STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY

CPE/EE 695: Applied Machine Learning

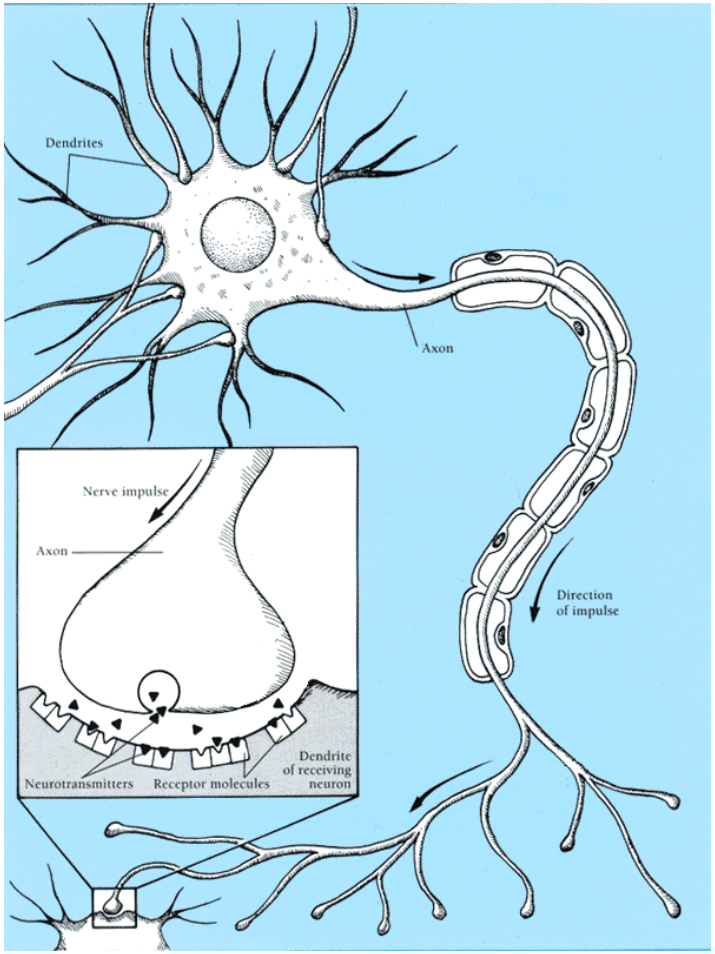
Lecture 7: Artificial Neural Networks

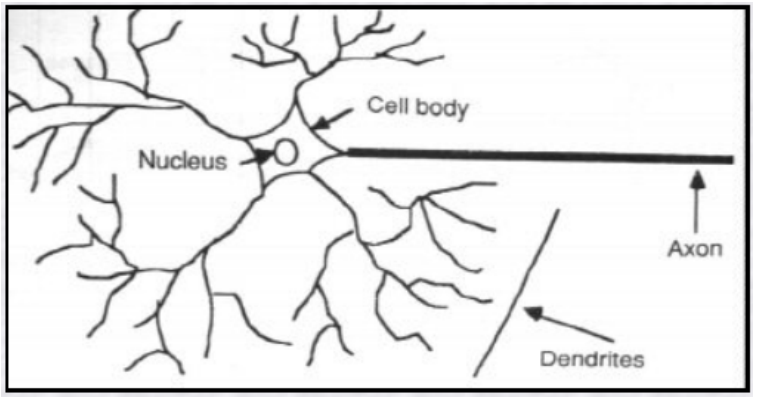
Dr. Shucheng Yu, Associate Professor
Department of Electrical and Computer Engineering
Stevens Institute of Technology

What are artificial neural networks?

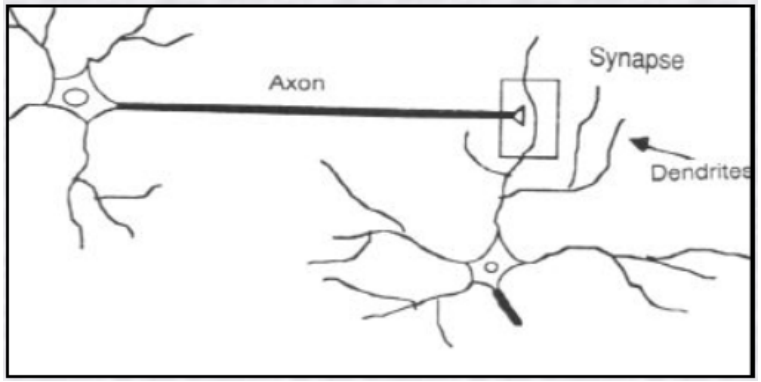
Biological inspiration

How human brain works?





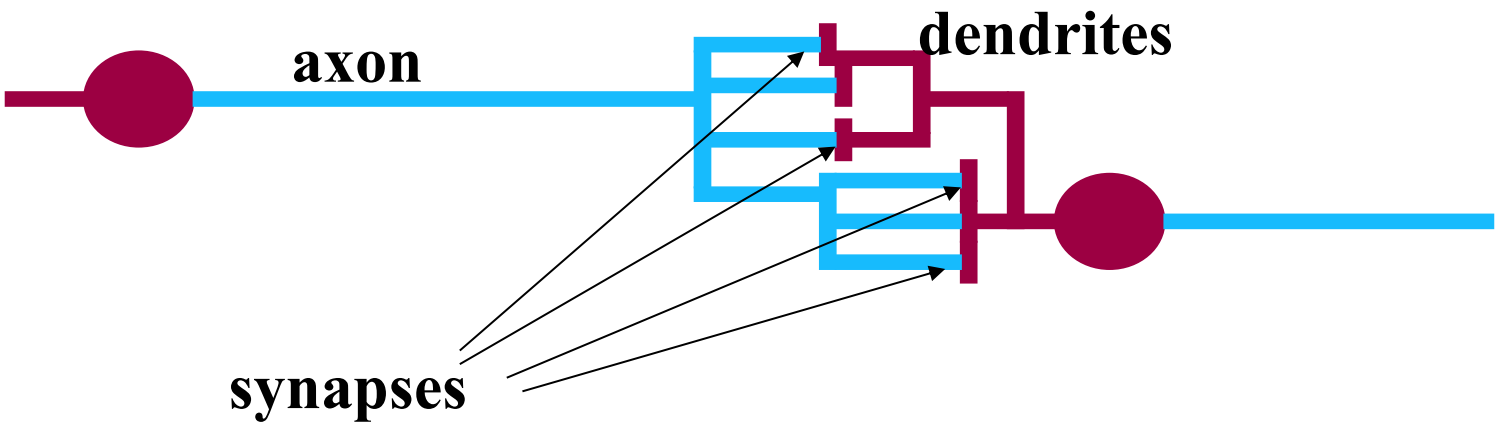
Components of a neuron



Components of a synapse

Neural Communication

- Electrical potential across cell membrane exhibits spikes called ***action potentials***.
- Spike originates in cell body, travels down axon, and causes synaptic terminals to release ***neurotransmitters***.
- Chemical diffuses across synapse to dendrites of other neurons.
- Neurotransmitters can be ***excitatory*** or ***inhibitory***.
- If net input of neurotransmitters to a neuron from other neurons is excitatory and exceeds some threshold, it fires an action.



Neural Speed Constraints

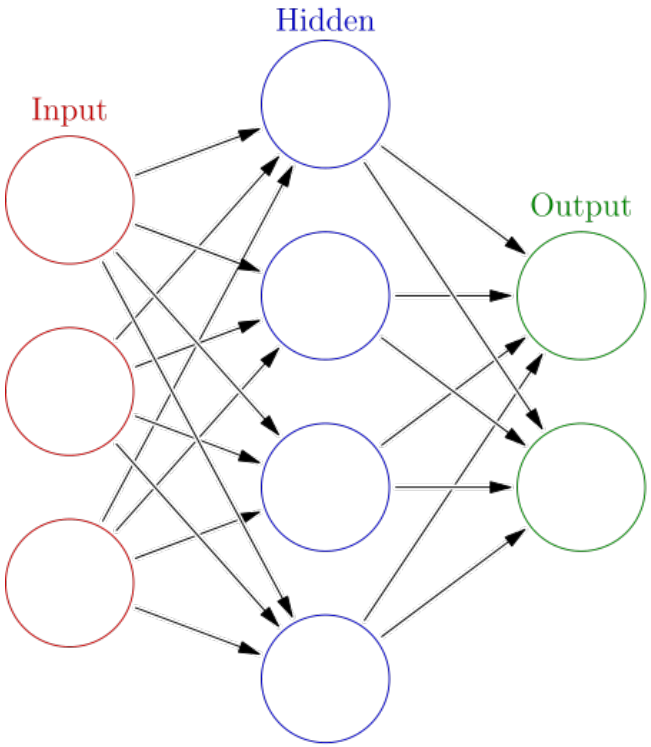
- Neurons have a “switching time” on the order of a few **milliseconds**, compared to **nanoseconds** for current computing hardware.
- However, neural systems can perform complex cognitive tasks (vision, speech understanding) in **tenths of a second**.
- Only time for performing 100 serial steps in this time frame, compared to orders of magnitude more for current computers.
- Must be exploiting “**massive parallelism**.”
- Human brain has about **10^{11} neurons** with an average of 10^4 connections each.

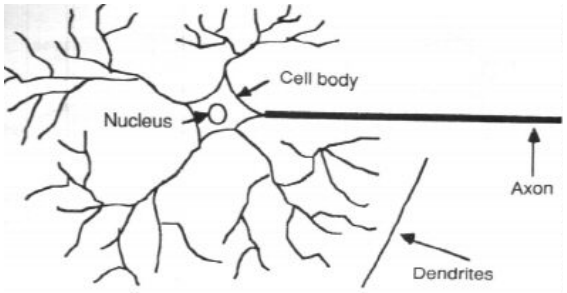
Real Neural Learning

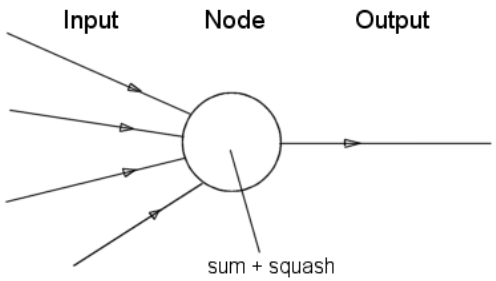
- Synapses change size and strength with experience.
- **Hebbian learning**: “When neuron A repeatedly participates in firing neuron B, the strength of the action of A onto B increases”
- “Neurons that fire together, wire together.”

What are artificial neural networks?

Connectionist systems inspired by the biological neural networks that constitute animal brains, learning (progressively improve performance) to do tasks by considering examples, generally without task-specific programming.







ANNs: goal and design

- Knowledge about the learning task is given in the form of a set of examples (training examples)
- An ANN is specified by:

  architecture: a set of neurons and weighted links connecting neurons.

  neuron model: the information processing unit of the ANN,

  learning algorithm: used for training the ANN by modifying the weights in order to solve the particular learning task correctly on the training examples.

The aim is to obtain an ANN that can be generalized well.

ANN application….

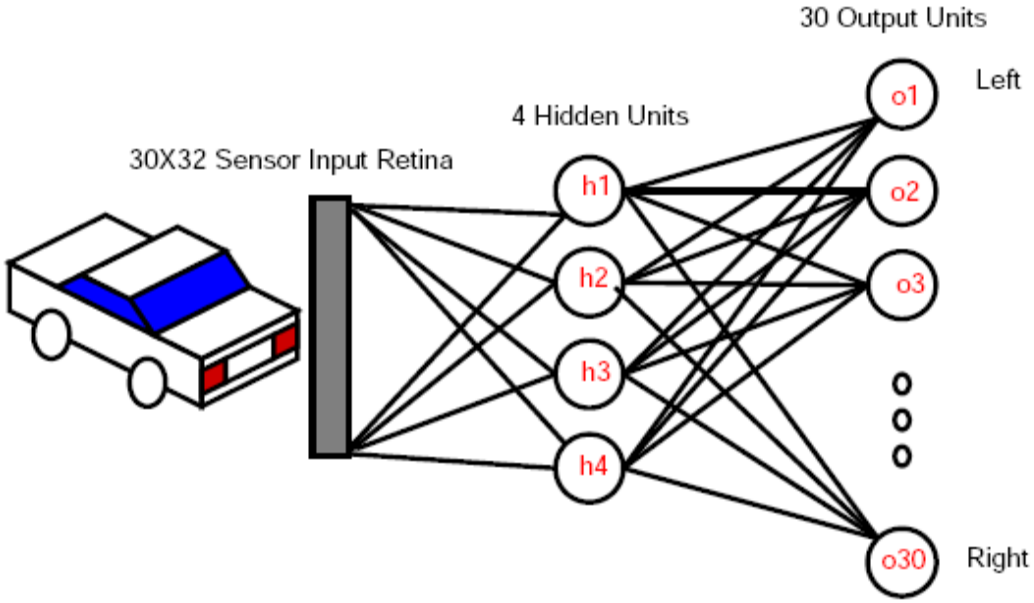
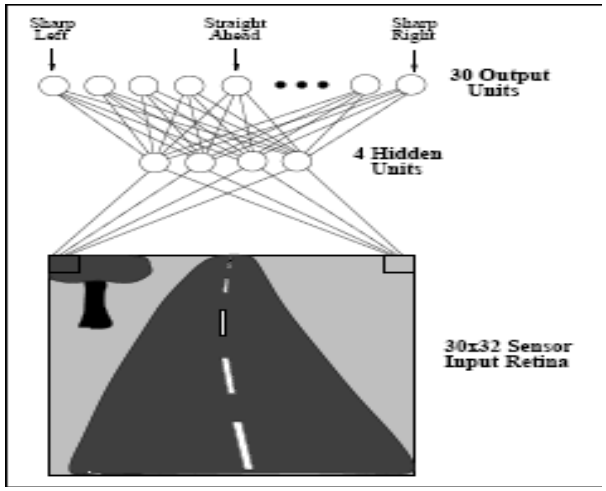


Figure 1: Main Structure



ALVINN [Pomerleau'93] drives 70 mph in highway

When to consider neural network models

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Forms of target function is unknown
- Human readability of result is unimportant
- Examples:
  - Speech recognition
  - Image classification
  - Financial prediction

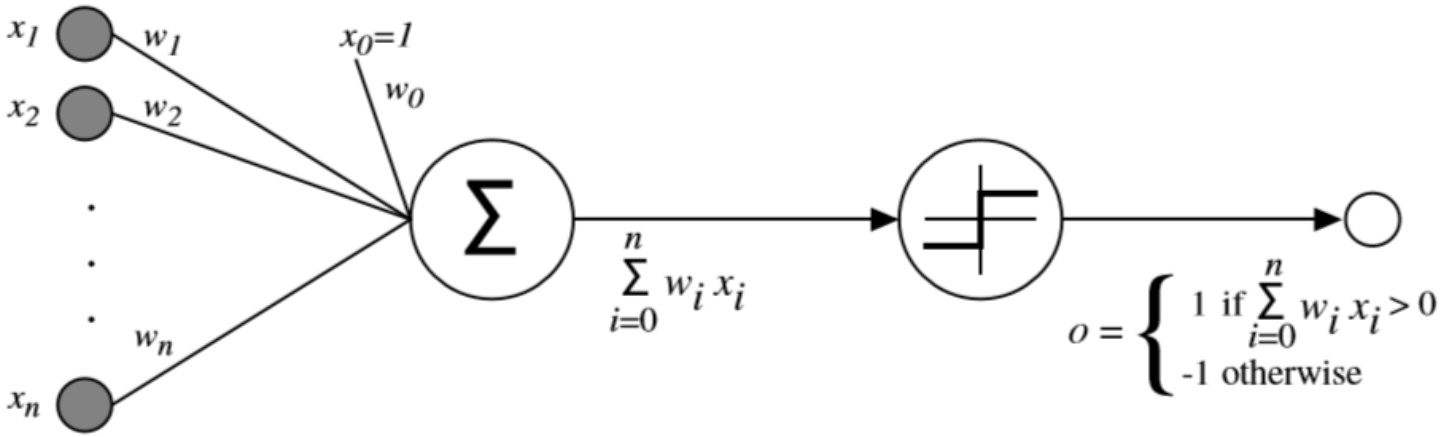
Neural Network Learning

Learning approach based on modeling adaptation in biological neural systems.

- **Perceptron**: Initial algorithm for learning simple neural networks (single layer) developed in the 1950's.
- **Backpropagation**: More complex algorithm for learning multi-layer neural networks developed in the 1980's.

Perceptron

W. McCulloch and W. Pitts (1943)



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

Sometimes we'll use simpler vector notation:

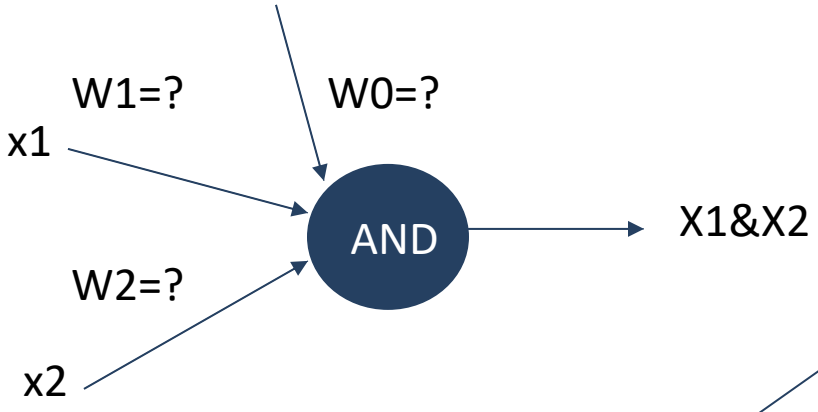
$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Representational power of perceptron

A single perceptron can be used to represent many Boolean functions

What weights represent "AND" function?

$g(x_1,x_2)$= AND(x_1,x_2)?



X1	X2	X1 & X2
0	0	0
0	1	0
1	0	0
1	1	1

w0 + w1*0 +w2*0 <=0
w0 + w1*0 +w2*1 <=0
w0 + w1*1 +w2*0 <=0
w0 + w1*1 +w2*1 >0

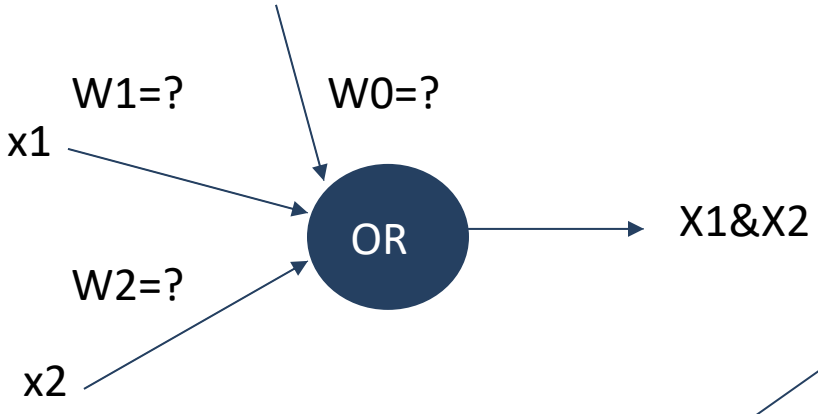
w0=-0.8, w1=w2=0.5

Representational power of perceptron

A single perceptron can be used to represent many Boolean functions

What weights represent "OR" function?

$g(x_1,x_2)$= OR(x_1,x_2)?



X1	X2	X1 || X2
0	0	0
0	1	1
1	0	1
1	1	1

w0 + w1*0 +w2*0 <=0
w0 + w1*0 +w2*1 >=0
w0 + w1*1 +w2*0 >=0
w0 + w1*1 +w2*1 >0

w0=-0.3, w1=w2=0.5

Representational power of perceptron

A single perceptron can be used to represent many Boolean functions

All of the primitive Boolean functions **AND, OR, NAND (¬AND), NOR (¬OR)** can be represented by perceptrons.

Representational power of perceptron

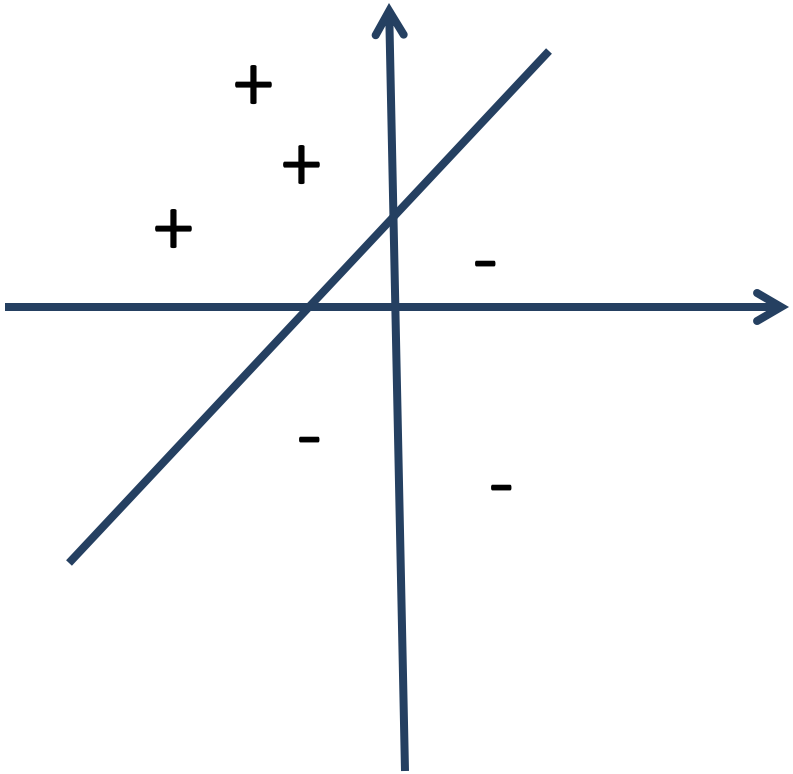
A single perceptron can be used to represent many Boolean functions

All of the primitive Boolean functions **AND, OR, NAND (¬AND), NOR (¬OR)** can be represented by perceptrons.

Any boolean function can be implemented by using a (2-level) combination of these primitives (functional completeness property)!

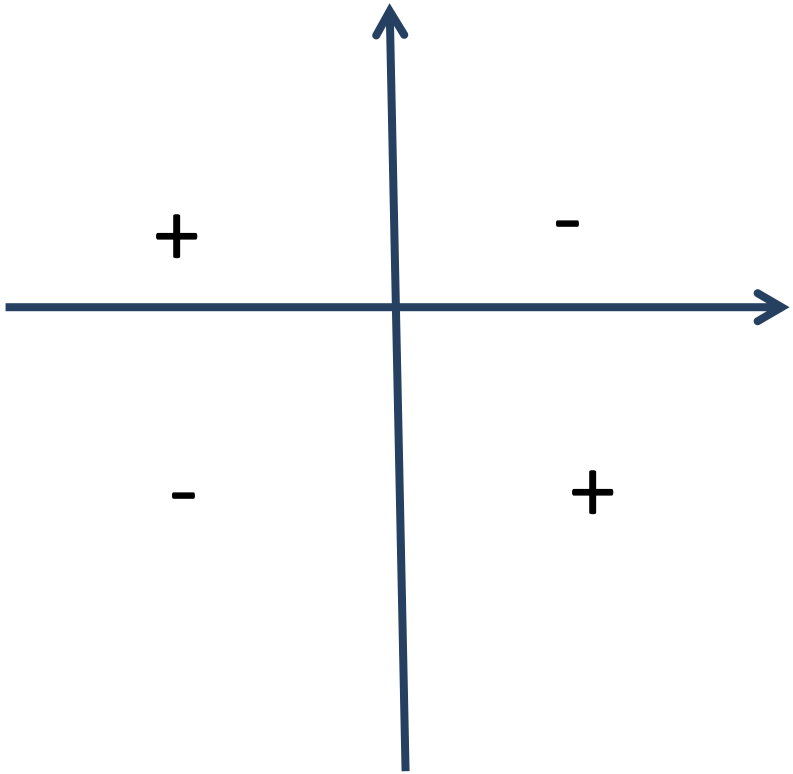
Decision surface of a perceptron

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional spaces of instances (i.e. points).



Decision surface of a perceptron

Single perceptrons can represent all of the primitive Boolean functions AND, OR, NAND, NOR, But fail on some functions (e.g., not linearly separable) like XOR



Not linearly separable!

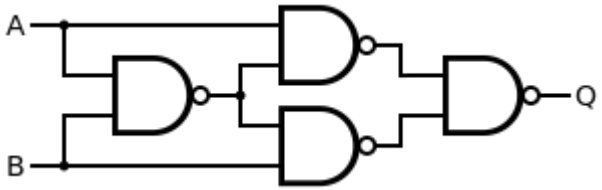
Representational power of perceptron

A single perceptron can be used to represent many Boolean functions

All of the primitive Boolean functions **AND, OR, NAND (¬AND), NOR (¬OR)** can be represented by perceptrons.

Some Boolean functions such as **XOR** can not be represented by single perceptrons (we need a network of them)

A XOR B = (¬A AND B) OR (A AND ¬B)



Perceptron training rule

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

where

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

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Perceptron training rule

Can prove it will converge

- If training data is linearly separable
- and η sufficiently small

Linear separability can be verified using several approaches (e.g., Linear Programming).

Let's say we have two sets A and B of points in $\mathbb{R}^M$:

$$A = \{a_1, ..., a_{N_1}\} \subset \mathbb{R}^M, B = \{b_1, ..., b_{N_2}\} \subset \mathbb{R}^M$$

And we want to know if there is a hyper-plane in $\mathbb{R}^M$ which separates A and B then we can formulate the necessary condition with two symmetrical inequalities:

A $\beta \in \mathbb{R}$ and an $h \in \mathbb{R}^M$ exist, such that we can say for all $a \in A : h^T a > \beta$ (1) and for all $b \in B : h^T b < \beta$ (2).

Gradient descent and delta rule

(least-mean-square (LMS) rule)

To understand, consider simpler *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

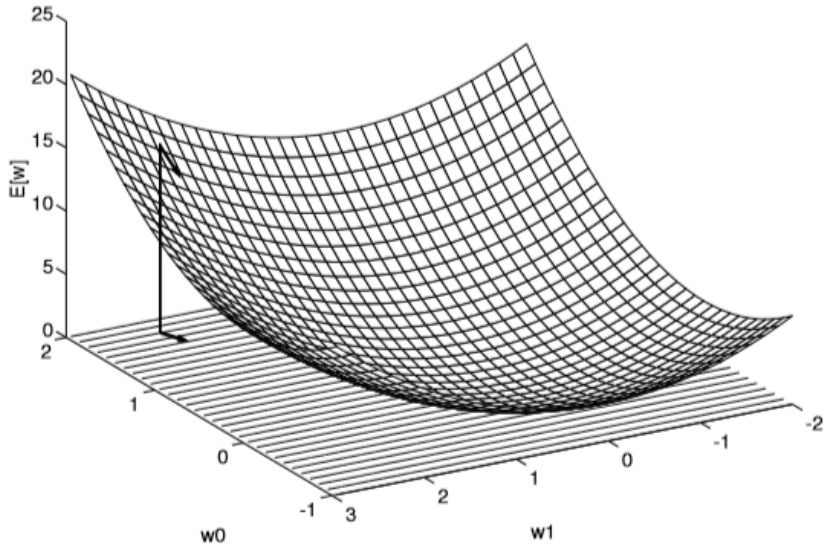
Unthresholded perceptron

Let's learn w_i's that minimize the squared error

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

Where D is set of training examples

Gradient descent



Gradient

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

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient descent

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

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

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

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

  - For each linear unit weight w_i, Do

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

Summary

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Stochastic approximation to gradient descent

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Key differences between standard gradient descent and stochastic gradient descent

- Standard gradient descent:
  - the error is summed over all examples before updating weights,
  - requires more computation per weight update step.

- Stochastic gradient descent
  - weights are updated upon examining each training example;
  - sometimes avoid falling into local minima, if there are multiple local minima, (why?)

<u>Both methods are commonly used in practice</u>

perceptron training rule and delta rule

perceptron training rule:

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

$$o(\vec{x}) = \begin{cases} 1 \text{ if } \vec{w} \cdot \vec{x} > 0 \\ -1 \text{ otherwise.} \end{cases}$$

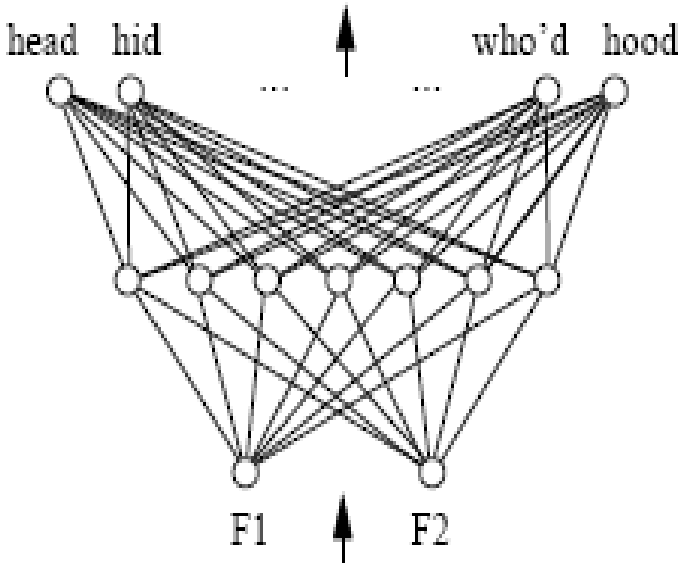
Delta rule:

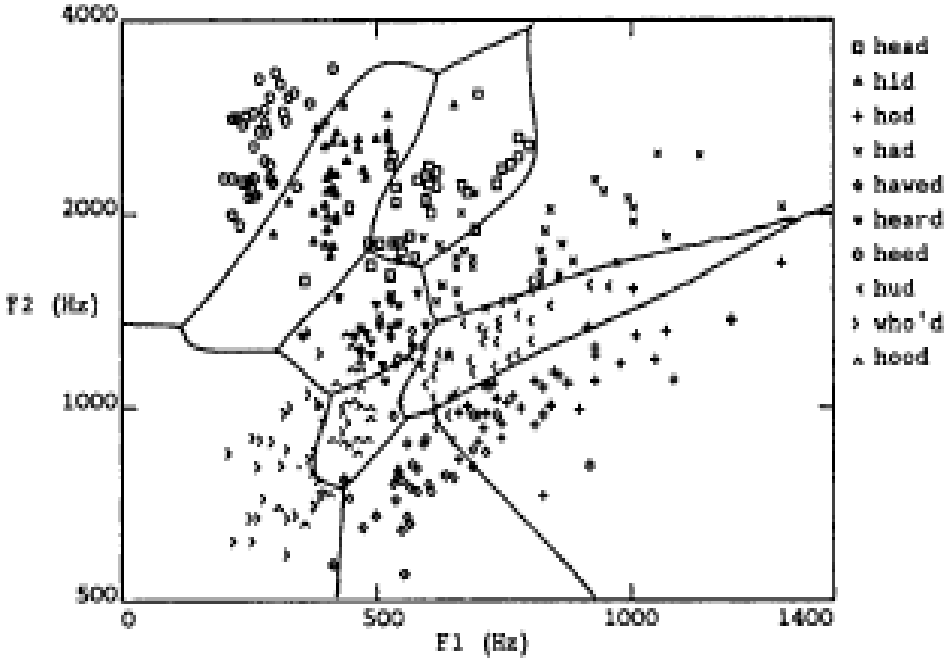
$$\Delta w_i = \eta(t - o)x_i$$

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

The definition of output *o* is different!
Perceptron rule updates weights based on the error in the *thresholded* perceptron output, whereas delta rule updates weights based on the error in the *unthresholded linear combination of inputs*

Is linear decision surface enough?





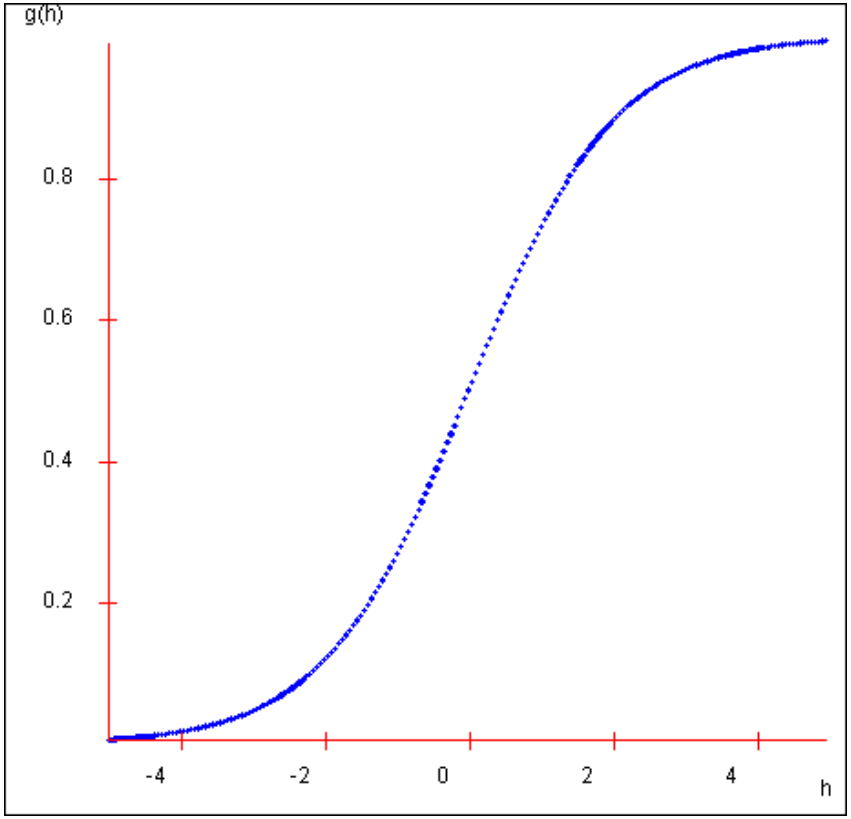
Multilayer cascaded linear units still produce only linear functions.

Multilayer perceptron units can represent highly nonlinear decision surface, but undifferentiable

The first question: a differentiable threshold unit

$$g(h) = \frac{1}{1+\exp(-h)}$$

sigmoid unit (logistic function, squashing function)

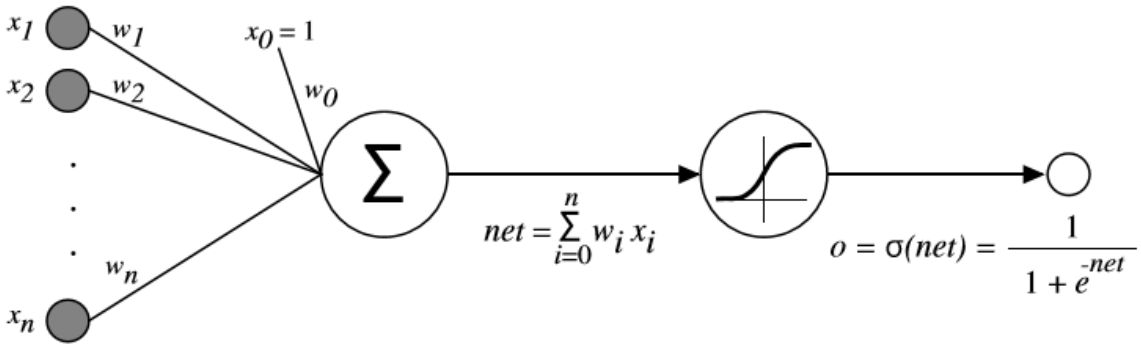


Note that if you rotate this curve through 180° centered on *(0,1/2)* you get the same curve.

i.e. *g(h)=1-g(-h)*

- Very much like a perceptron, but based on smoothed, differentiable threshold.
- Other differentiable functions with easily calculated derivatives are sometimes used.

Sigmoid unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Error gradient for a sigmoid unit

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

But we know:

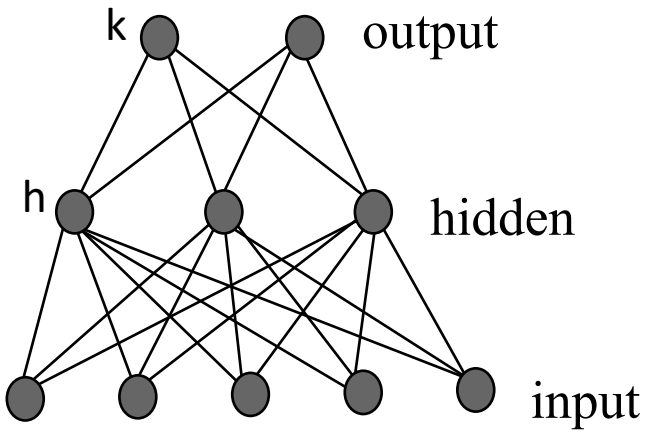
$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Backpropagation algorithm



Initialize all weights to small random numbers.
Until satisfied, Do

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit k

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

  3. For each hidden unit h

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

  4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

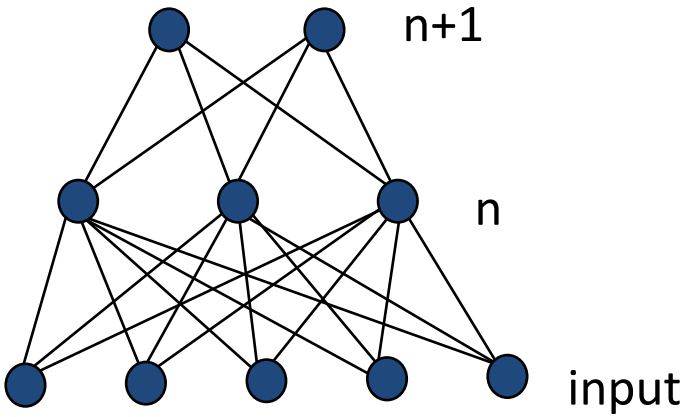
where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

Derivation of the Backpropagation algorithm

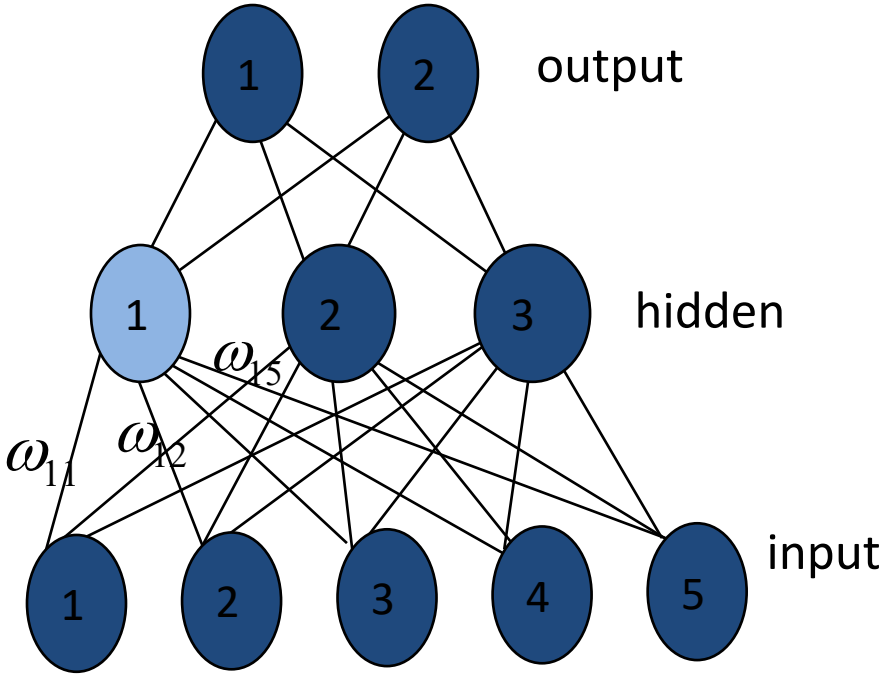
1. Propagates inputs forward in the usual way, i.e.
   - All outputs are computed using sigmoid thresholding of the inner product of the corresponding weight and input vectors.
   - All outputs at stage *n* are connected to all the inputs at stage *n*+1

2. Propagates the errors backwards by apportioning them to each unit according to the amount of this error the unit is responsible for.



Derivation of the Backpropagation algorithm

- $\vec{x}_j =$ input vector for unit j (x_{ji} = ith input to the jth unit)
- $\vec{w}_j =$ weight vector for unit j (w_{ji} = weight on x_{ji})
- $z_j = \vec{w}_j \cdot \vec{x}_j$, the weighted sum of inputs for unit j
- o_j = output of unit j ($o_j = \sigma(z_j)$)
- t_j = target for unit j
- *Downstream*(j) = set of units whose immediate inputs include the output of j
- *Outputs* = set of output units in the final layer

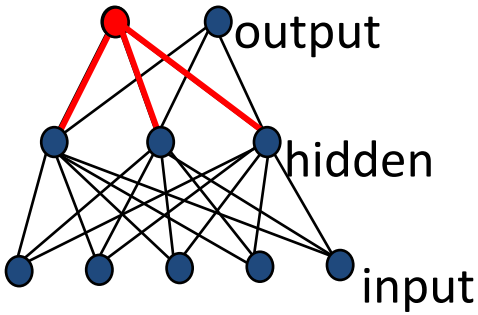


$$\vec{x}_1 =< x_{11} x_{12} x_{13} x_{14} x_{15} >$$

Derivation of the Backpropagation algorithm

For *output* units

calculate $\frac{\partial E}{\partial w_{ji}}$ for each input weight w_{ji} for each output unit j. Note first that since z_j is a function of w_{ji}



$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}}$$
$$= \frac{\partial E}{\partial z_j} x_{ji}$$

$$E = \frac{1}{2} \sum_{k \in Outputs} (t_k - \sigma(z_k))^2$$

the outputs of all units $k \neq j$ are independent of w_{ji}, we can drop the summation and consider just the contribution to E by j.

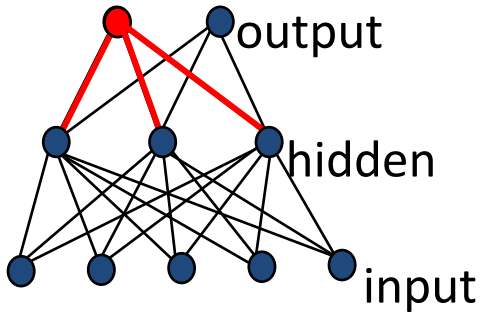
$$\delta_j = \frac{\partial E}{\partial z_j} = \frac{\partial}{\partial z_j} \frac{1}{2} (t_j - o_j))^2$$
$$= -(t_j - o_j) \frac{\partial o_j}{\partial z_j}$$
$$= -(t_j - o_j) \frac{\partial}{\partial z_j} \sigma(z_j)$$
$$= -(t_j - o_j)(1 - \sigma(z_j))\sigma(z_j)$$
$$= -(t_j - o_j)(1 - o_j)o_j$$

Derivation of the Backpropagation algorithm

For *output* units

So:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_i j} = \eta \delta_j x_{ji}$$

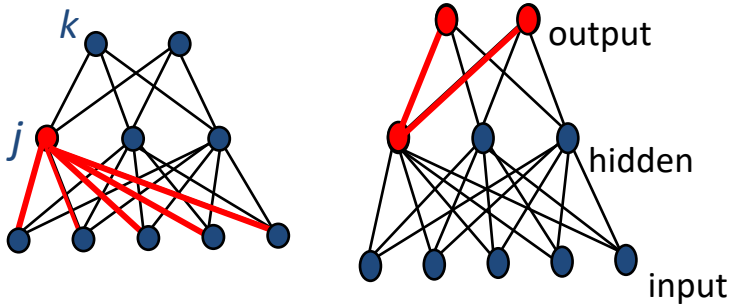


$$\begin{aligned}
\delta_j = \frac{\partial E}{\partial z_j} &= \frac{\partial}{\partial z_j} \frac{1}{2}(t_j - o_j))^2 \\
&= -(t_j - o_j)\frac{\partial o_j}{\partial z_j} \\
&= -(t_j - o_j)\frac{\partial}{\partial z_j}\sigma(z_j) \\
&= -(t_j - o_j)(1 - \sigma(z_j))\sigma(z_j) \\
&= -(t_j - o_j)(1 - o_j)o_j
\end{aligned}$$

Derivation of the Backpropagation algorithm

For *Hidden* units

We want to calculate $\frac{\partial E}{\partial w_{ji}}$ for each input weight w_{ji} for each hidden unit j. Note that w_{ji} influences just z_j which influences o_j which influences $z_k \forall k \in Downstream(j)$ each of which influence E. So we can write



$$\frac{\partial E}{\partial w_{ji}} = \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}}$$

$$= \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j} \cdot x_{ji}$$

Again note that all the terms except x_{ji} in the above product are the same regardless of which input weight of unit j we are trying to update.

Also: $\frac{\partial E}{\partial z_k} = \delta_k$, $\frac{\partial z_k}{\partial o_j} = w_{kj}$ and $\frac{\partial o_j}{\partial z_j} = o_j(1 - o_j)$.

So:
$$\delta_j = \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j}$$

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

More on Backpropagation algorithm

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

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

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

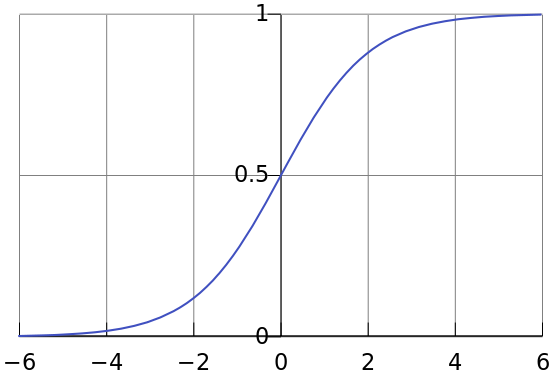
Convergence of backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses



Expressive capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

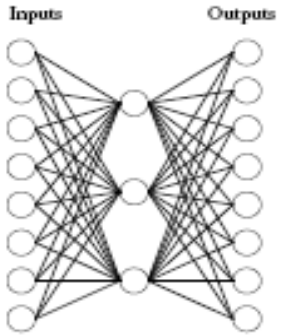
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Hidden Layer Representations

- Trained hidden units can be seen as ***newly constructed features*** that make the target concept linearly separable in the transformed space.
- On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..
- However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.

Learning Hidden layer representation



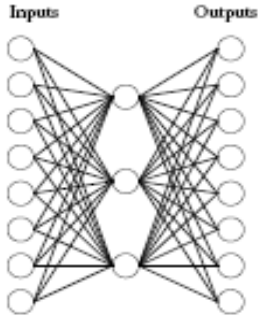
A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Learning Hidden layer representation

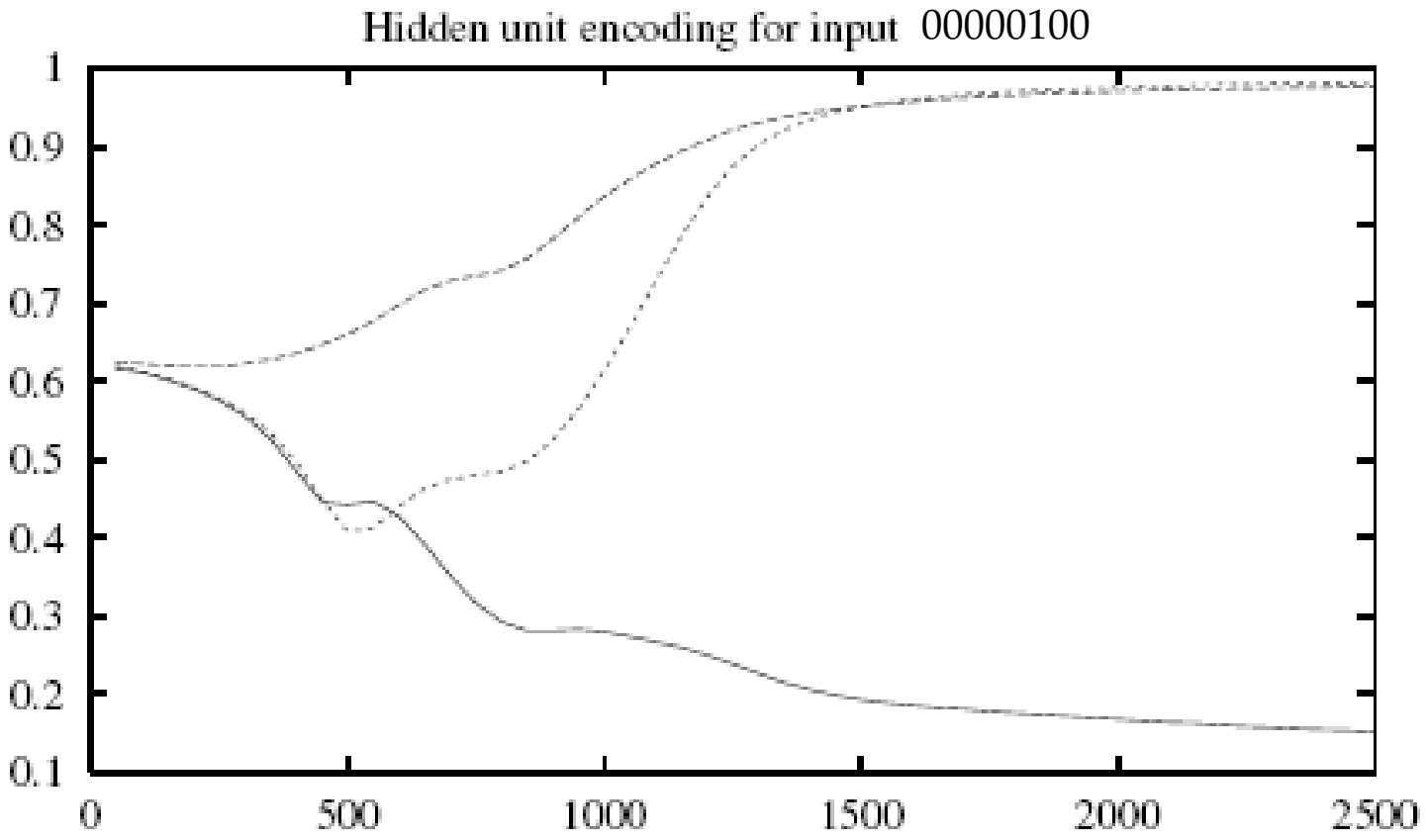
A network:



Learned hidden layer representation:

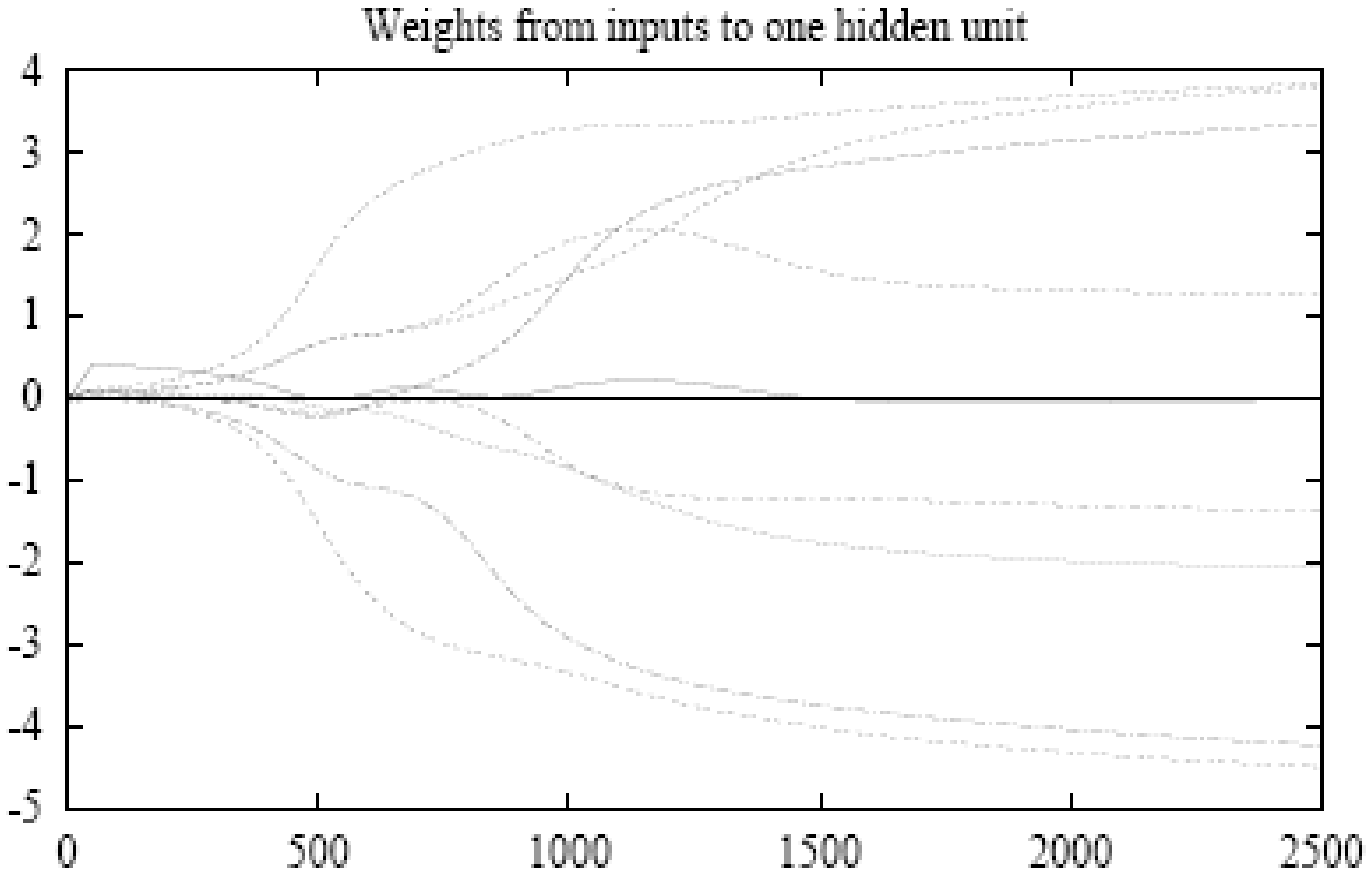
Input		Hidden Values				Output	
10000000	→	.89	.04	.08	→	10000000	100
01000000	→	.01	.11	.88	→	01000000	001
00100000	→	.01	.97	.27	→	00100000	010
00010000	→	.99	.97	.71	→	00010000	111
00001000	→	.03	.05	.02	→	00001000	000
00000100	→	.22	.99	.99	→	00000100	011
00000010	→	.80	.01	.98	→	00000010	101
00000001	→	.60	.94	.01	→	00000001	110

Evolution of the hidden layer representation

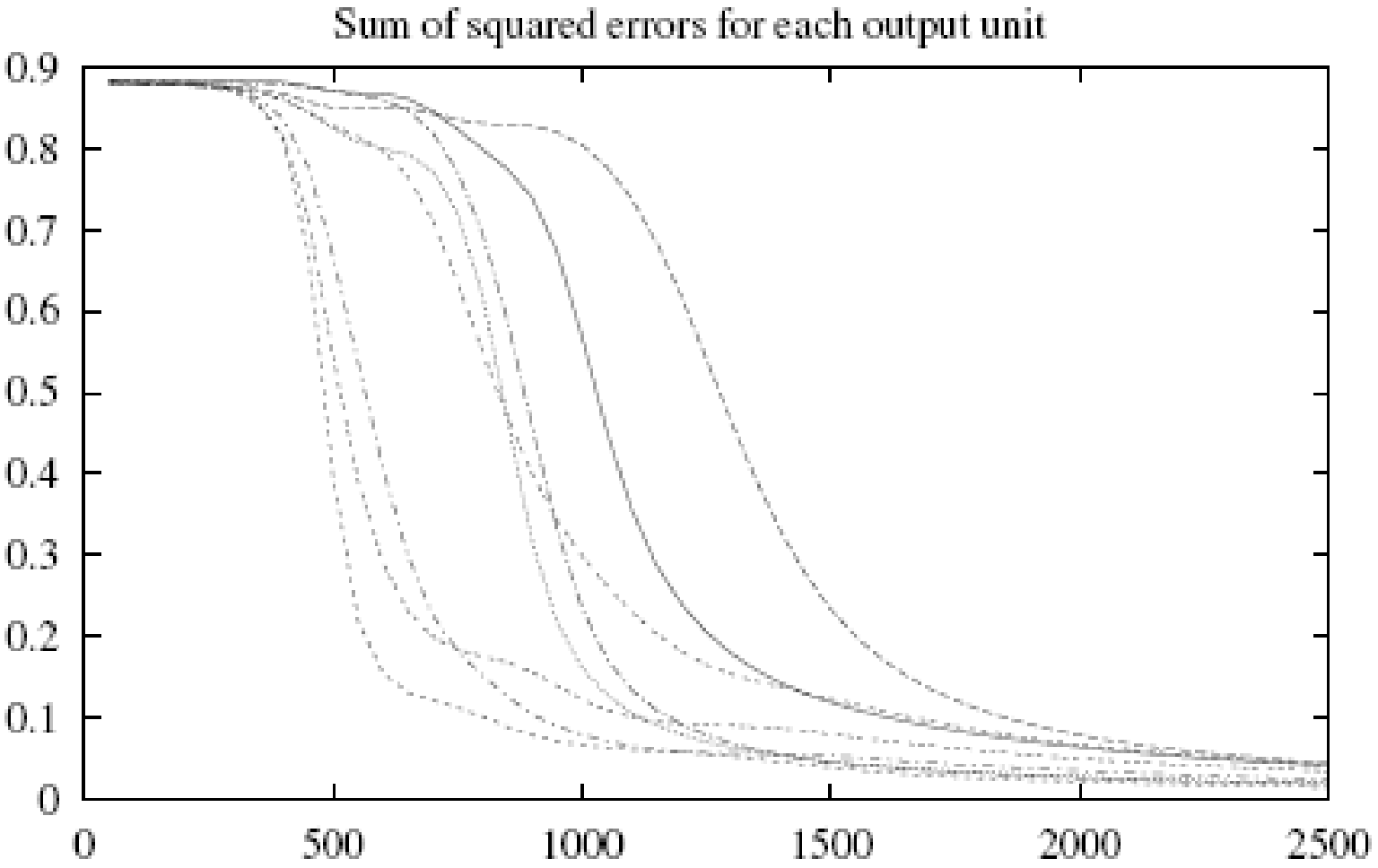


Three hidden unit values for one of the possible inputs

Weights from inputs to one hidden unit

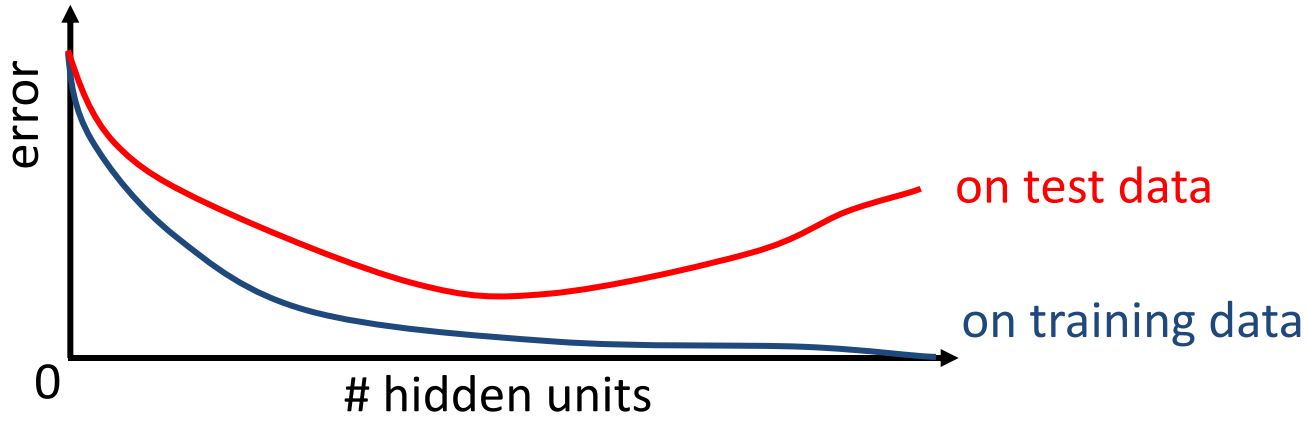


Sum of squared errors for each output unit



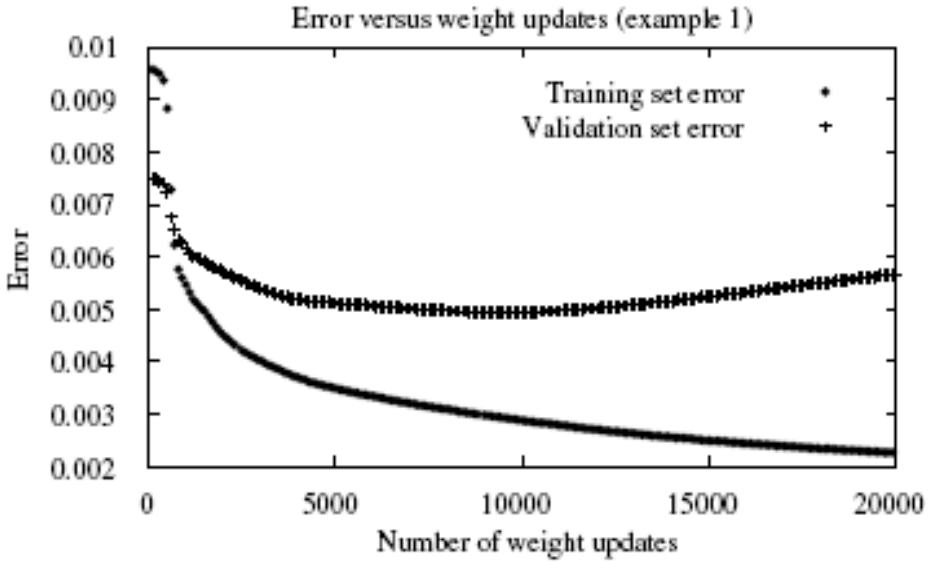
Determining the Best Number of Hidden Units

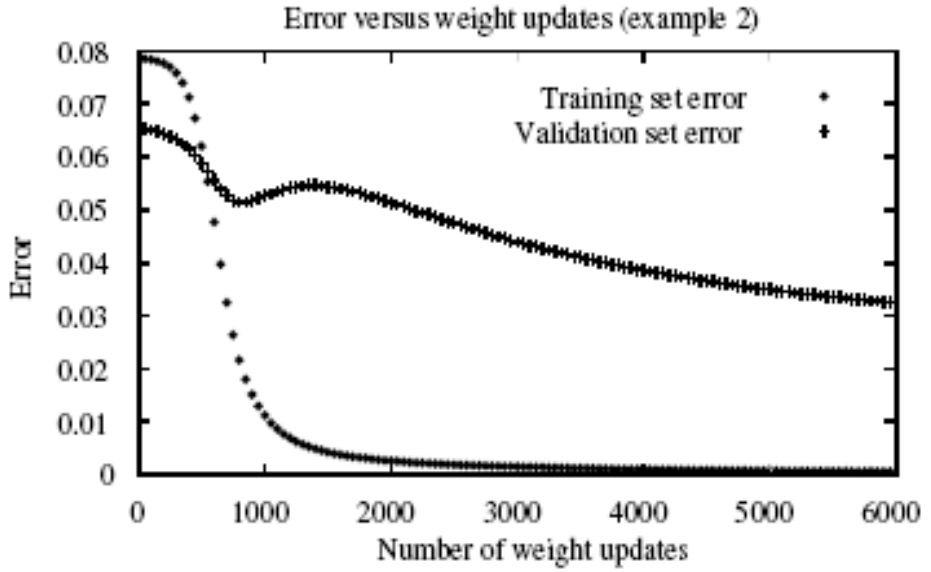
- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.



- Start small and increase the number until satisfactory results are obtained
- Use internal cross-validation to empirically determine an optimal number of hidden units.

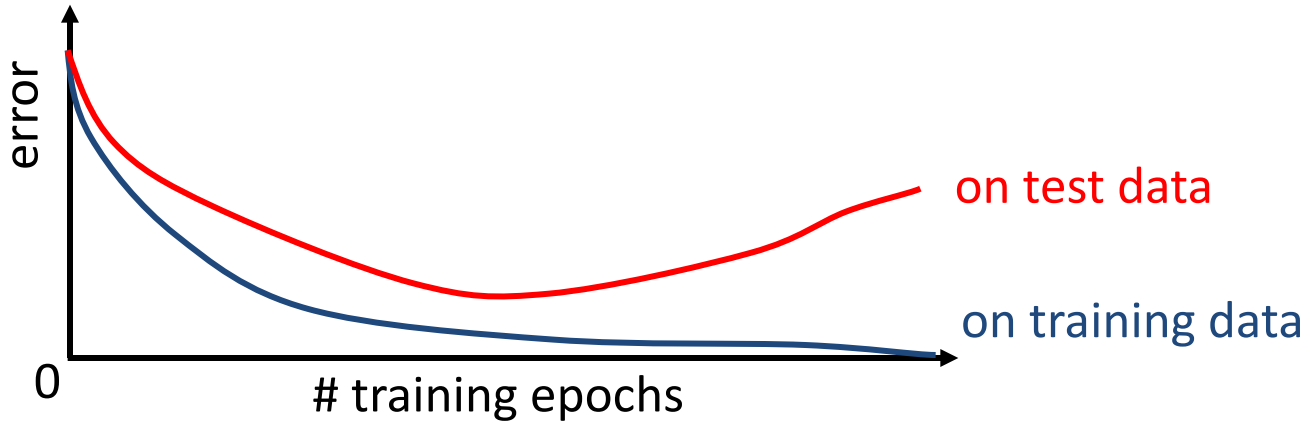
Overfitting of ANNs





Over-fitting Prevention

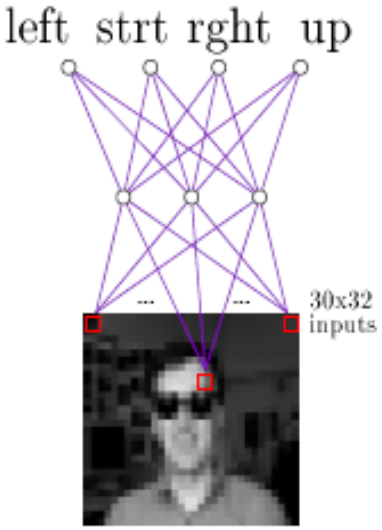
- Running too many epochs can result in over-fitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.
- To avoid losing training data for validation:
  - Use internal K-fold CV on the training set to compute the average number of epochs that maximizes **generalization accuracy**.
  - Train final network on complete training set for this many epochs.

Example: NN for face recognition

20 different people and 32 images each with resolution 120x128

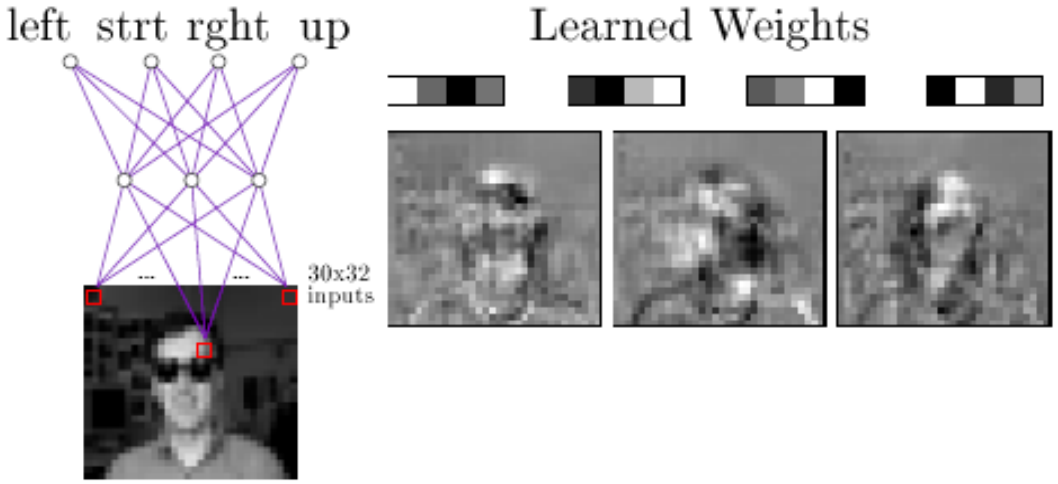


Typical input images

Learned hidden unit weights

Problems to consider:

- Input preprocessing
- Network Structure
- Output encoding
- Learning method



Typical input images

http://www.cs.cmu.edu/~tom/faces.html

Input Data Preprocessing

- The curse of Dimensionality
  - The quantity of training data grows exponentially with the dimension of the input space
  - In practice, we only have limited quantity of input data

    Increasing the dimensionality of the problem leads to giving a poor representation of the mapping

Preprocessing methods

- Normalization
  - Translate input values so that they can be exploitable by the neural network
- Component reduction
  - Build new input variables in order to reduce their number
  - No lost of information about their distribution

Normalization

- Inputs of the neural net are often of different types with different orders of magnitude (E.g. Pressure, Temperature, etc.)
- It is necessary to normalize the data so that they have the same impact on the model (i.e., same range as hidden unit and output activations).
- Center and reduce the variables

Normalization

$$\overline{x_i} = \frac{1}{N}\sum_{n=1}^{N} x_i^n$$ ← Average on all points

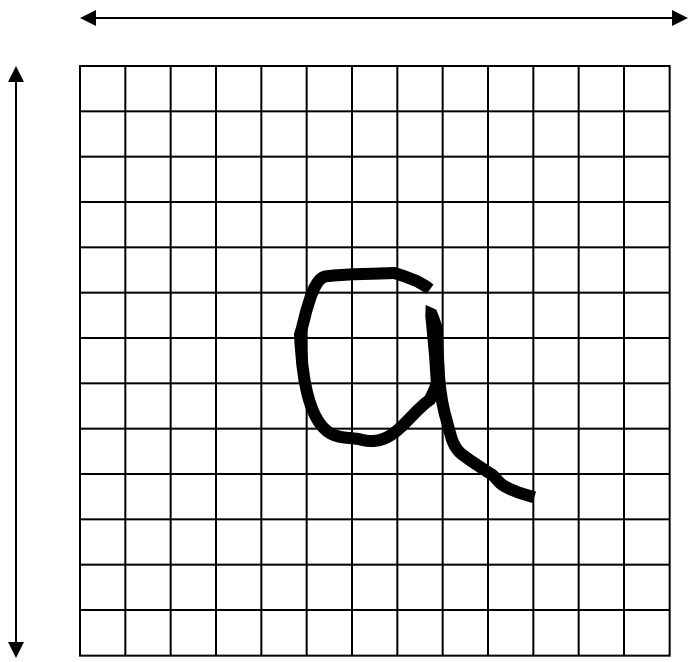
$$\sigma_i^2 = \frac{1}{N-1}\sum_{n=1}^{N}\left(x_i^n - \overline{x_i}\right)^2$$ ← Variance calculation

$$x_i^n = \frac{x_i^n - \overline{x_i}}{\sigma_i}$$ ← Variables transposition

Components reduction

- Sometimes, the number of inputs is too large to be exploited
- The reduction of the input number simplifies the construction of the model
- Goal : Better representation of the data in order to get a more synthetic view without losing relevant information
- Reduction methods (PCA, LDA, SOM etc.)

Character recognition example



- Image 256x256 pixels
- 8 bits pixels values (grey level)

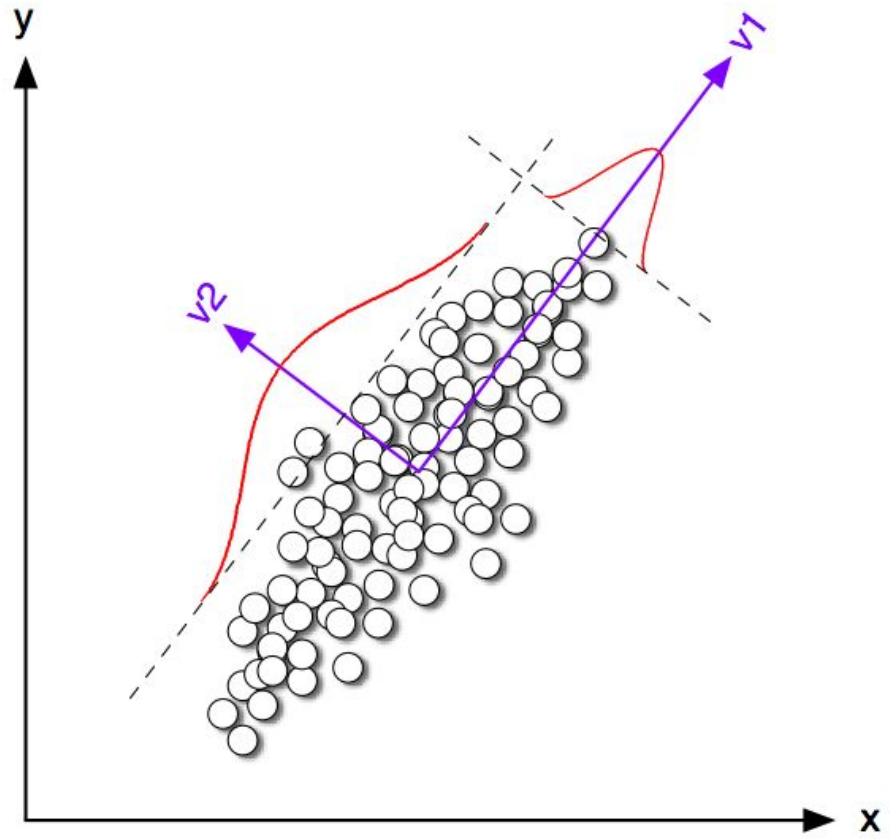
$$2^{256\times256\times8} \approx 10^{158000} \text{ different images}$$

- Necessary to extract features

Linear Principal Components Analysis (PCA)

- Principle
  - Linear projection method to reduce the number of parameters
  - Transfer a set of correlated variables into a new set of uncorrelated variables
  - Map the data into a space of lower dimensionality
  - Form of unsupervised learning
- Properties
  - It can be viewed as a rotation of the existing axes to new positions in the space defined by original variables
  - New axes are orthogonal and represent the directions with maximum variability

Example of data representation using PCA



Other methods

Neural pre-processing

- Use a neural network to reduce the dimensionality of the input space
- Overcomes the limitation of PCA
- Auto-associative mapping => form of unsupervised training

Intelligent preprocessing

- Use an “a priori” knowledge of the problem to help the neural network in performing its task
- Reduce the dimension of the problem by extracting the relevant features manually
- More or less complex algorithms to process the input data

Alternative Error Functions

Penalize large weights:

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

Train on target slopes as well as values:

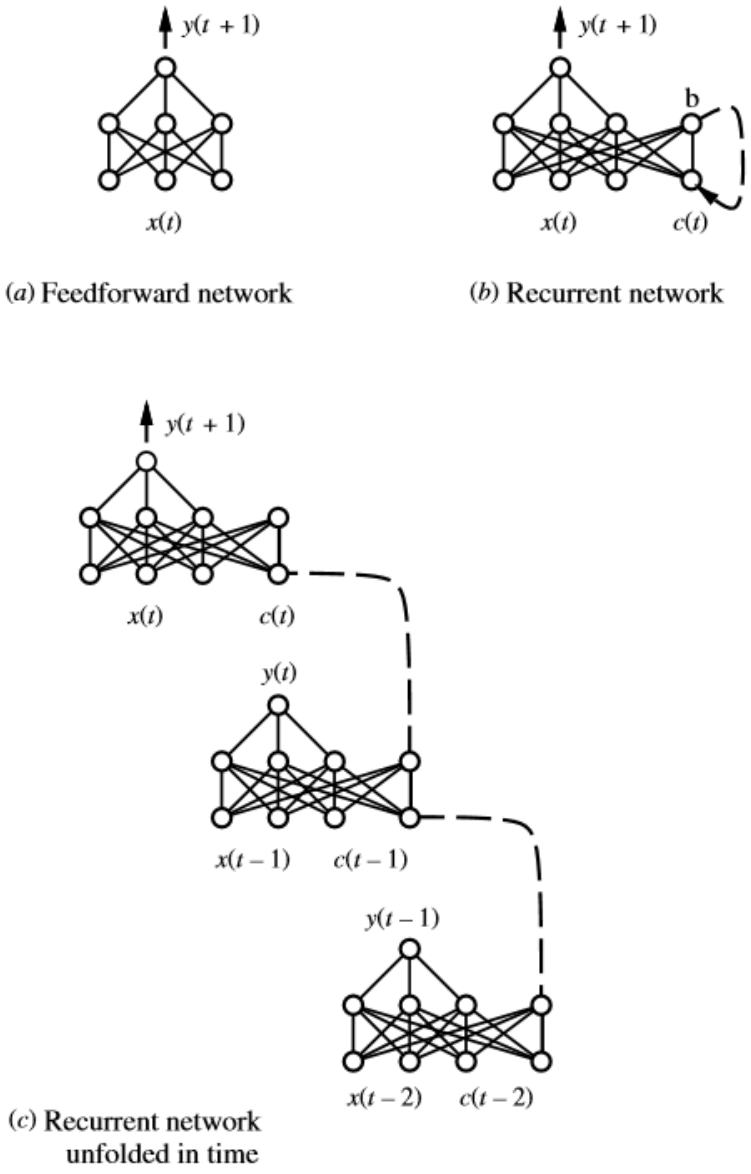
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Tie together weights:

- e.g., in phoneme recognition network

Recurrent Networks

- Apply to time series data
- use feedback and can learn finite state machines with "backpropagation through time."



Other Issues in Neural Nets

- Alternative Error Minimization Procedures:
  - Linear search
  - Conjugate gradient (exploits 2nd derivative)

- Dynamically Modifying Network Structure:
  - Grow network until able to fit data
    - Cascade Correlation
    - Upstart
  - Shrink large network until unable to fit data
    - Optimal Brain Damage

Summary of Major NN Models

Model	*Architecture* *Neuron Characteristic*	*Learning Algorithm*	*Application*
Perceptron	Single-node, feedforward Binary-threshold	Supervised, error-correction	Pattern classification
Adaline	Single-node, feedforward Linear	Supervised, gradient descent	Regression
Multilayer perceptron	Multilayered, feedforward nonlinear sigmoid	Supervised, gradient descent	Function approximation
Reinforcement learning	Multilayered Binary-threshold	Supervised reward-punishment	Control
Support vector machines	Multilayered kernel based, binary-threshold	Supervised quadratic optimization	Classification, regression
Radial basis function net	Multilayered distance based, linear	Supervised gradient descent	Interpolation, regression, classification
Hopfield network	Single layer, feedback Binary threshold/sigmoid	Outer product correlation	CAM, optimization

Summary of Major ANN Models

Model	*Architecture* *Neuron Characteristic*	*Learning Algorithm*	*Application*
Boltzmann machine	Two layered, feedback Binary threshold	Stochastic gradient descent	Optimization
BSB	Single layered, feedback Linear threshold	Outer product correlation	Clustering
Bidirectional associative memory	Two layered, feedback Binary threshold	Outer product correlation	Associative memory
Adaptive resonance theory	Two layered Binary, faster-than-linear	Unsupervised competitive	Clustering, classification
Vector quantization	Single layer, feedback Faster than linear	Supervised, unsupervised competitive	Quantization, clustering
Mexican hat net	Single layer, feedback Linear threshold	None fixed weights	Activity clustering
Kohonen SOFM	Single layer Linear threshold	Unsupervised soft-competitive	Topological mapping
Pulsed neuron models	Single/multilayer Pulsed/IF neuron	None	Coincidence detection Temporal processing

Acknowledgement

Part of the slide materials were based on Dr. Rong Duan's Fall 2016 course CPE/EE 695A Applied Machine Learning at Stevens Institute of Technology.

Reference

The lecture notes in this lecture are mainly based on the following textbooks:

T. M. Mitchell, Machine Learning, McGraw Hill, 1997. ISBN: 978-0-07-042807-2

STEVENS

INSTITUTE *of* TECHNOLOGY

THE INNOVATION UNIVERSITY®

stevens.edu