of the performance function
- The larger the learning rate, the bigger the step.
- If learning rate is set too large, then the algorithm becomes unstable.
- If the learning rate is set too small, the algorithm takes a long time to converge.
- Stopping conditions: max number of iterations, performance gets below the Goal, gradient magnitude smaller than a minimum, maximum training time reached, etc.

Faster Training

- **GD** often too slow for practical problems
- Several high-performance algorithms can converge from 10 to 100 times faster
- The faster algorithms fall into two categories:
  1. Those using heuristic techniques
  2. Those using standard numerical optimization techniques

Faster Algorithms using Heuristic Functions

- GD with momentum
- Adaptive learning rate backpropagation
- Resilient backpropagation

Faster Algorithms using Heuristic Functions

Gradient Descent with Momentum

- Makes the network to respond to the local gradient, **plus** the recent trends in the error surface

$$\Delta w_{start\text{-}layer,end\text{-}layer}(n) = r\,\delta_{end\text{-}layer}\,a_{end\text{-}layer} + \alpha \Delta w_{start\text{-}layer,end\text{-}layer}(n-1)$$

- Two training parameters:
  - Learning rate
  - Momentum:
    - value between 0 (no momentum) and close to 1 (lots of momentum).
    - If momentum = 1 , then network insensitive to the local gradient ➔ does not learn properly.

- Remains quite slow

Faster Algorithms using Heuristic Functions

Adaptive learning rate backpropagation

- With SGD, learning rate is kept constant throughout training. ➔ algorithm performance very sensitive to proper setting of learning rate.
- If *lr* is too large, the algorithm can oscillate and become unstable.
- If *lr* is too small, the algorithm takes too long to converge.
- Possible to improve the performance of SGD by allowing the learning rate to change during training.
- Adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable.
- Learning rate is made responsive to the complexity of the local error surface.

Adaptive learning rate backpropagation (cont.)

- If new error exceeds the old error by more than a predefined ratio (typically 1.04),
  - the new weights and biases are discarded.
  - The learning rate is decreased (typically multiplying by a factor of 0.7).

  Otherwise, the new weights and biases, are kept.
- If new error less than old error, the learning rate is increased (typically multiplying by a factor 1.05).
- A near-optimal learning rate can be obtained for the given problem
- One can combine Adaptive learning rate with Momentum

Faster Algorithms using Heuristic Functions

Resilient Backpropagation:

Problem with Sigmoid functions: their slopes approach zero as the input gets large

➔ the gradient can have a very small magnitude

➔ small changes in weights and biases, even though the weights and biases are far from their optimal values.

- Resilient backpropagation attempts to eliminate harmful effects of the magnitudes of the gradient
- Magnitude of the gradient has no effect on the weight update
- Only the sign of the gradient is used to determine the direction of the weight update;

Resilient Backpropagation (cont.)

- The size of weight change is determined by a separate update value.
- The update value for each weight and bias is increased by a factor *delt_inc* whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations.
- The update value is decreased by a factor *delt_dec* whenever the derivative with respect to that weight changes sign from the previous iteration.
- If the derivative is zero, then the update value remains the same.
- So: Whenever the weights are oscillating, the weight change is reduced.
- If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change increases.

Resilient Backpropagation (cont.)

- Generally much faster than the standard steepest descent algorithm.
- Requires only a modest increase in memory requirements

Faster Algorithms using Standard Numerical Optimization Techniques

- **Conjugate Gradient Algorithms**
- Basic Backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient), the direction in which the performance function is decreasing most rapidly.
- This does not necessarily produce the fastest convergence.
- In **conjugate gradient algorithms** a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions.
- In most of conjugate gradient algorithms, the step size is adjusted at each iteration.
- A search is made along the conjugate gradient direction to determine the step size that minimizes the performance function along that line

Conjugate Gradient Algorithms

- Fletcher-Reeves Update
- Polak-Ribiére Update
- Powell-Beale Restarts
- Scaled Conjugate Gradient

Conjugate Gradient Algorithms

- All the CG algorithms start by searching in the SGD direction on the first iteration.

$$P_o = - g_o$$

- A line search is then performed to determine the optimal distance to move along the current search direction:

$$x_{k+1} = x_k + \alpha_k P_k$$

- The next search direction is determined so that it is conjugate to previous search directions.
- The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$p_k = - g_k + \beta_k p_{k-1}$$

Conjugate Gradient Algorithms

- The various versions of the conjugate gradient algorithm are distinguished by the way in which the constant β_k is computed.
- For the **Fletcher-Reeves** update the procedure is

$$\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$$

- For the **Polak-Ribiére** update, the constant k is computed by

$$\beta_k = \frac{\Delta g_{k-1}^T g_k}{g_{k-1}^T g_{k-1}}$$

- Etc.

Faster Algorithms using Standard Numerical Optimization Techniques

- **Quasi-Newton Algorithms**
- Basic step of Newton's method is

$$X_{k+1} = -X_k + A_k^{-1} g_k$$

where A_k^{-1} is the Hessian matrix (second derivatives) of the performance.

- Newton's method often converges faster than conjugate gradient methods.
- Computing the Hessian matrix for feedforward neural networks is complex and expensive.
- A class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives.
- These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm.

Some ANN Applications

- **Diagnosis**
  - Closest to pure concept learning and classification
  - Some ANNs can be post-processed to produce probabilistic diagnoses
- **Prediction and Monitoring**
  - *aka* prognosis (sometimes forecasting)
  - Predict a continuation of (typically numerical) data
- **Decision Support Systems**
  - *aka* recommender systems
  - Provide assistance to human "subject matter" experts in making decisions
    - Design (manufacturing, engineering)
    - Therapy (medicine)
    - Crisis management (medical, economic, military, computer security)
- **Control Automation**
  - Mobile robots
  - Autonomic sensors and actuators
- **Many, Many More**

Strengths of a Neural Network

- **Power**: Model complex functions, nonlinearity built into the network
- **Ease of use**:
  - Learn by example
  - Very little user domain-specific expertise needed
- **Intuitively appealing**: based on model of biology, will it lead to genuinely intelligent computers/robots?

General Advantages / Disadvantages

- **Advantages**
  - Adapt to unknown situations
  - Robustness: fault tolerance due to network redundancy
  - Autonomous learning and generalization

- **Disadvantages**
  - Complexity of finding the “right” network structure
  - “Black box”

Issues and Open Problems in (classical) ANN Research

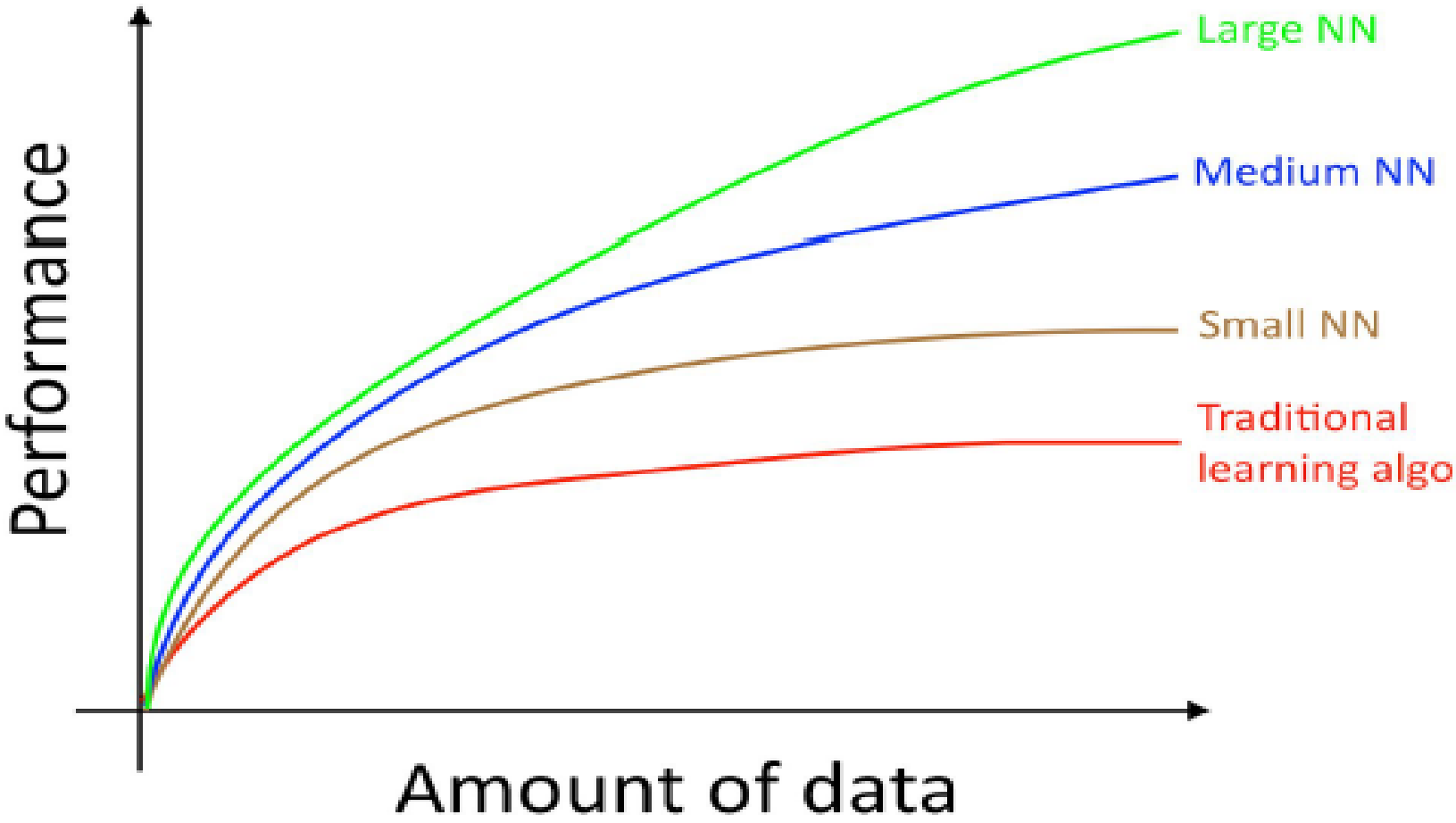
- **Hybrid Approaches**
  - Incorporating knowledge and analytical learning into ANNs
    - Knowledge-based neural networks
    - Explanation-based neural networks
    - Combining uncertain reasoning and ANN learning and inference
    - Probabilistic ANNs
    - Bayesian networks

Issues and Open Problems in (classical) ANN Research

- **Global Optimization with ANNs**
  - Hybrid models
  - Relationship to genetic algorithms
- **Understanding ANN Output**
  - Knowledge extraction from ANNs
    - Rule extraction
    - Other decision surfaces
  - Decision support and KDD applications
- **Many More Issues (Robust Reasoning, Representations, etc.)**

Going Deep?

Andrew Ng, Machine Learning Yearning, --Technical Strategy for AI Engineers in the Era of Deep Learning, Draft Version, 2018, deeplearning.ai



This presentation was based on the following:

- *Tom Mitchell*, *Machine Learning*, McGraw Hill, 1997
- Lecture notes by Prof. Hsu (Kansas State University) based on Tom Mitchell's Machine Learning book.
- Training Functions and Fast Training, Matlab documentation, Mathworks.
- Andrew Ng, Machine Learning and Deep Learning online course, Coursera.
- Dave Touretzki, « Artificial Neural Networks » Lecture notes, Carnegie Mellon University, http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15782-f06/syllabus.html
- Nicolas Galoppo von Borries, Introduction to Neural Networks, COMP290-058 Motion Planning. (http://slideplayer.com/slide/9882811/)

Thank you!