Artificial Neural Networks (Shallow Network)

Machine learning by Tom Mitchell: Chapter 4

Machine Learning for Beginners: An Introduction to Neural Networks
https://victorzhou.com/blog/intro-to-neural-networks/

Due Date Reminder

Redeem your Google Cloud coupons? Any problem?

Software installation & use (editors), jupyter + google colab

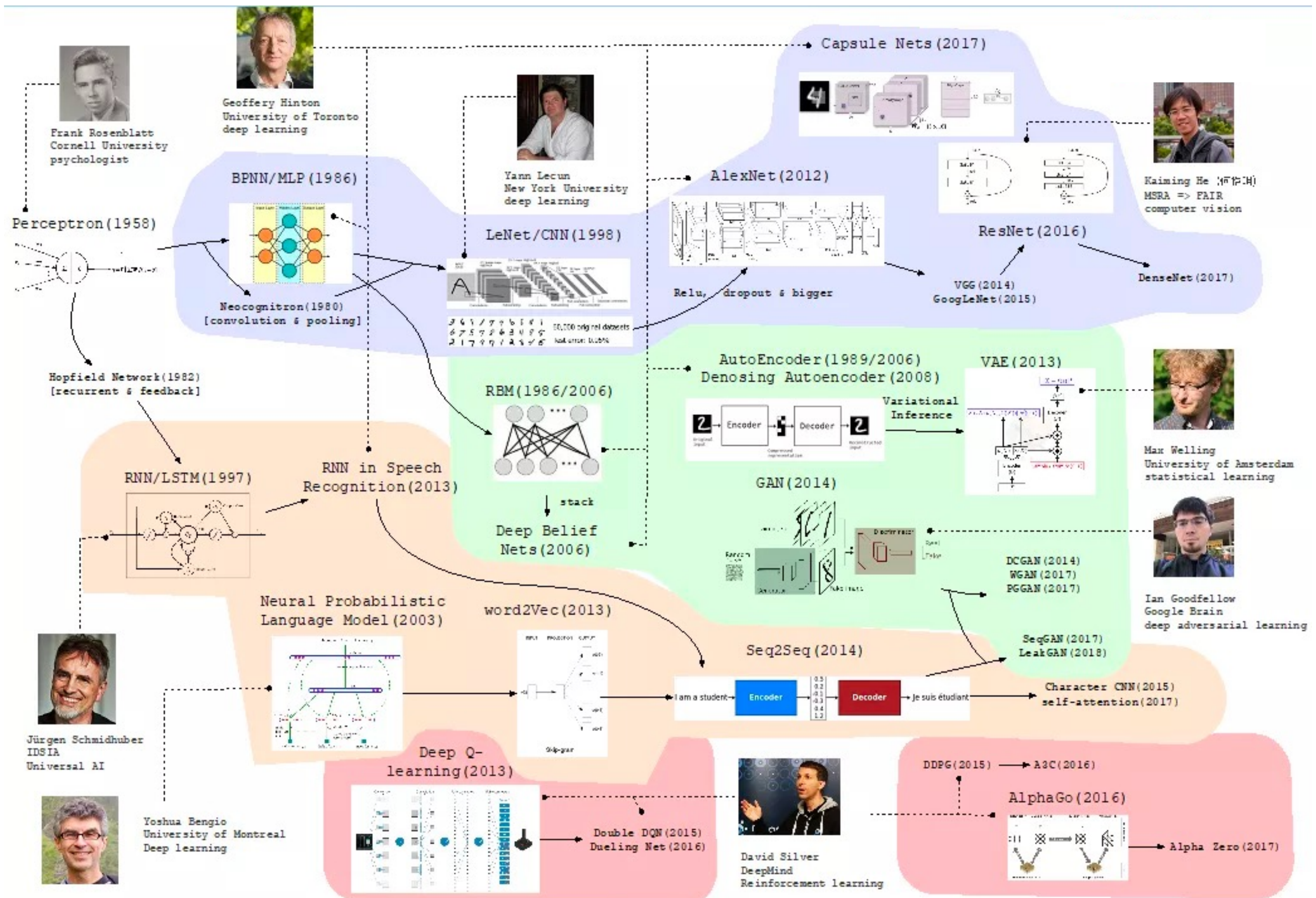
Deadline for choosing project groups (also support each other), with tentative project title, for paper review and presentation: 6 pm, 04/18/2023

Topic for paper review + presentation can be the same as your final project.

Topics already taken:

-
-

Last Lecture



- Cool applications
- NN history
- NN basics

Feedforward

Self-learning
Autoencoder
VAE

Recurrent

Reinforcement Learn

Demo

Image Classification on MNIST using CNN

Convolutional neural networks (CNN) – the concept behind recent breakthroughs and developments in deep learning.

Demo: 10-class

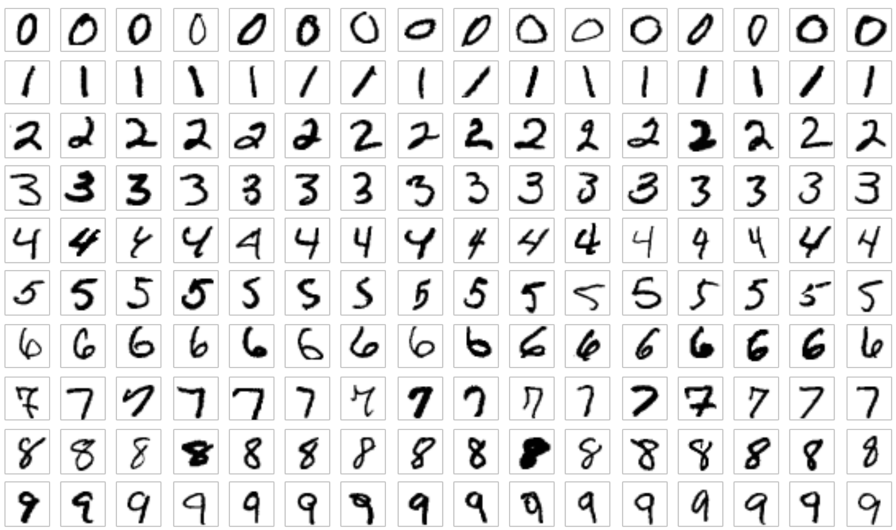
- MNIST

TO-DO: choose 1 task, select 2-5 classes, ~2 weeks

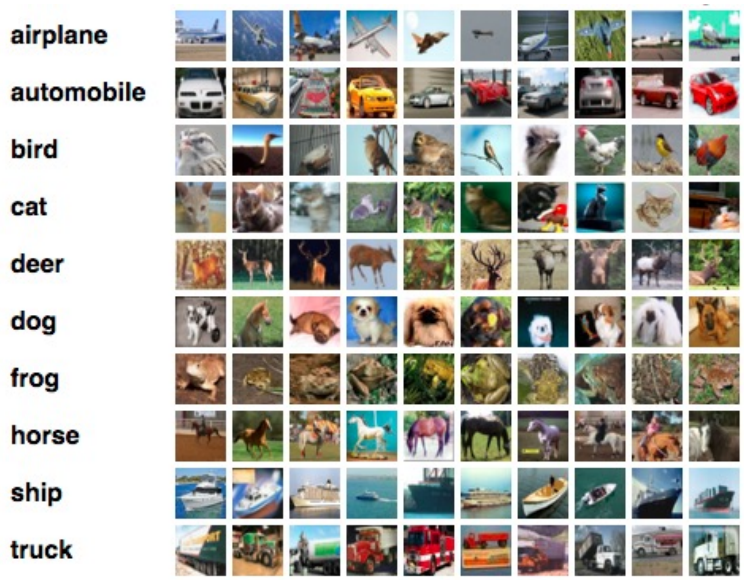
- CIFAR-10
- ImageNet

MNIST

Modified National Institute of Standards and Technology



CIFAR-10 Dataset



ImageNet

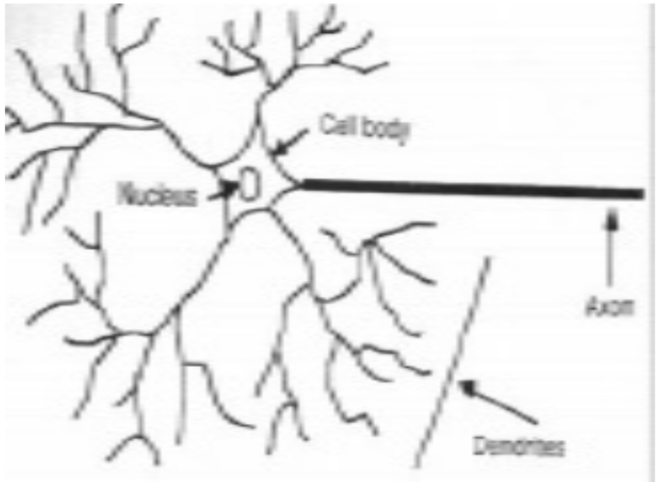
Outline

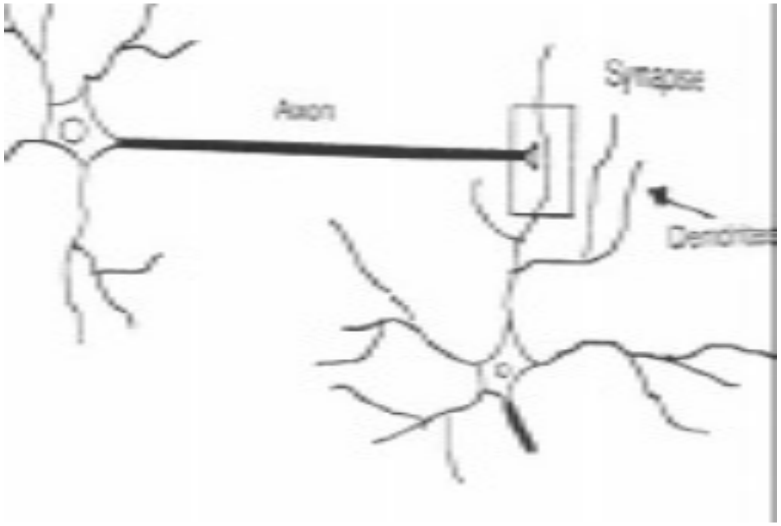
- Introduction
- Perceptrons
- Multilayer networks
- Backpropagation Algorithm
- Example: Face Recognition
- Advanced Topics

Introduction

Consider humans

- Neuron switching time ~.001 second
- # neurons ~10^{10}
- Connections per neuron ~10^{4-5}
- Scene recognition time ~.1 second
- 100 inference step does not seem like enough
- *must use lots of parallel computation!*

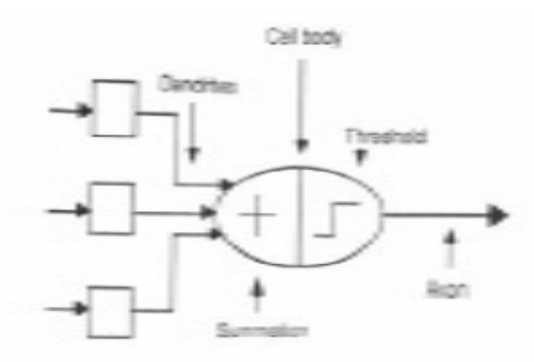


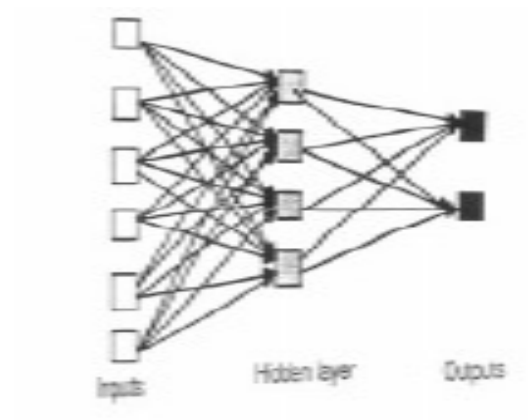


Introduction

Properties of artificial neural nets (ANNs):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

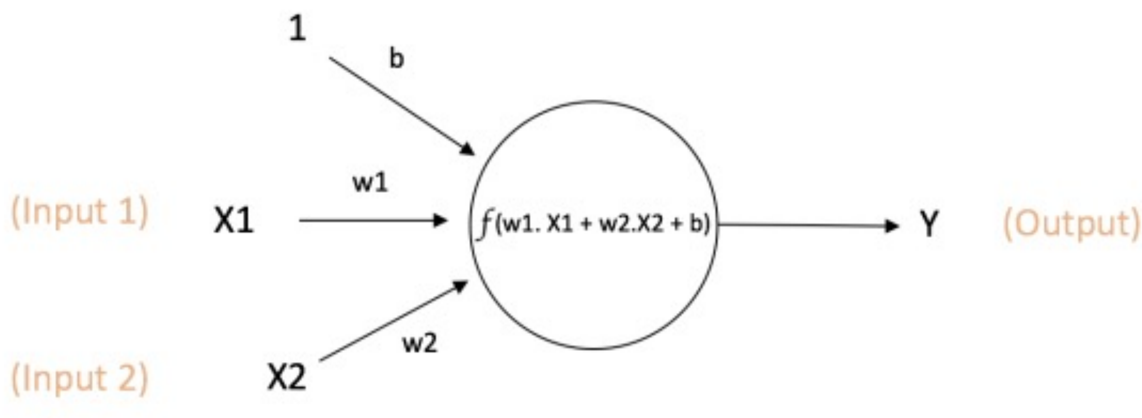




A Quick Introduction to Neural Networks

https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

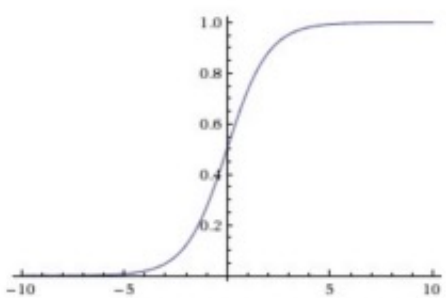
A Single Neuron



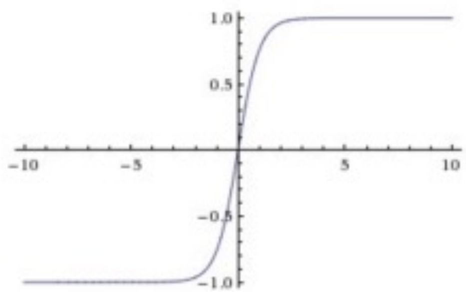
Output of neuron = Y= f(w1. X1 + w2.X2 + b)

Components of neuron/node/unit: inputs, weights (w), bias (b), activation function

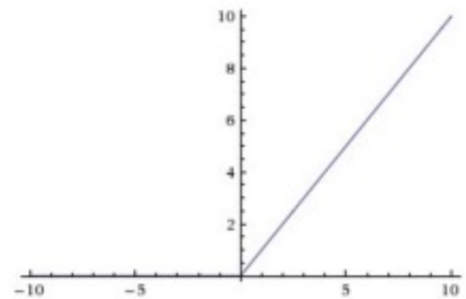
Activation Function: Non-Linearity



Sigmoid



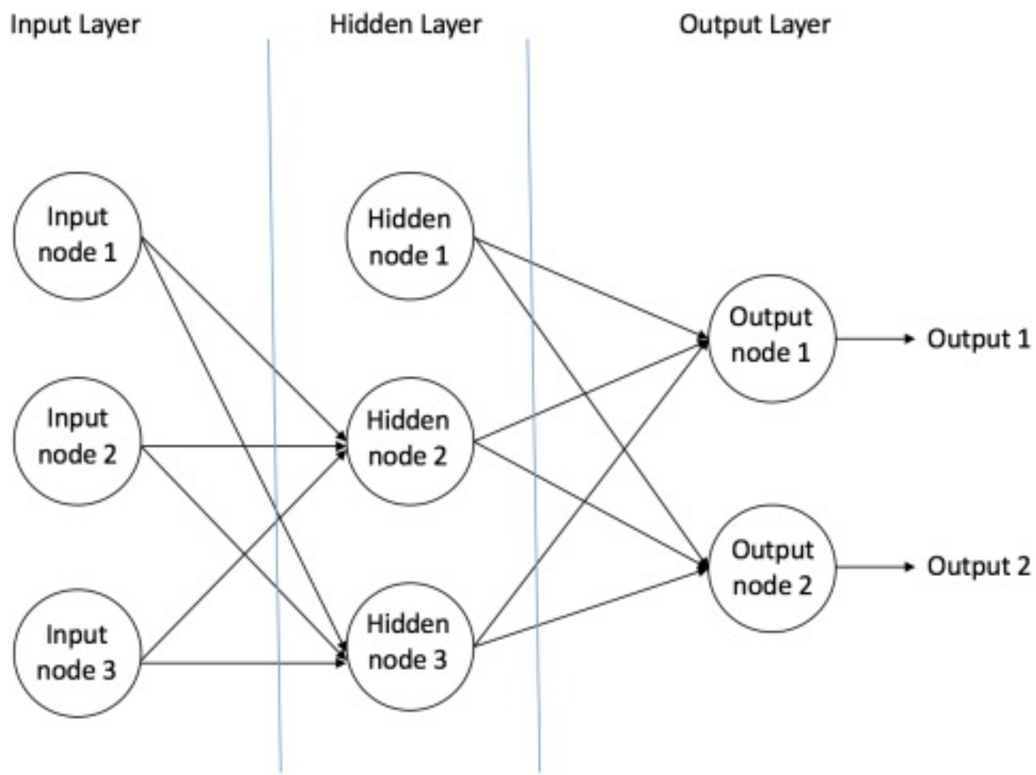
tanh



ReLU

- **Sigmoid:** $\sigma(x) = 1 / (1 + \exp(-x))$, [0 1]
- **Tanh:** $\tanh(x) = 2\sigma(2x) - 1$, [-1 1]
- **ReLU (**Rectified Linear Unit**):** $f(x) = \max(0, x)$

Feedforward Neural Network

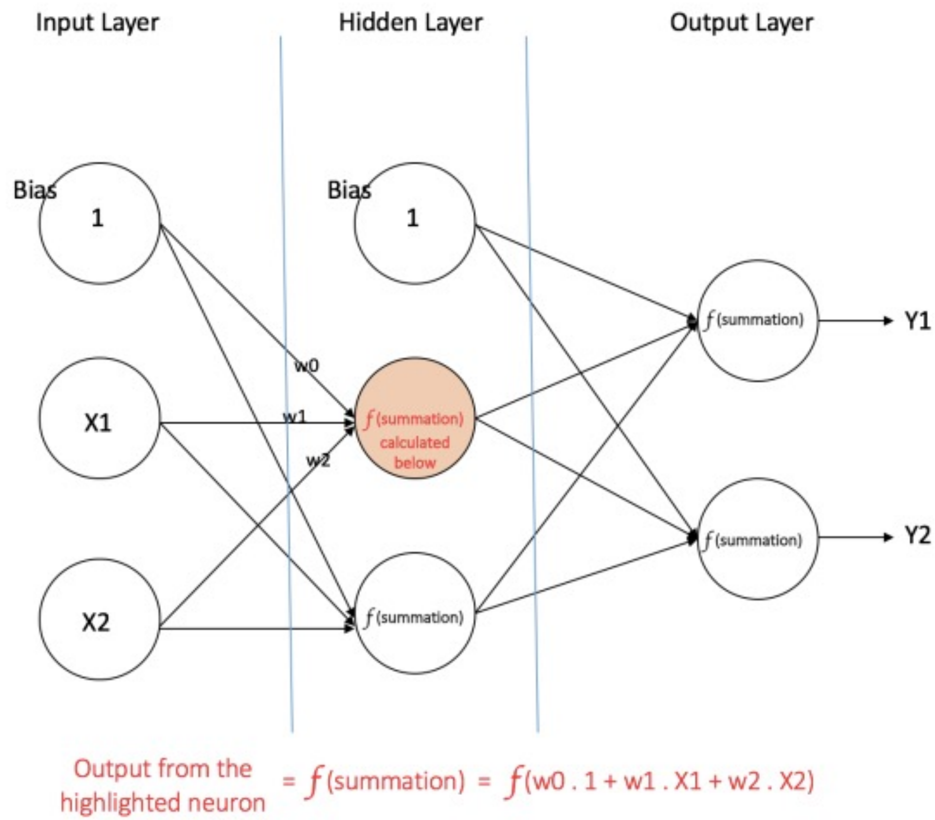


Layers, connections or edges, weights

Two examples:

- Single Layer Perceptron
- Multi Layer Perceptron

Multi Layer Perceptron (MLP)



A multi layer perceptron having one hidden layer

Understanding Multi Layer Perceptrons (MLP): example

Student-marks dataset:
2 inputs, 1 output (1/0)

Hours Studied	Mid Term Marks	Final Term Result
35	67	1 (Pass)
12	75	0 (Fail)
16	89	1 (Pass)
45	56	1 (Pass)
10	90	0 (Fail)

Prediction:
Input to the network = [25, 70]
Desired output from the network (target) = [1, 0]

Hours Studied	Mid Term Marks	Final Term Result
25	70	?

Training our MLP: The Back-Propagation (BP) Algorithm

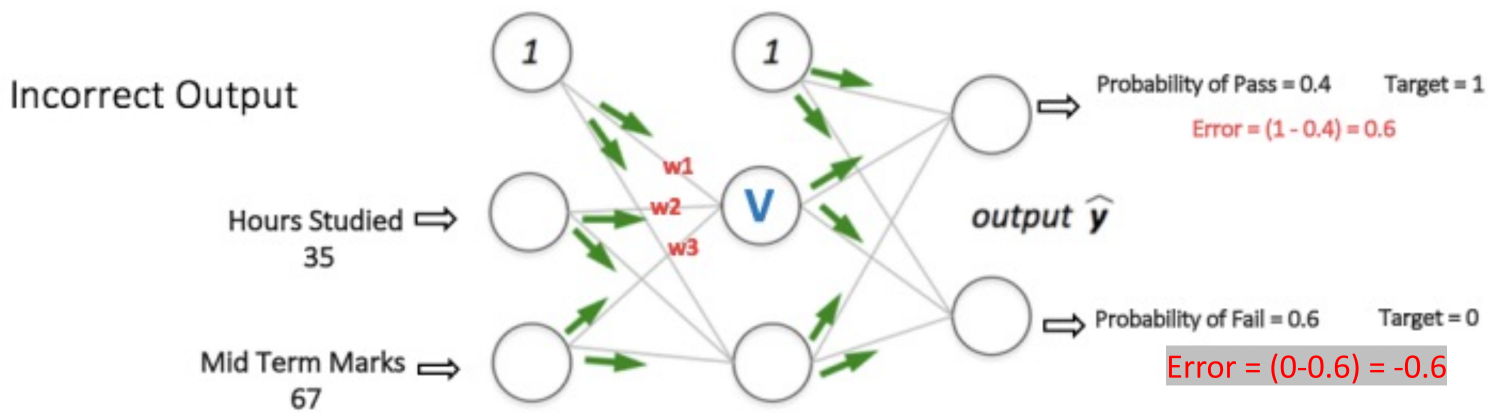
Student-marks dataset:
2 inputs, 1 output (1/0)

Step 1: Forward Propagation

Hours Studied	Mid Term Marks	Final Term Result
35	67	1 (Pass)
12	75	0 (Fail)
16	89	1 (Pass)
45	56	1 (Pass)
10	90	0 (Fail)

Input to the network = [35, 67]

$$\tilde{y} = \boldsymbol{f}\ (1*w1 + 35*w2 + 67*w3)$$



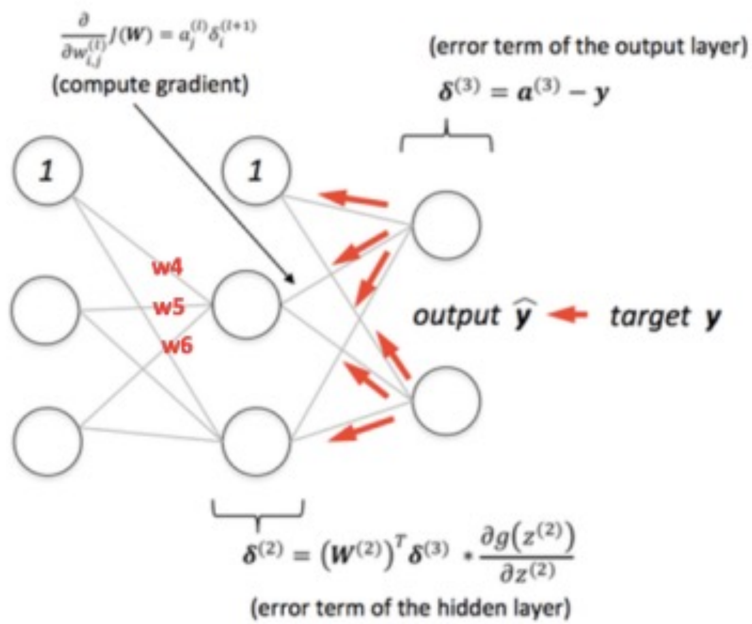
forward propagation step in a multi layer perceptron

Training our MLP: The Back-Propagation (BP) Algorithm

Step 2: Back Propagation and Weight Updating

Use an optimization method such as *Gradient Descent* to 'adjust' **all** weights in the network with an aim of reducing the error at the output layer

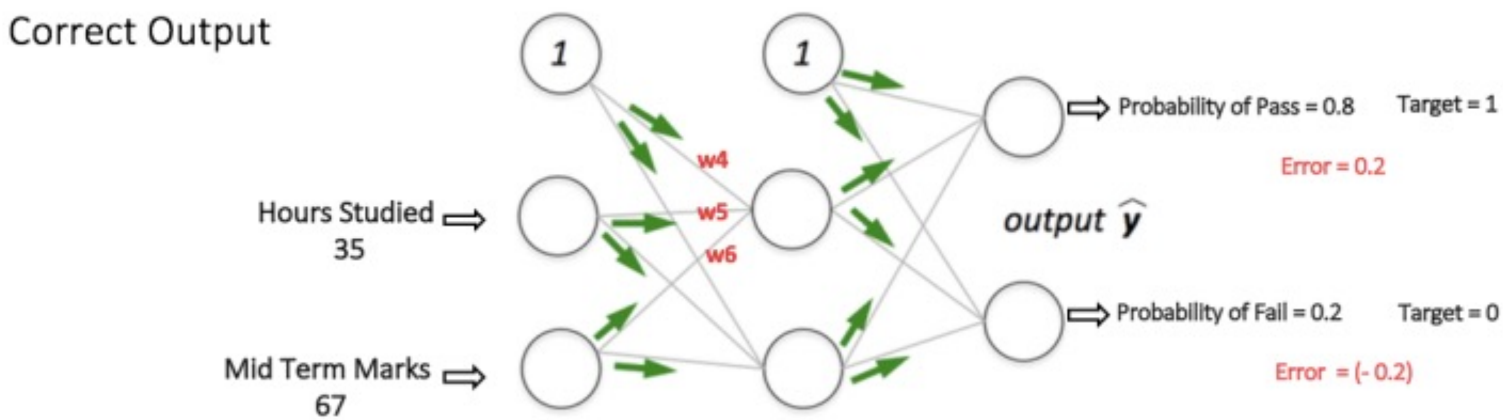
Backpropagation
+
Weights Adjusted



backward propagation and weight updating step in a multi layer perceptron

Training our MLP: The Back-Propagation (BP) Algorithm

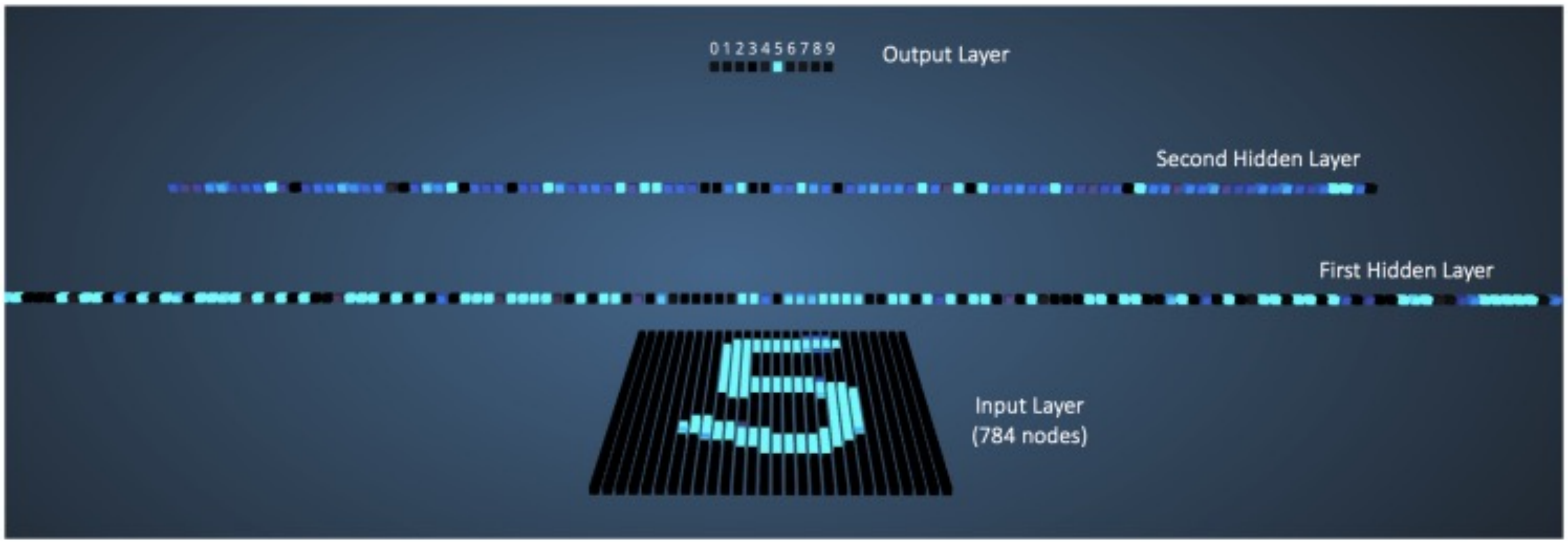
Step 2: Back Propagation and Weight Updating



MLP network now performs better on the same input

3d Visualization of a Multi Layer Perceptron

http://scs.ryerson.ca/~aharley/vis/fc/



visualizing the network for an input of ‘5’

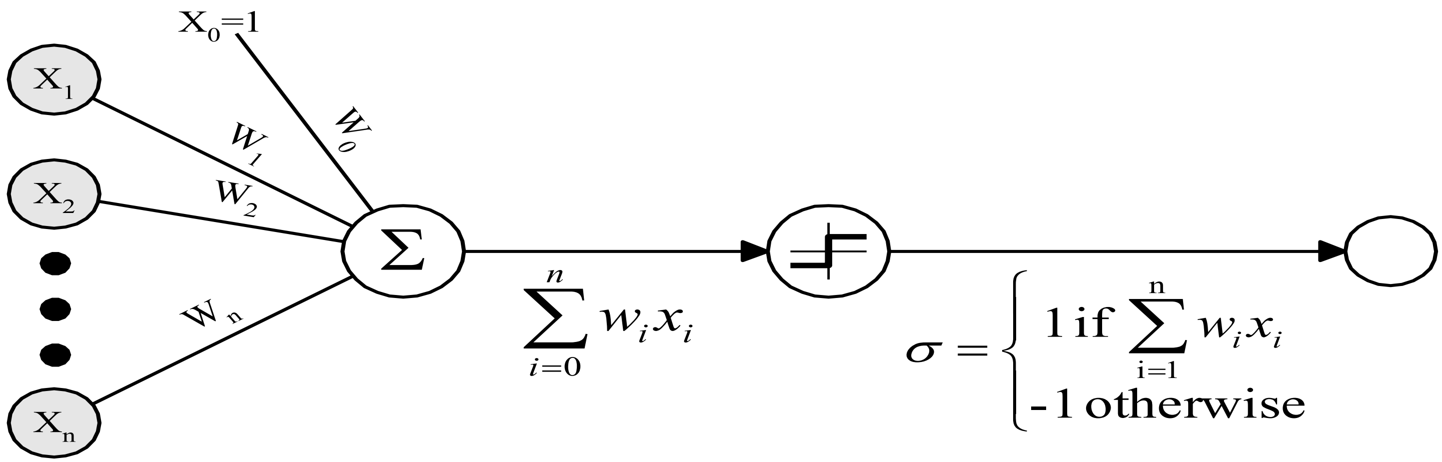
When to Consider Neural Networks

- 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
- Form of target function is unknown
- Human readability of result is *unimportant*

Examples:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

Perceptron



- Input values → Linear weighted sum → Threshold
- Given real-valued inputs X_1 through X_n, the output $o(x_1,\ldots,x_n)$ computed by the perceptron is

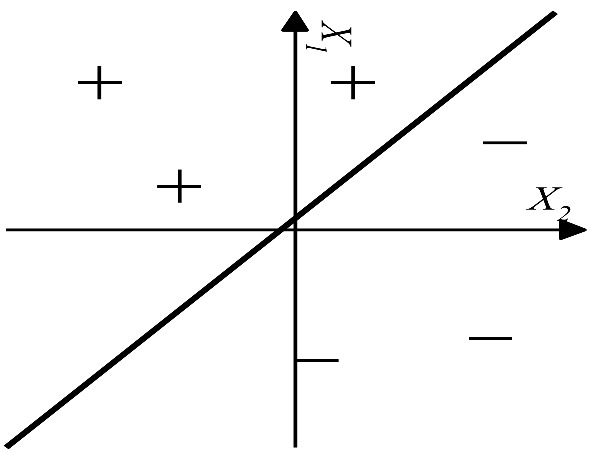
$$o(x_1,\ldots,x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where w_i is a real-valued constant, or weight

Decision Surface of Perceptron: $\vec{w} \cdot \vec{x} = 0$

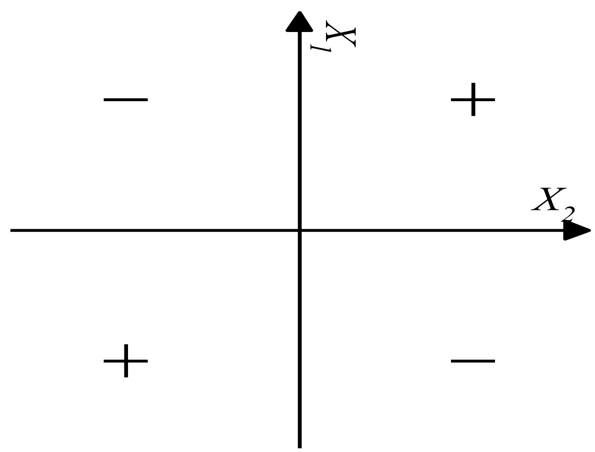
Linearly separable case :

Possible to classify by hyperplane



Linearly inseparable case :

Impossible to classify



Perceptron Training Rule

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

where

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

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

Can prove it will converge

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

Gradient Descent

To understand, consider simple *linear unit*, where

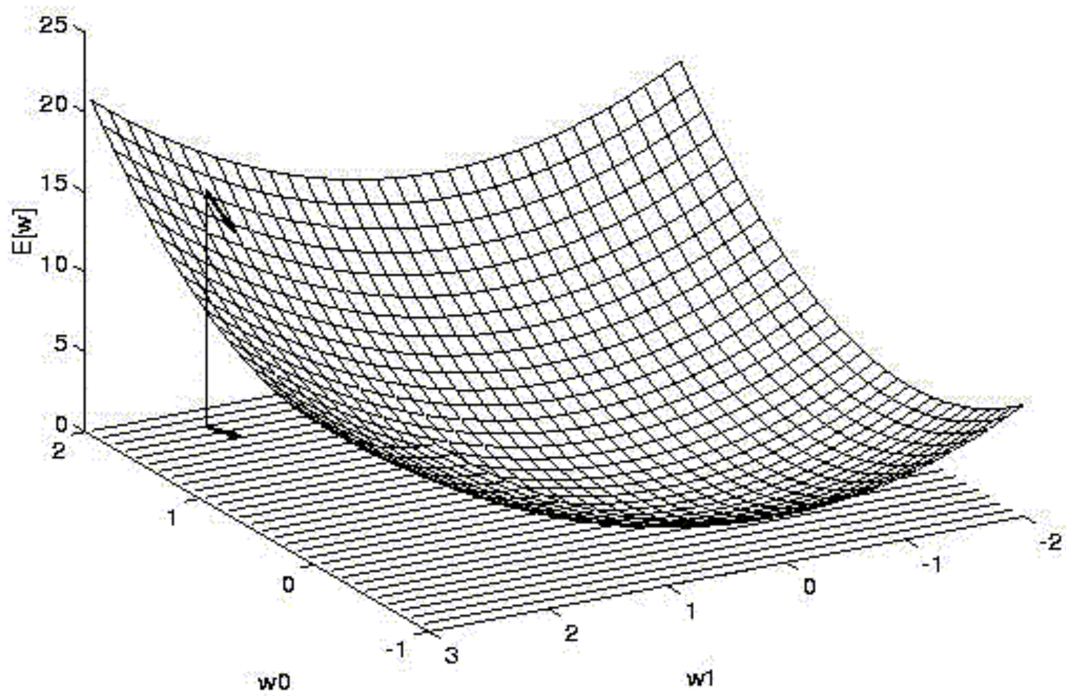
$$o = w_0 + w_1 x_1 + ... + w_n x_n$$

Idea: 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 the set of training examples

Hypothesis space



- W_0, W_1 plane represents the entire hypothesis space.
- For linear units, this error surface must be parabolic with a single global minimum. And we desire a hypothesis with this minimum.

Gradient (steepest) Descent Rule

Error (for all training examples):

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

$$\text{Gradient} \qquad \nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]$$

direction : steepest increase in *E*.

$$\text{Training rule:} \ \Delta w_i = -\eta \nabla E[\vec{w}]$$

$$\text{i.e.,} \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

(The negative sign : decreases *E*.)

Derivation of Gradient Descent

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small η is used.

Gradient descent and delta rule

Search through the space of possible network weights, iteratively reducing the error *E* between the training example target values and the network outputs

GRADIENT-DESCENT(*training_examples*, η)

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).

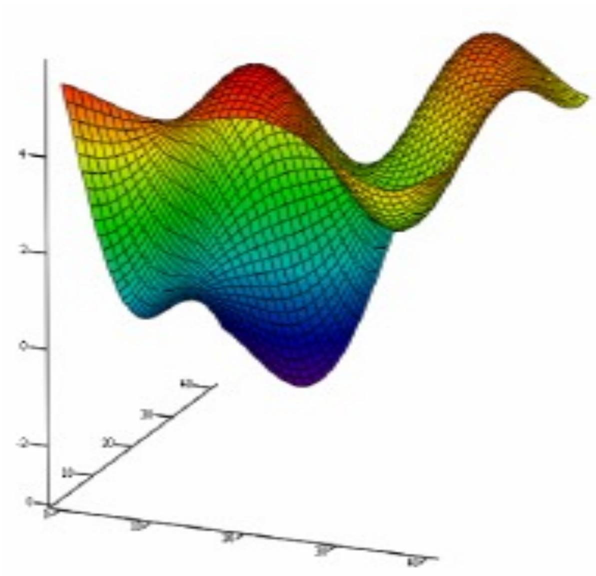
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight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

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

  - For each linear unit weight w_i, Do

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

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. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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

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

Stochastic Gradient Descent (i.e. incremental mode) can sometimes avoid falling into local minima because it uses the various gradient of *E* rather than overall gradient of *E*.

Summary

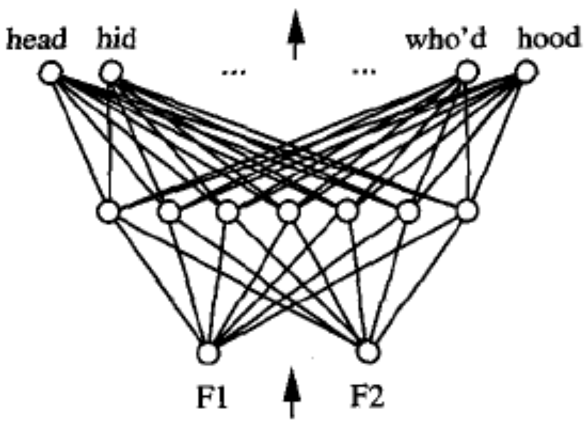
Perceptron training rule

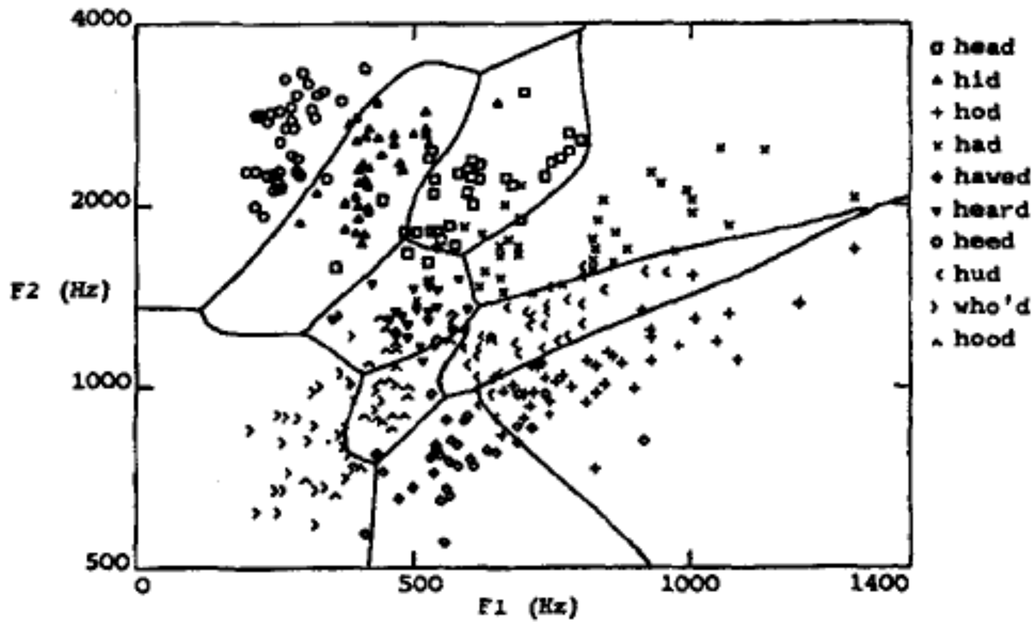
- Perfectly classifies training examples
- Converge, provided the training examples are linearly separable

Delta Rule uses gradient descent

- Converge asymptotically to hypothesis with minimum squared error
- Converge regardless of whether training data are linearly separable

Multilayer Networks and Backpropagation



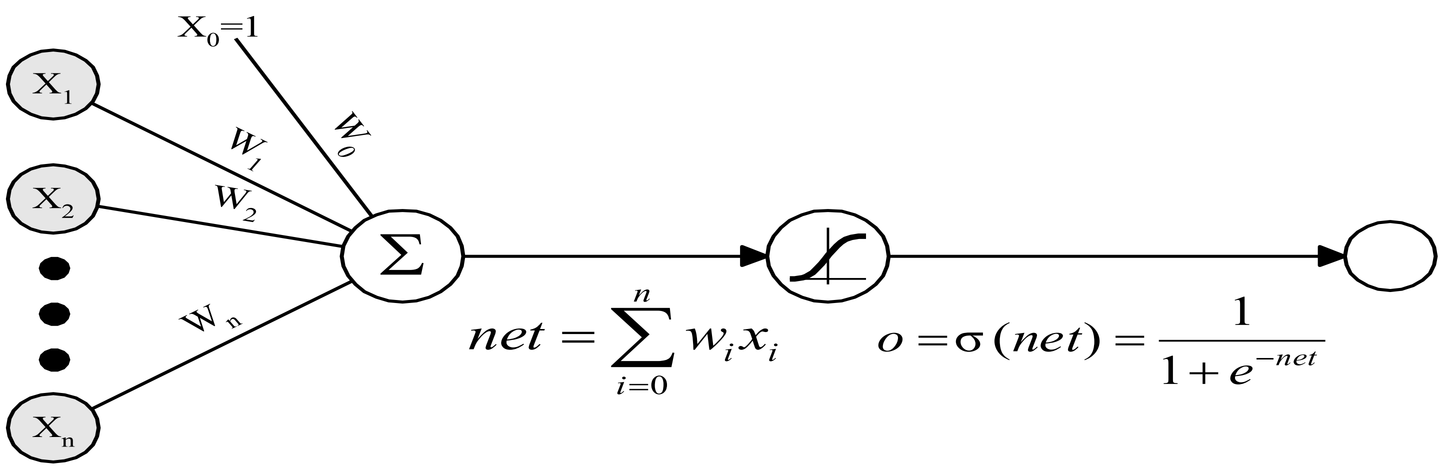


- Speech recognition example of multilayer networks learned by the backpropagation algorithm :
- trained to recognize 1 of 10 vowel sounds occurring in the context “h_d”, e.g. ‘had’, ‘hid’
- Highly nonlinear decision surfaces

Sigmoid threshold Unit

What type of unit as the basis for multilayer networks

- Perceptron : not differentiable -> can't use gradient descent
- Linear Unit : multi-layers of linear units -> still produce only linear function
- Sigmoid Unit : differentiable threshold function



$$net = \sum_{i=0}^{n} w_i x_i \qquad o = \sigma(net) = \frac{1}{1+e^{-net}}$$

Sigmoid Unit

$\sigma(x)$ is the sigmoid function

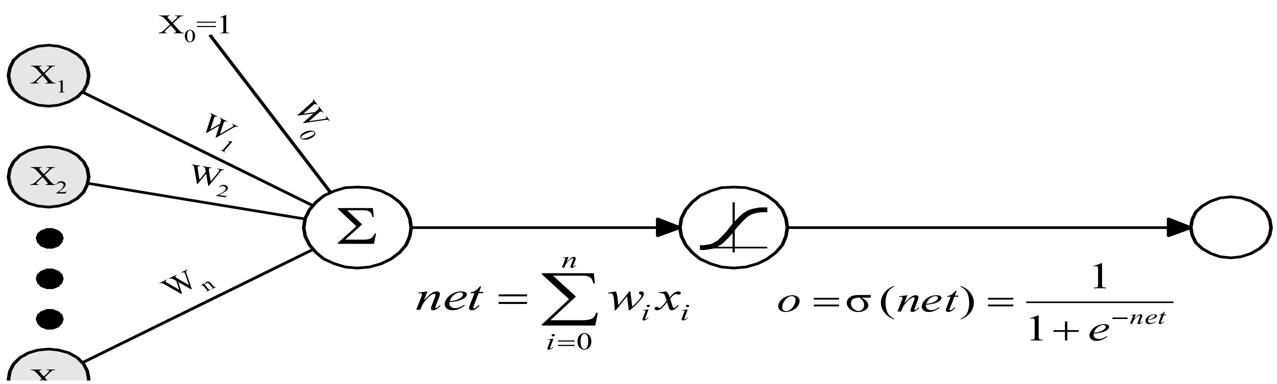
$$\frac{1}{1+e^{-x}}$$

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

Output ranges between 0 *and* 1

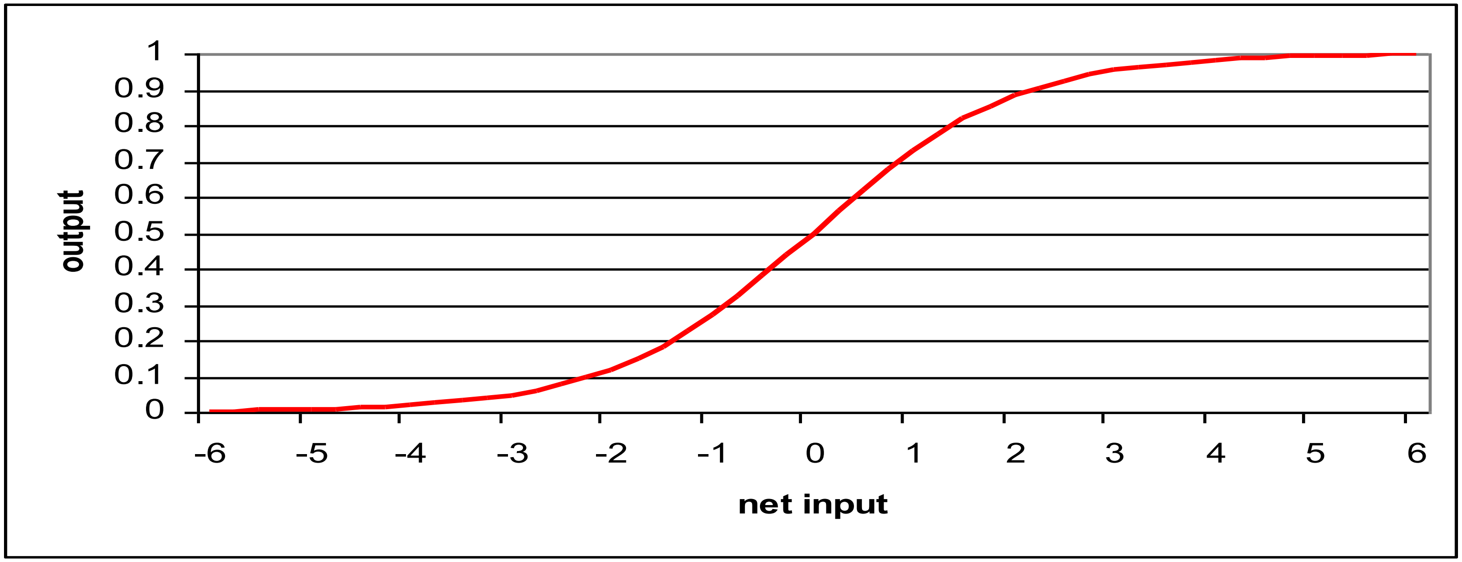
We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ *Backpropagation*



The Sigmoid Function

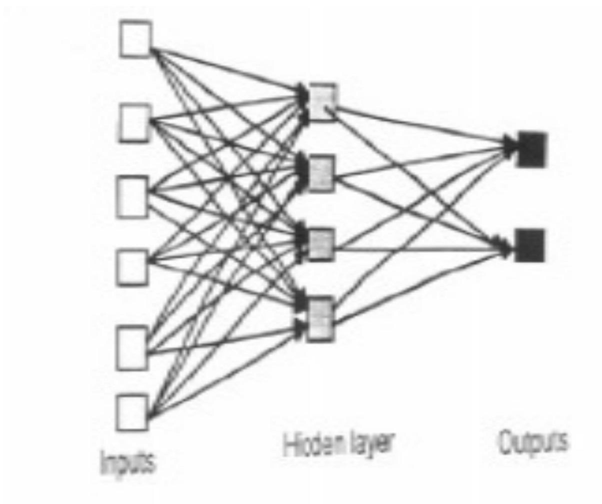
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Sort of a rounded step function
Unlike step function, can take derivative (makes learning possible)

Backpropagation (BP) Algorithm

Two layered feedforward networks



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

More on Backpropagation

- Adding momentum

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

- *n*-th iteration update depends on (*n*-1)th iteration
- α : constant between 0 and 1 -> momentum
- Role of momentum term :
    - Keep the ball rolling through small local minima in the error surface.
    - Gradually increase the step size of the search in regions where the gradient is unchanging, thereby speeding convergence

Remarks on Backpropagation Algorithm

- Convergence and local minima
  Perhaps not global minimum...
  - Add momentum
  - Stochastic gradient descent
  - Train multiple nets with different initial weights

Remarks on Backpropagation Algorithm

- Expressive capabilities of ANNs

– Boolean functions:

Every boolean function can be represented by network with two layers of units where the number of hidden units required grows exponentially.

– Continuous functions:

Every bounded continuous function can be approximated with arbitrarily small error, by network with two layers of units [Cybenko 1989; Hornik et al. 1989]

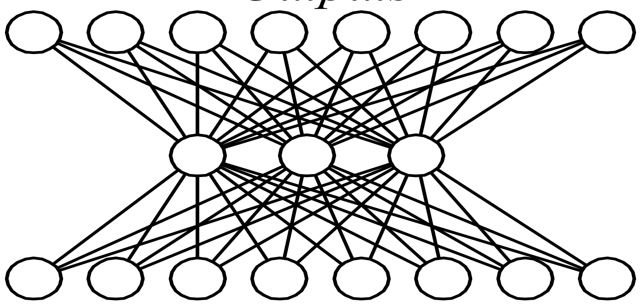
– Arbitrary functions:

Any function can be approximated to arbitrary accuracy by a network with three layers of units [Cybenko 1988].

Back-propagation Using Gradient Descent

- Advantages
  - Relatively simple implementation
  - Standard method and generally works well
- Disadvantages
  - Slow and inefficient
  - Can get stuck in local minima resulting in sub-optimal solutions

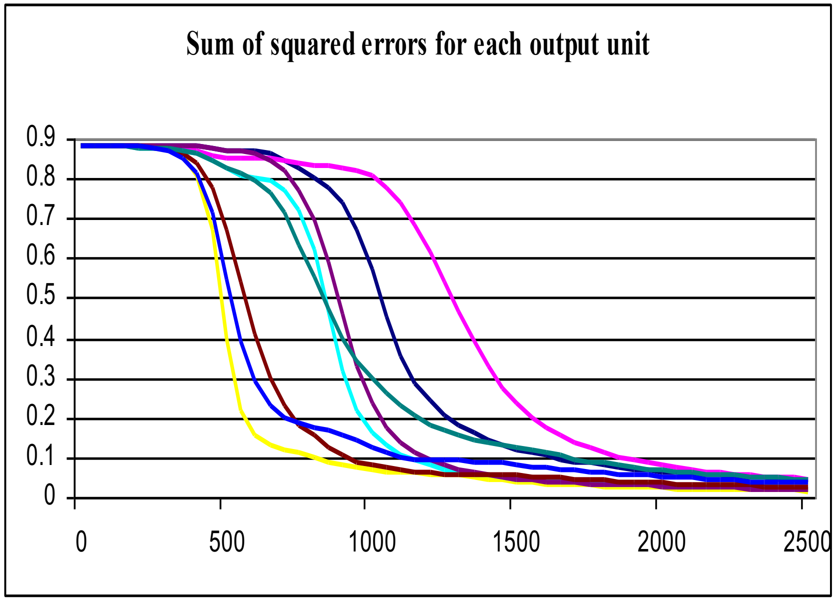
Learning Hidden Layer Representations



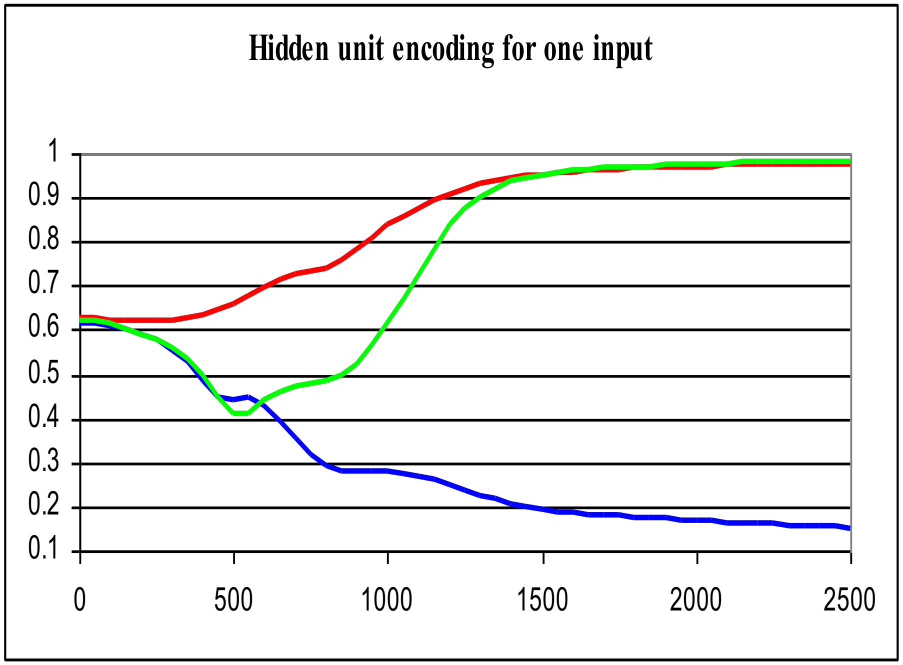
- This 8x3x8 network was trained to learn the identity function.
- 8 training examples are used.
- After 5000 training iterations, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right.

Input				Output
10000000	$\rightarrow$	.89 .04 .08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01 .11 .88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01 .97 .27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99 .97 .71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03 .05 .02	$\rightarrow$	00001000
00000100	$\rightarrow$	.22 .99 .99	$\rightarrow$	00000100
00000010	$\rightarrow$	.80 .01 .98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60 .94 .01	$\rightarrow$	00000001

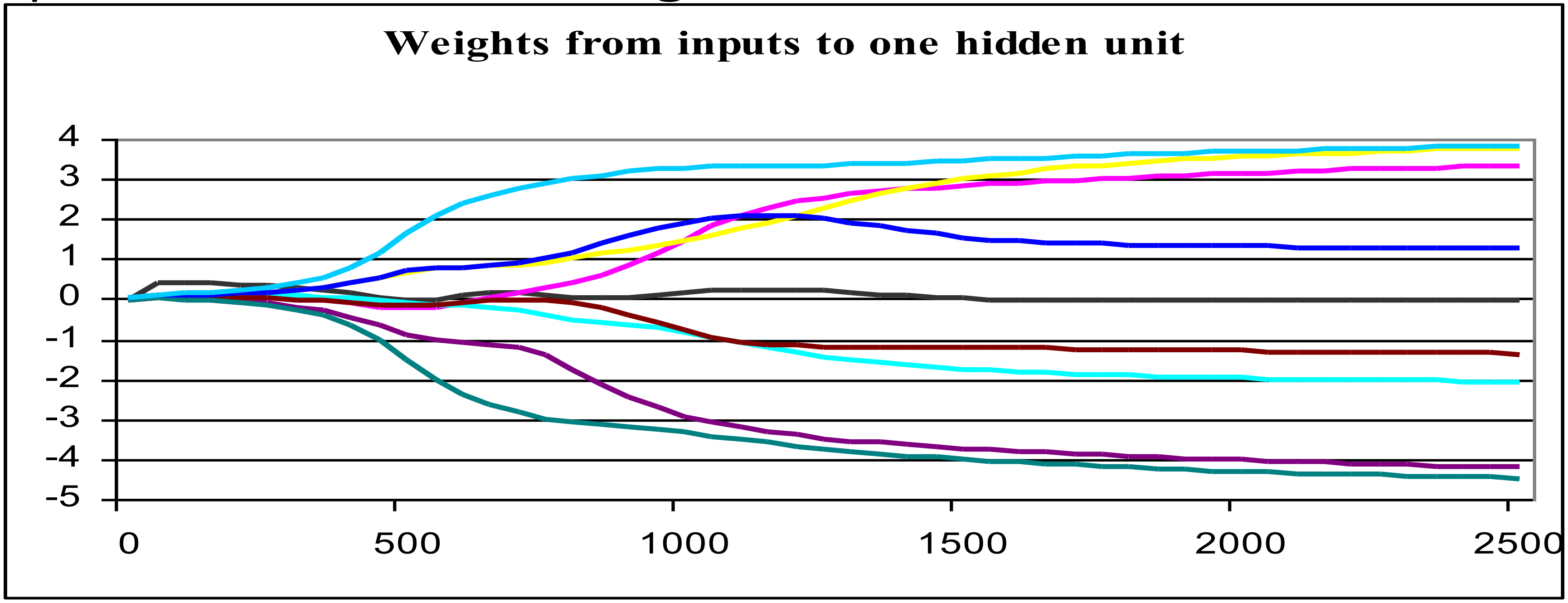
Output Unit Error during Training



Hidden Unit Encoding



Input to Hidden Weights



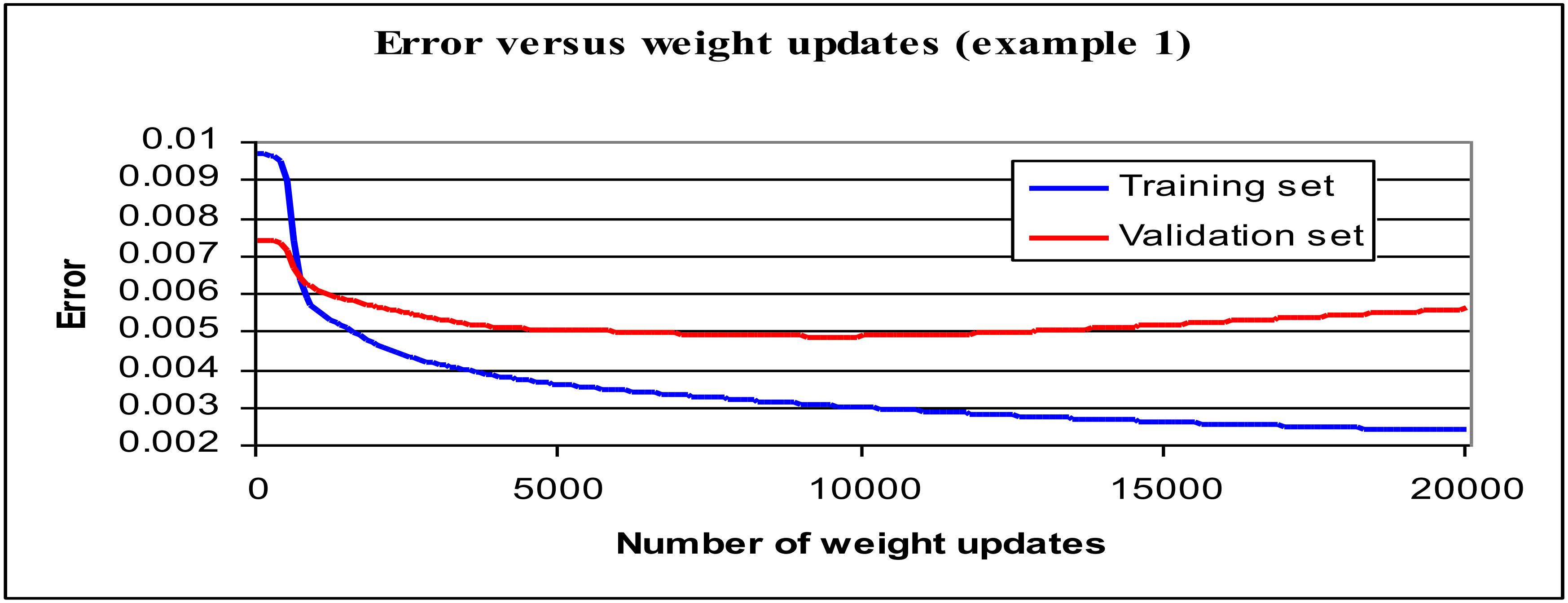
Remarks on Backpropagation Algorithm

Generalization, overfitting, and stopping criterion

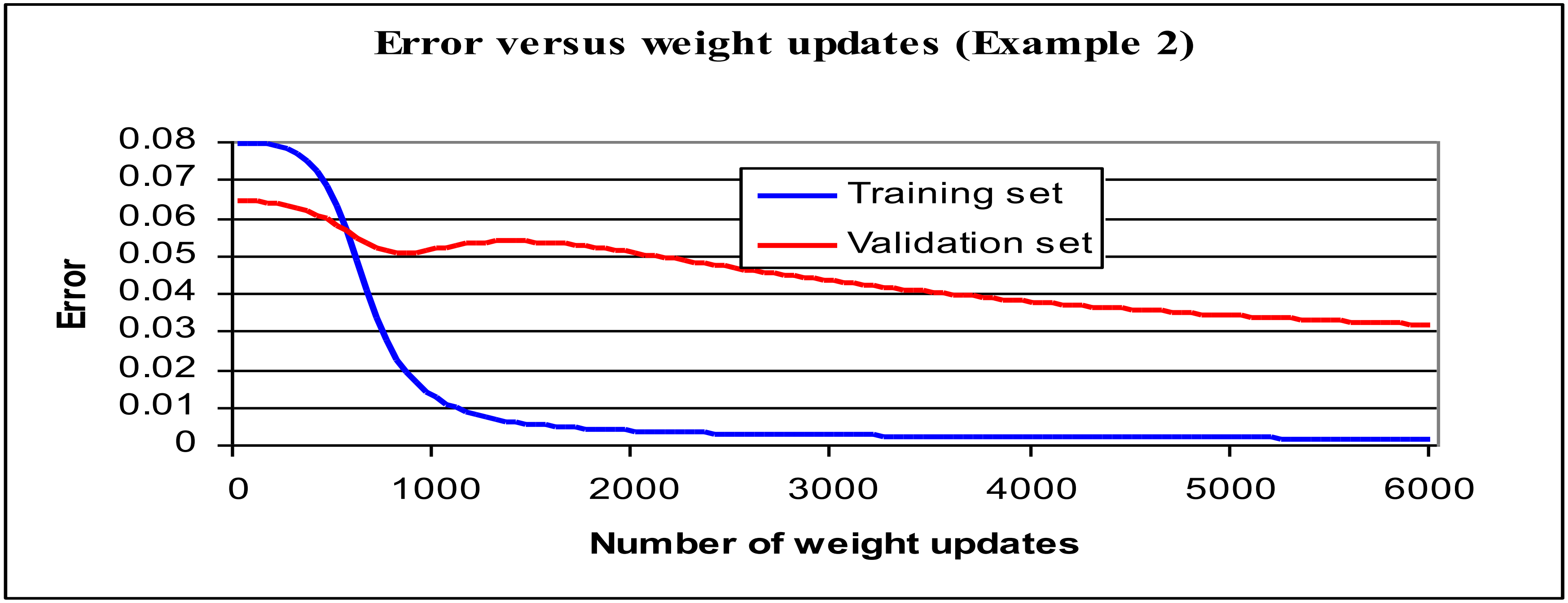
– Termination condition
• Until the error *E* falls below some predetermined threshold
(overfitting problem)

– Techniques to address the overfitting problem
• Weight decay: Decrease each weight by some small factor during each iteration.
• Cross-validation
• *k*-fold cross-validation (small training set)

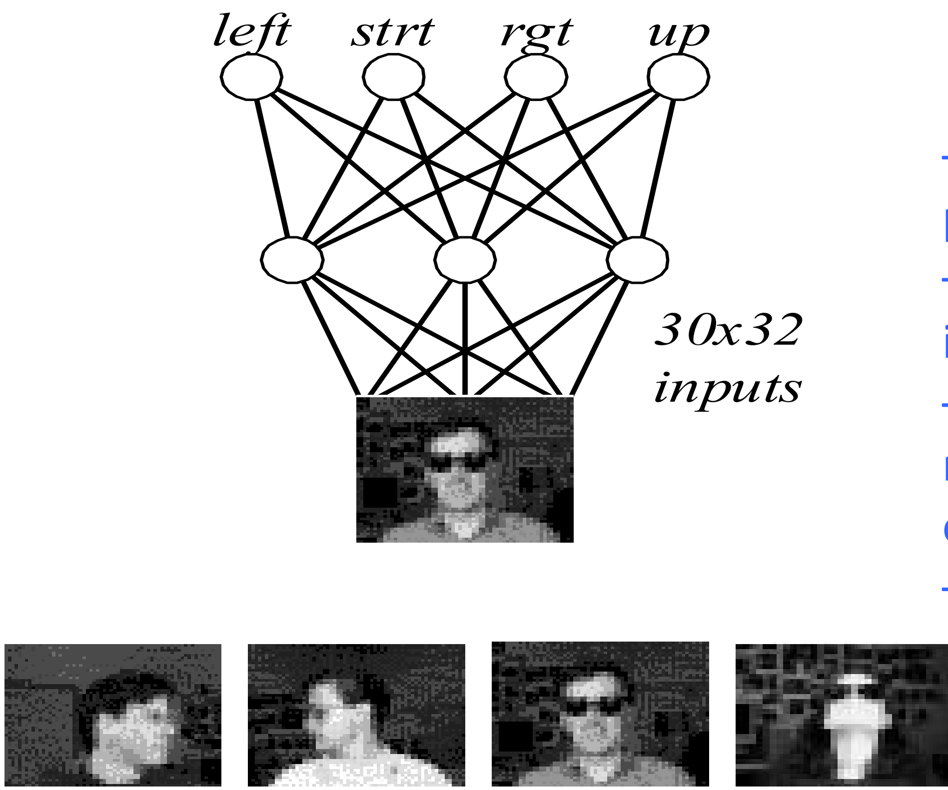
Overfitting in ANNs



Overfitting in ANNs



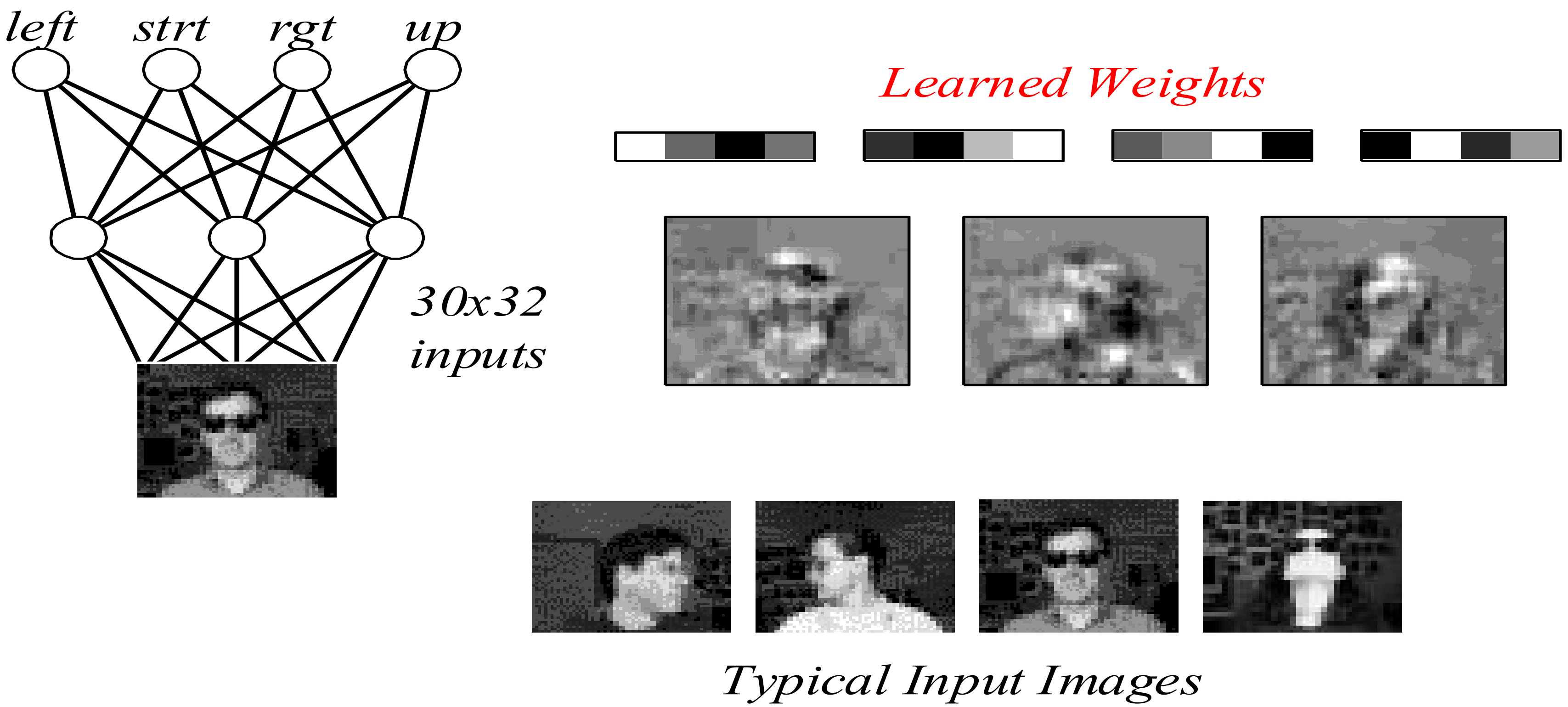
Example: Neural Nets for Face Recognition



Typical Input Images

- Training images : 20 different persons with 32 images per person.
- (120x128 resolution → 30x32 pixel image)
- After 260 training images, the network achieves an accuracy of 90% over a separate test set.
- Algorithm parameters : η=0.3, α=0.3

Learned Network Weights



Setting the parameter values

- How are the weights initialized?
- Do weights change after the presentation of each pattern or only after all patterns of the training set have been presented?
- How is the value of the learning rate chosen?
- When should training stop?
- How many hidden layers and how many nodes in each hidden layer should be chosen to build a feedforward network for a given problem?
- How many patterns should there be in a training set?
- How does one know that the network has learnt something useful?

Neural Networks: Advantages

- **Distributed representations**
- **Simple computations**
- **Robust with respect to noisy data**
- **Robust with respect to node failure**
- **Empirically shown to work well for many problem domains**
- **Parallel processing**

Neural Networks: Disadvantages

- **Training is slow**
- **Interpretability is hard**
- **Network topology layouts ad hoc**
- **Can be hard to debug**
- **May converge to a local, not global, minimum of error**
- **Not known how to model higher-level cognitive mechanisms**
- **May be hard to describe a problem in terms of features with numerical values**

Applications

- Classification:
  - Image recognition
  - Speech recognition
  - Diagnostic
  - Fraud detection
  - Face recognition ..

- Regression:
  - Forecasting (prediction on base of past history)
  - Forecasting e.g., predicting behavior of stock market

- Pattern association:
  - Retrieve an image from corrupted one
  - …

- Clustering:
  - clients profiles
  - disease subtypes
  - …

Reading

Chapter 6 of the Deep Learning Book :
Deep Feedforward Networks

Deep Learning, I. Goodfellow, Y. Bengio and A. Courville.
http://www.deeplearningbook.org/