

Introduction to Machine Learning

CS307 --- Fall 2022

Neural Networks

Reading:

Section 20.5, R&N

Section 4.4, Mitchell

Connectionist Models of Learning

Neural Networks

Characterized by:

- A large number of very simple neuron-like processing elements.
- A large number of weighted connections between the elements.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

Neural Networks

Rich history, starting in the early 40s. (McCulloch & Pitts 1943)

Two views:

- **Modeling the brain.**

Solving problems under the constraints similar to those of the brain may lead to solutions to AI problems that would otherwise be overlooked.

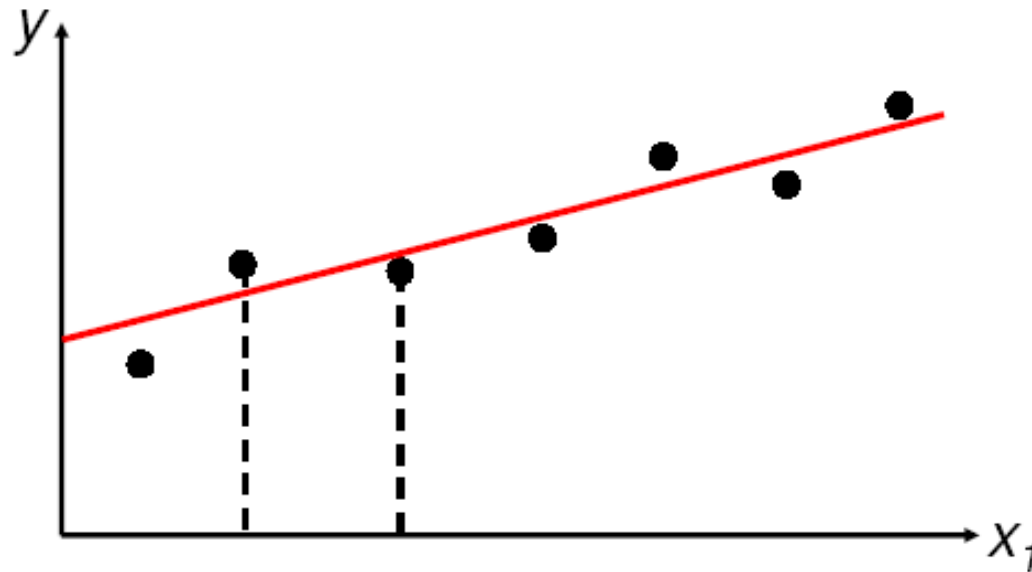
- **“Just” representation of complex functions.**

Continuous; contrast decision trees.

Much progress on both fronts.


Drawn interest from: *Neuro-science, Cognitive science, AI, Physics, Statistics, and CS / EE.*

Linear Regression



- We have training data $X = \{x_1^k\}$, $k=1, \dots, N$ with corresponding output $Y = \{y^k\}$, $k=1, \dots, N$
- We want to find the parameters that predict the output Y from the data X in a linear fashion: $y^k \approx w_0 + w_1 x_1^k$

Linear Regression



A graph illustrating linear regression. The vertical axis is labeled 'y'. Several black dots representing data points are plotted. A red line represents the linear regression fit, showing a positive correlation between the data points.

Notations:

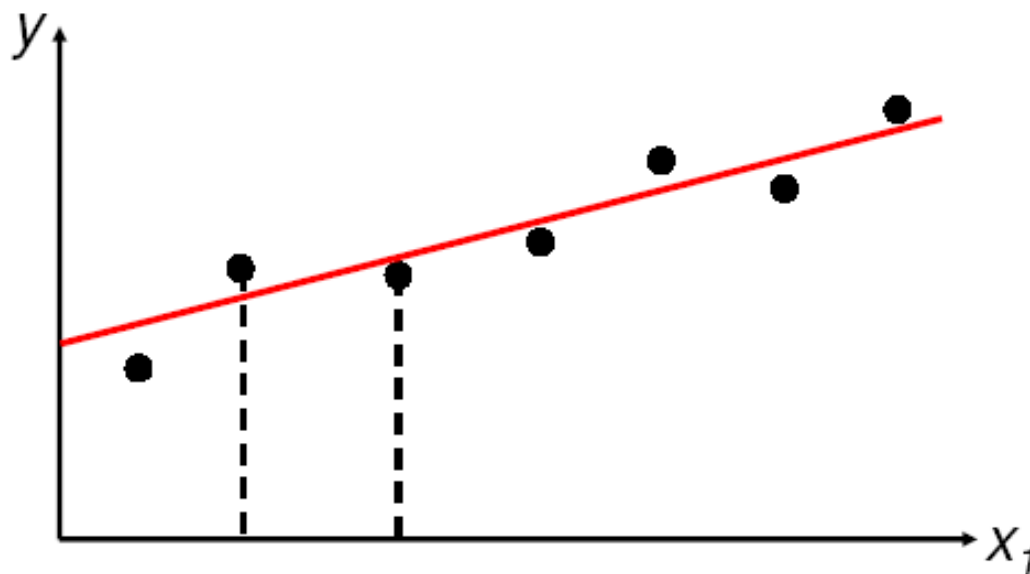
Superscript: Index of the data point in the training data set; $k = k^{\text{th}}$ training data point

Subscript: Coordinate of the data point;

x_1^k = coordinate 1 of data point k .

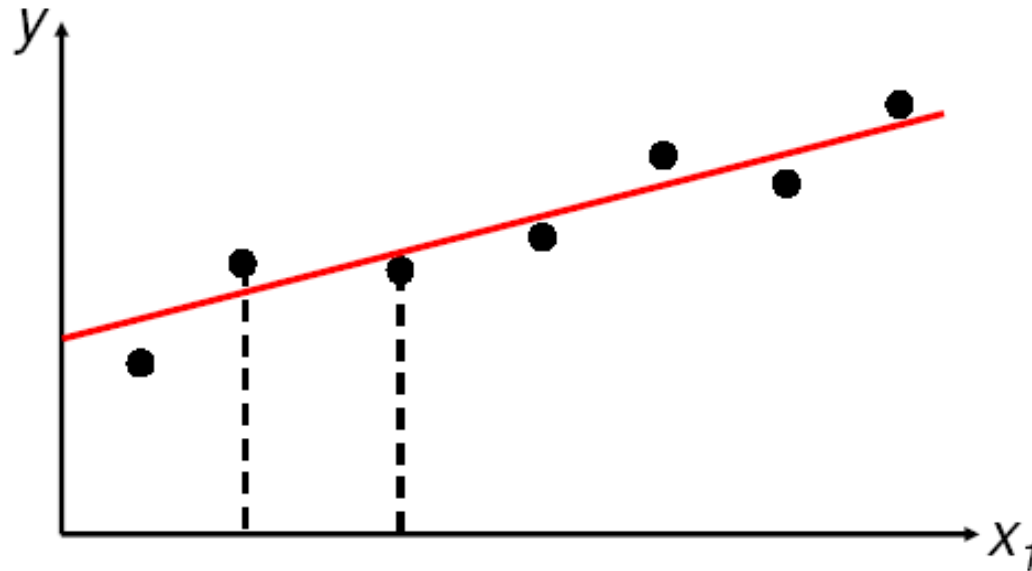
- We have training data $X = \{x_1^k\}$, $k=1, \dots, N$ with corresponding output $Y = \{y^k\}$, $k=1, \dots, N$
- We want to find the parameters that predict the output Y from the data X in a linear fashion: $y^k \approx w_0 + w_1 x_1^k$

Linear Regression



- It is convenient to define an additional “fake” attribute for the input data: $x_0 = 1$
- We want to find the parameters that predict the output Y from the data X in a linear fashion: $y^k \approx w_0 x_0^k + w_1 x_1^k$

More Convenient Notations



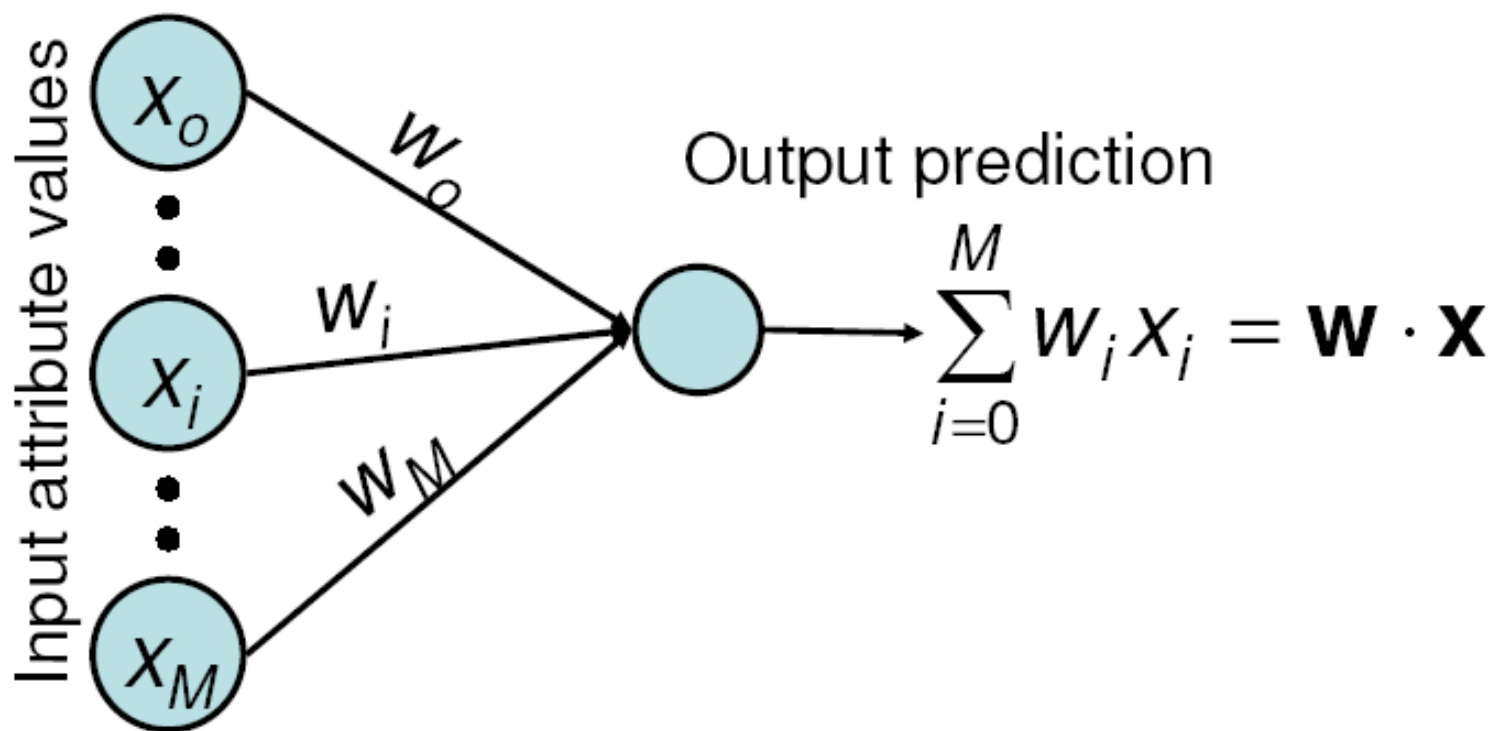
- Vector of attributes for each training data point:

$$\mathbf{x}^k = [x_0^k, \dots, x_M^k]$$

- We seek a vector of parameters: $\mathbf{w} = [w_0, \dots, w_M]$ such that we have a linear relation between prediction Y and attributes X :

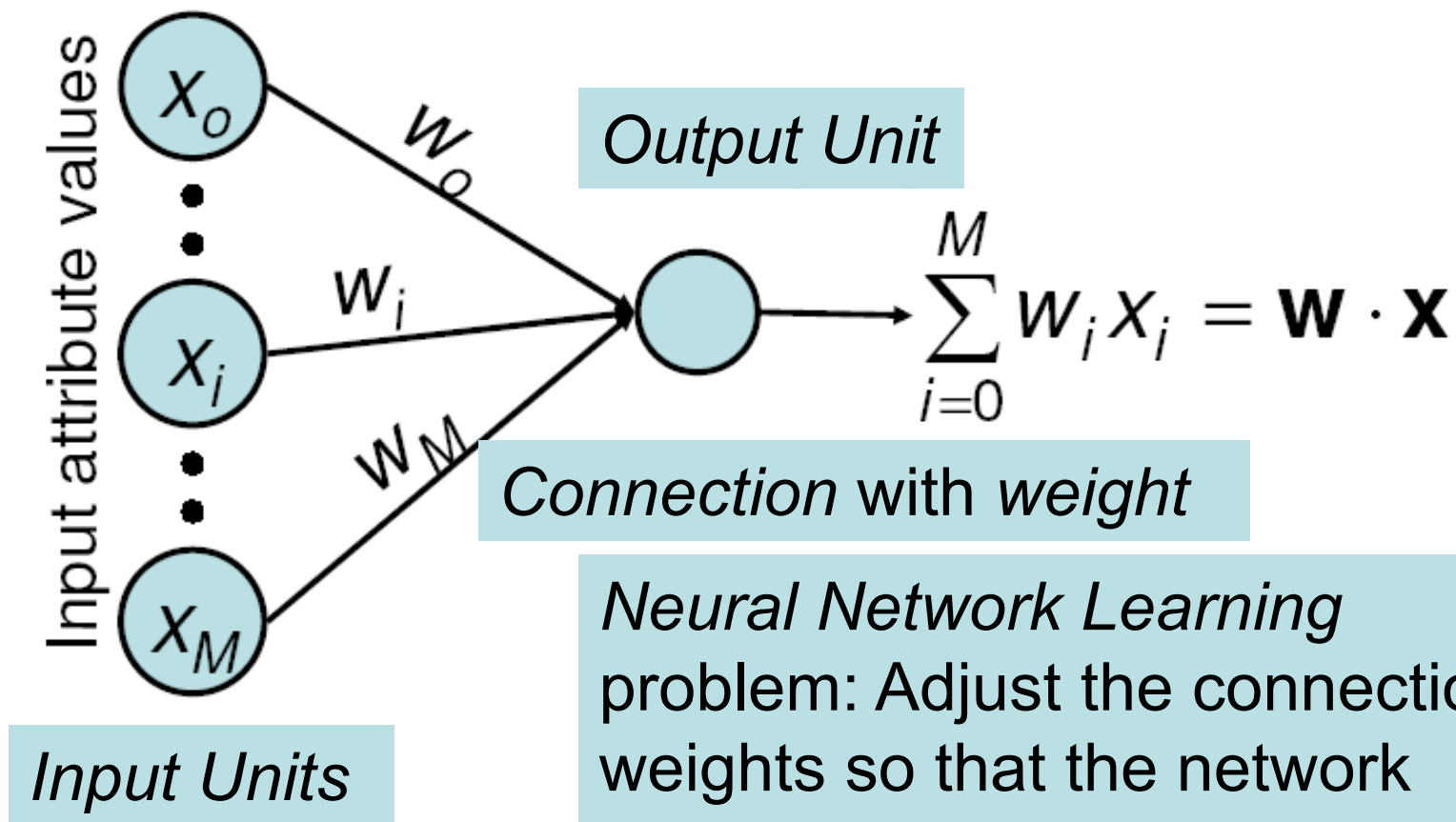
$$y^k \approx w_0 x_0^k + w_1 x_1^k + \dots + w_M x_M^k = \sum_{i=0}^M w_i x_i^k = \mathbf{w} \cdot \mathbf{x}^k$$

Neural Network: Linear Perceptron



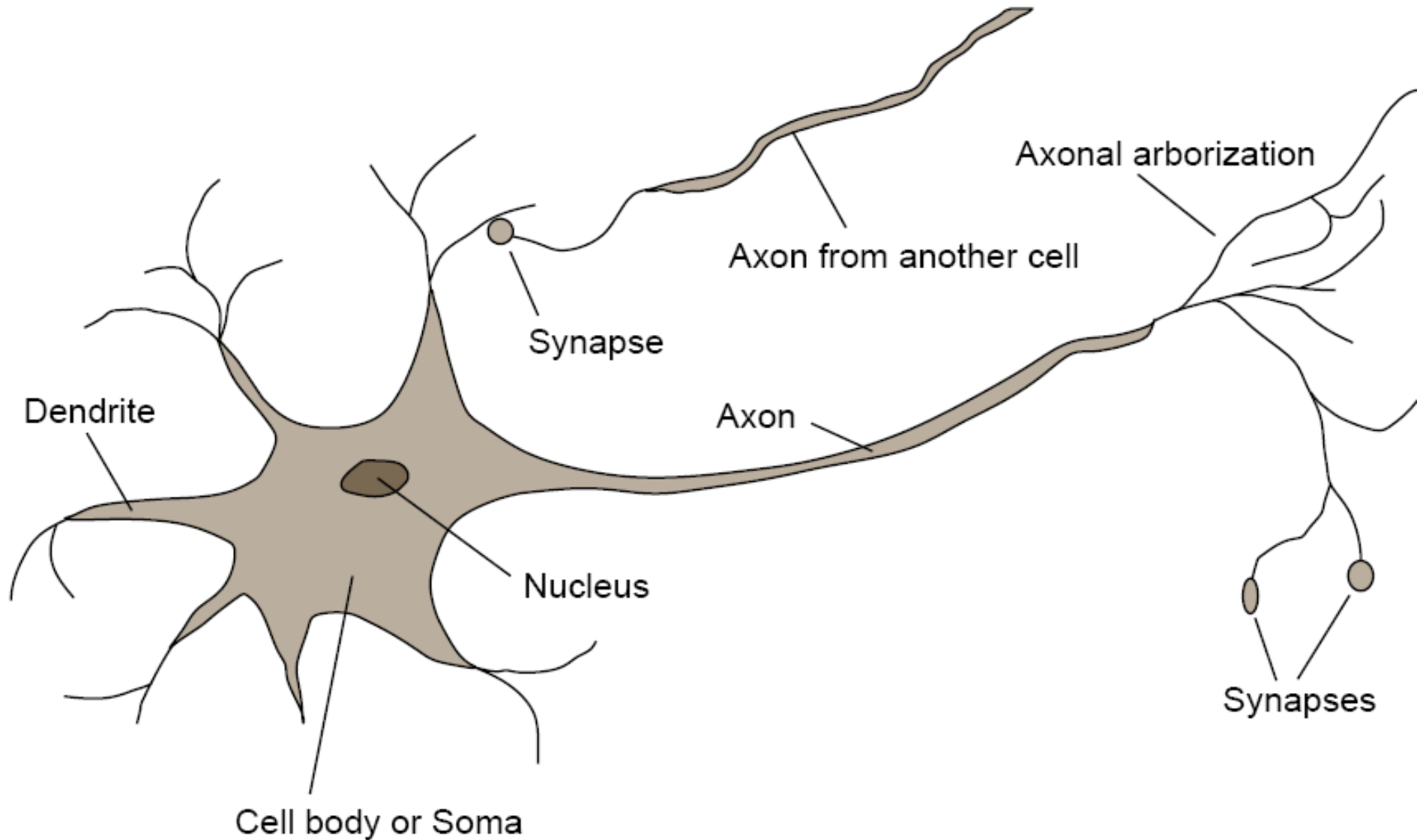
Neural Network: Linear Perceptron

Note: This input unit corresponds to the “fake” attribute $x_0 = 1$.
Called the *bias*

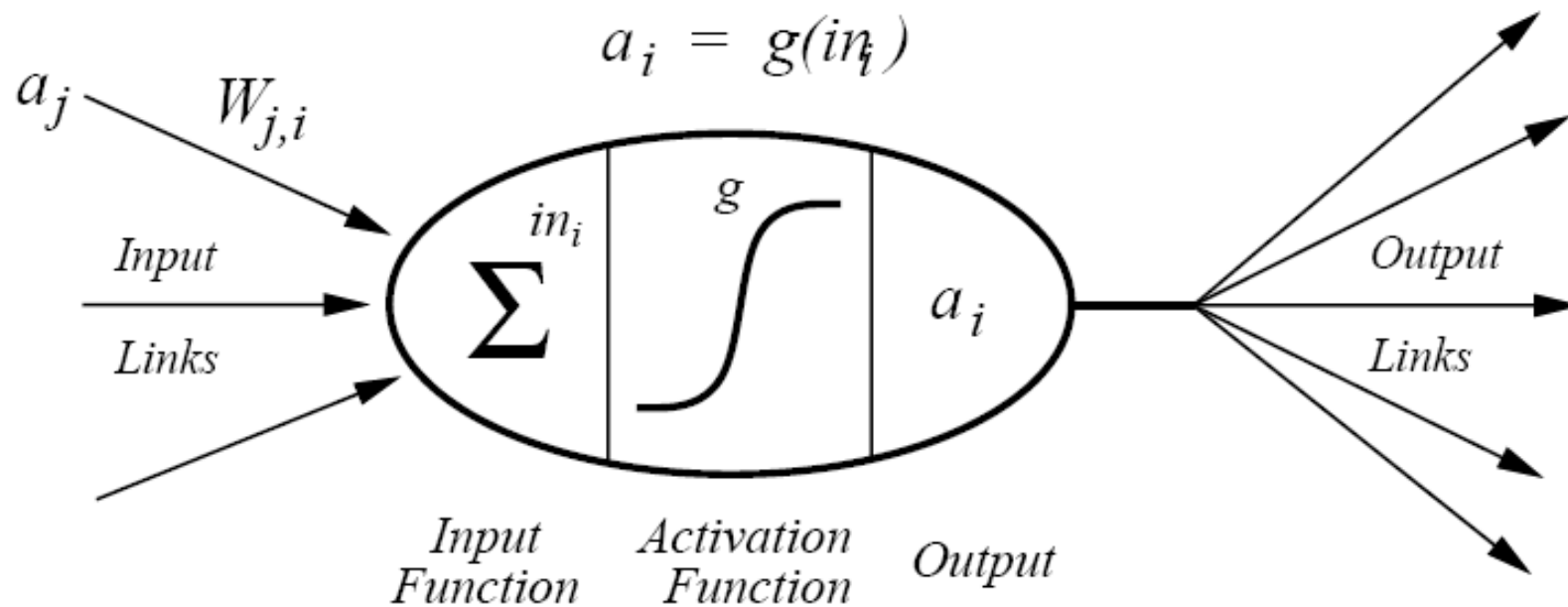


Neural Network Learning
problem: Adjust the connection weights so that the network generates the correct prediction on the training data.

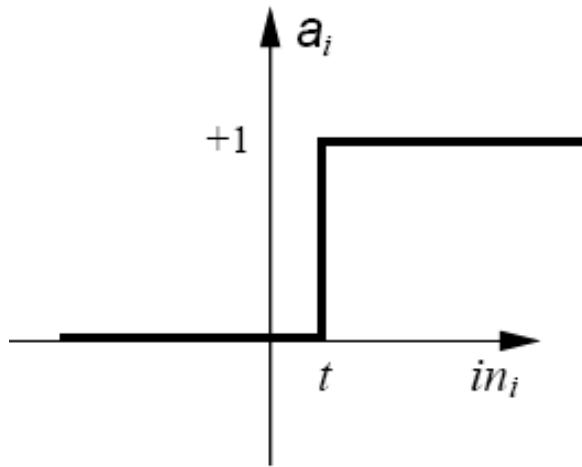
Perceptron: A Mathematical Model of a Neuron



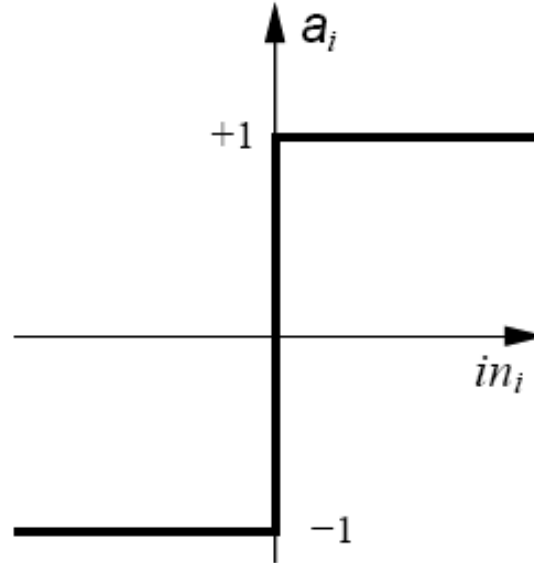
A Perceptron: The General Case



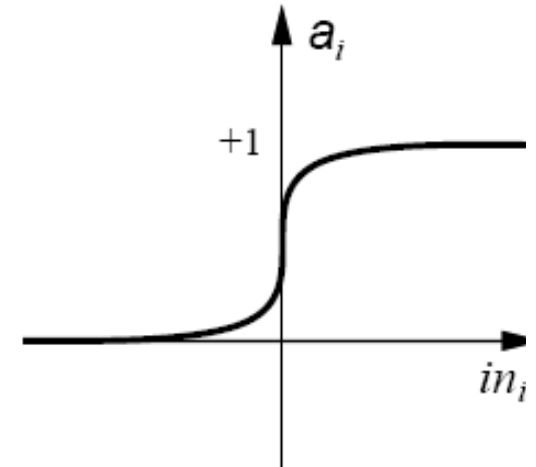
Commonly-Used Activation Functions



(a) Step function



(b) Sign function



(c) Sigmoid function

The Perceptron Training Algorithm

```
function NEURAL-NETWORK-LEARNING(examples) returns network  
  
  network  $\leftarrow$  a network with randomly assigned weights  
  repeat  
    for each e in examples do  
      O  $\leftarrow$  NEURAL-NETWORK-OUTPUT(network, e)  
      T  $\leftarrow$  the observed output values from e  
      update the weights in network based on e, O, and T  
    end  
  until all examples correctly predicted or stopping criterion is reached  
  return network
```

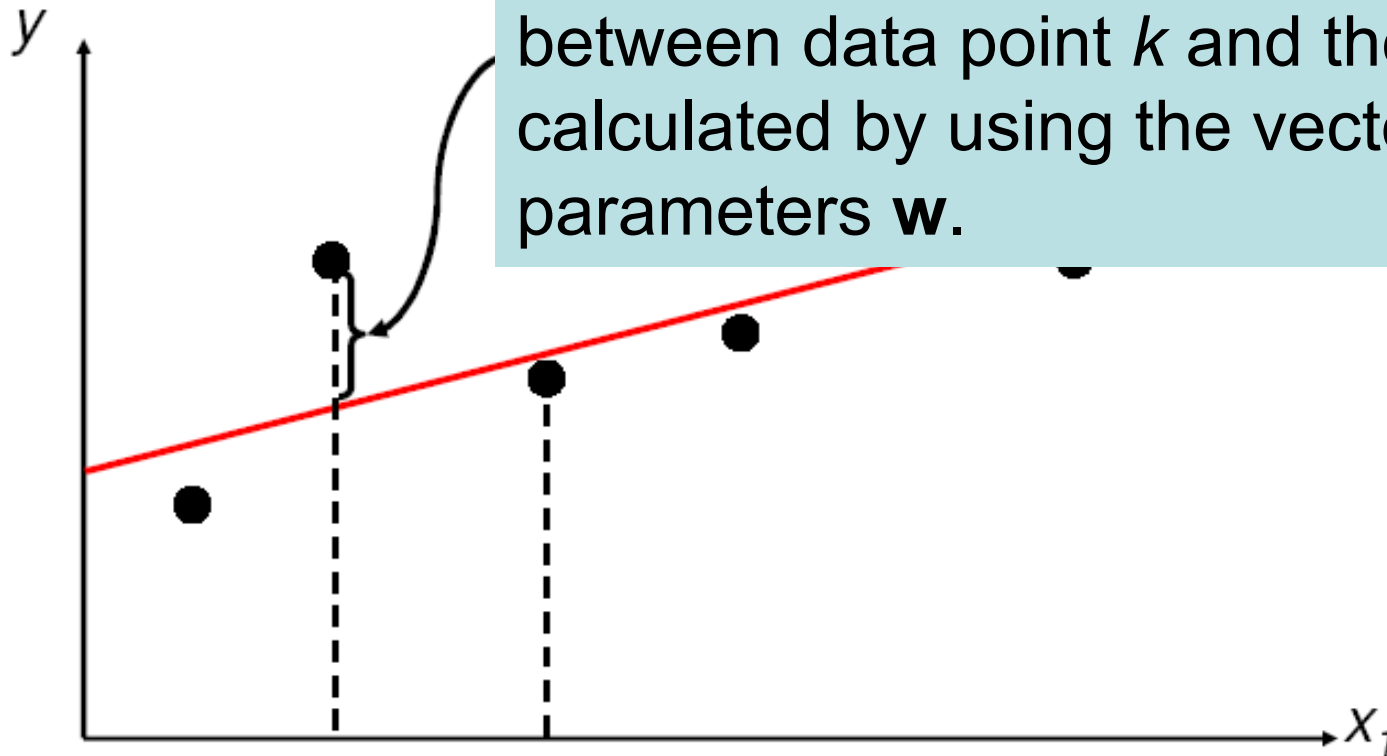
But how do we update the weights?

Linear Regression: Gradient Descent

- We seek a vector of parameters: $\mathbf{w} = [w_0, \dots, w_M]$ that minimizes the error between the prediction Y and the data X :

$$\begin{aligned} E &= \sum_{k=1}^N \left(y^k - (w_0 x_0^k + w_1 x_1^k + \dots + w_M x_M^k) \right)^2 \\ &= \sum_{k=1}^N \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right)^2 \\ &= \sum_{k=1}^N \delta_k^2 \quad \delta_k = y^k - \mathbf{w} \cdot \mathbf{x}^k \end{aligned}$$

Linear δ_k is the error between the input \mathbf{x} and the prediction y at data point k .
Graphically, it is the “vertical” distance between data point k and the prediction calculated by using the vector of linear parameters \mathbf{w} .



$$x_M^k))^2$$

$$= \sum_{k=1}^N \delta_k^2 \quad \delta_k = y^k - \mathbf{w} \cdot \mathbf{x}^k$$

Gradient Descent

- The minimum of E is reached when the derivatives with respect to each of the parameters w_i is zero:

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= -2 \sum_{k=1}^N \left(y^k - (w_o x_o^k + w_1 x_1^k + \dots + w_M x_M^k) \right) x_i^k \\ &= -2 \sum_{k=1}^N \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right) x_i^k \\ &= -2 \sum_{k=1}^N \delta_k x_i^k\end{aligned}$$

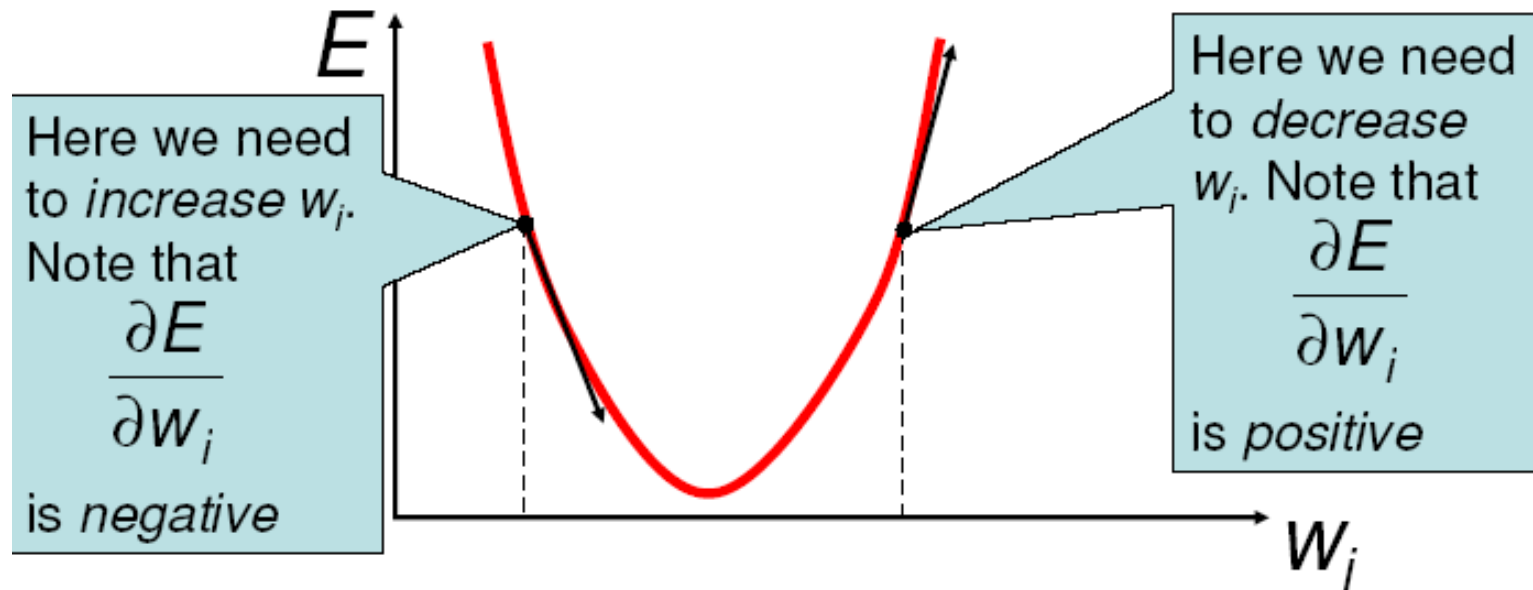
Gradient Descent

- The minimum of E is reached when the derivatives with respect to each of the parameters w_i is zero:

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= -2 \sum_{k=1}^N \left(y^k - (w_o x_o^k + w_1 x_1^k + \dots + w_M x_M^k) \right) x_i^k \\ &= -2 \sum_{k=1}^N \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right) x_i^k \\ &= -2 \sum_{k=1}^N \delta_k x_i^k\end{aligned}$$

Note that the contribution of training data element number k to the overall gradient is $-\delta_k x_i^k$

Gradient Descent Update Rule



- Update rule: Move in the direction opposite to the gradient direction

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

Perceptron Training

- Given input training data x^k with corresponding value y^k
 1. Compute error:

$$\delta_k \leftarrow y^k - \mathbf{w} \cdot \mathbf{x}^k$$

2. Update NN weights:

$$w_i \leftarrow w_i + \alpha \delta_k x_i^k$$

Perceptron Training

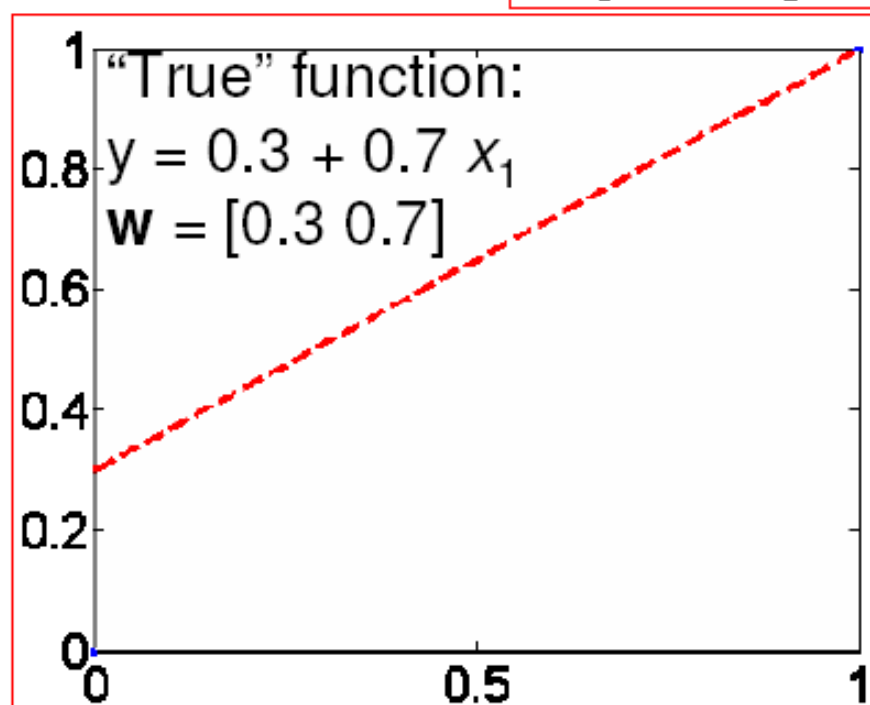
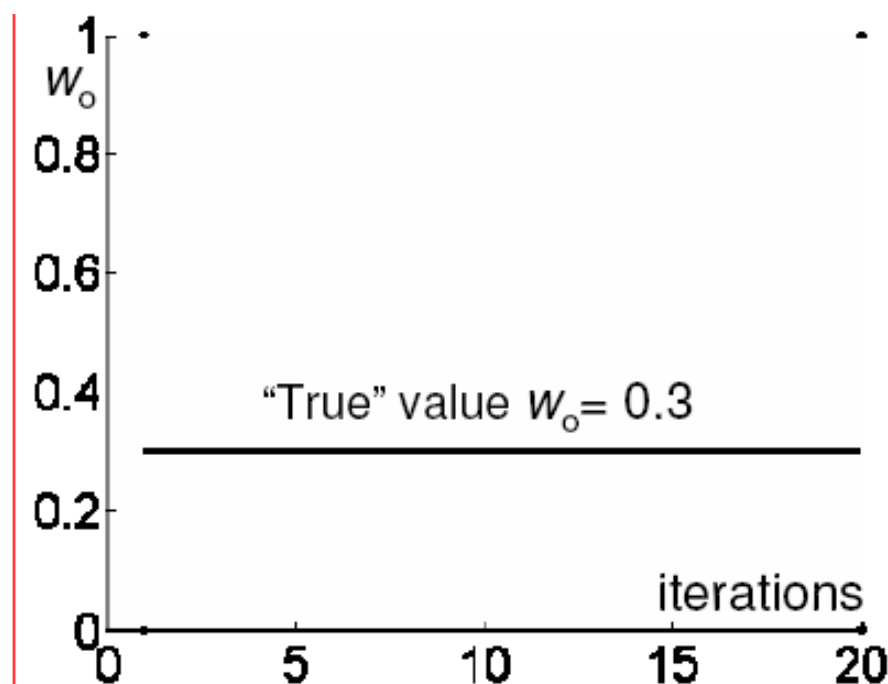
α is the learning rate.

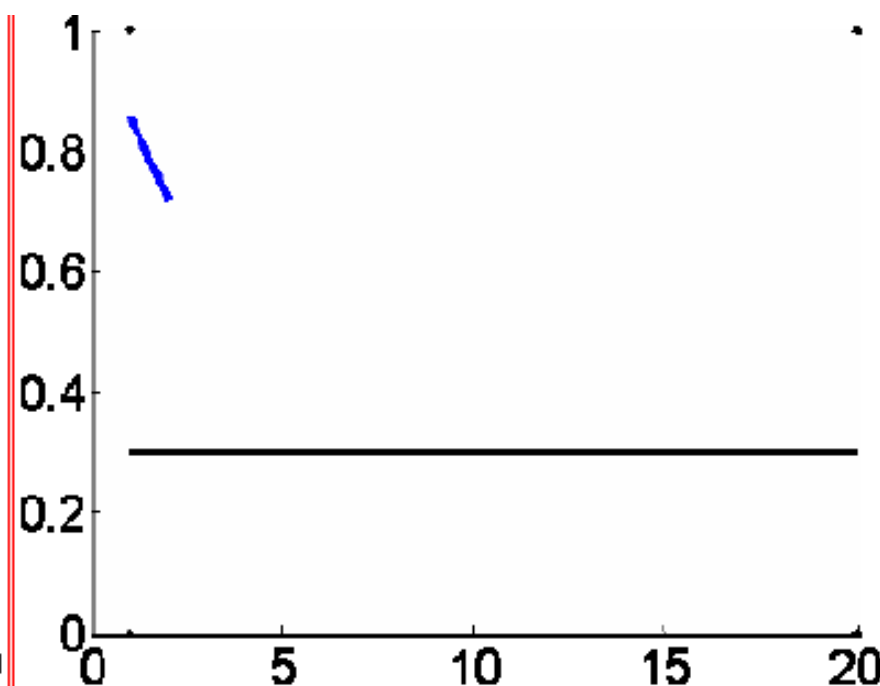
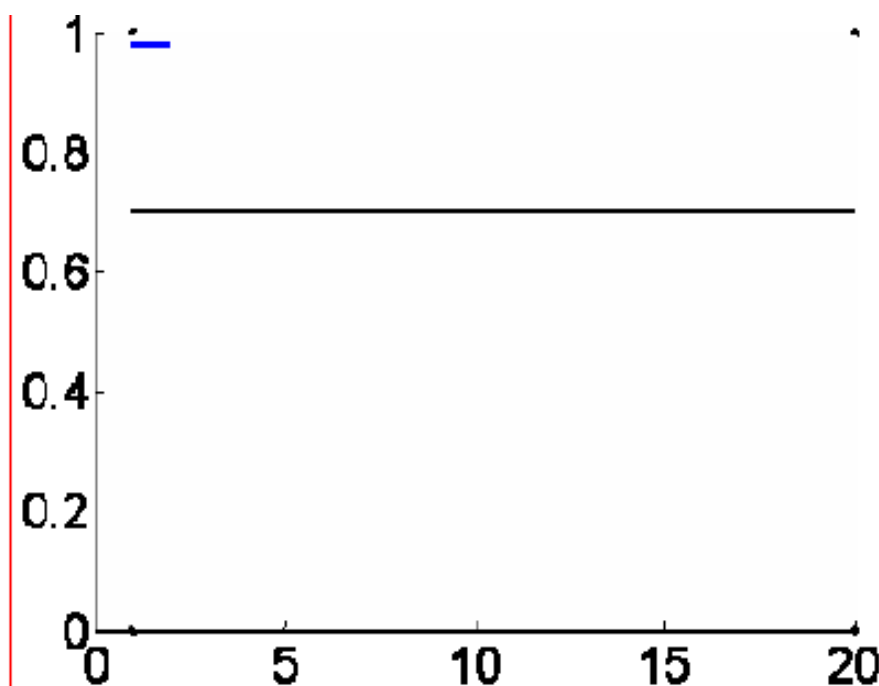
α too small: May converge slowly and may need a lot of training examples

α too large: May change \mathbf{w} too quickly and spend a long time oscillating around the minimum.

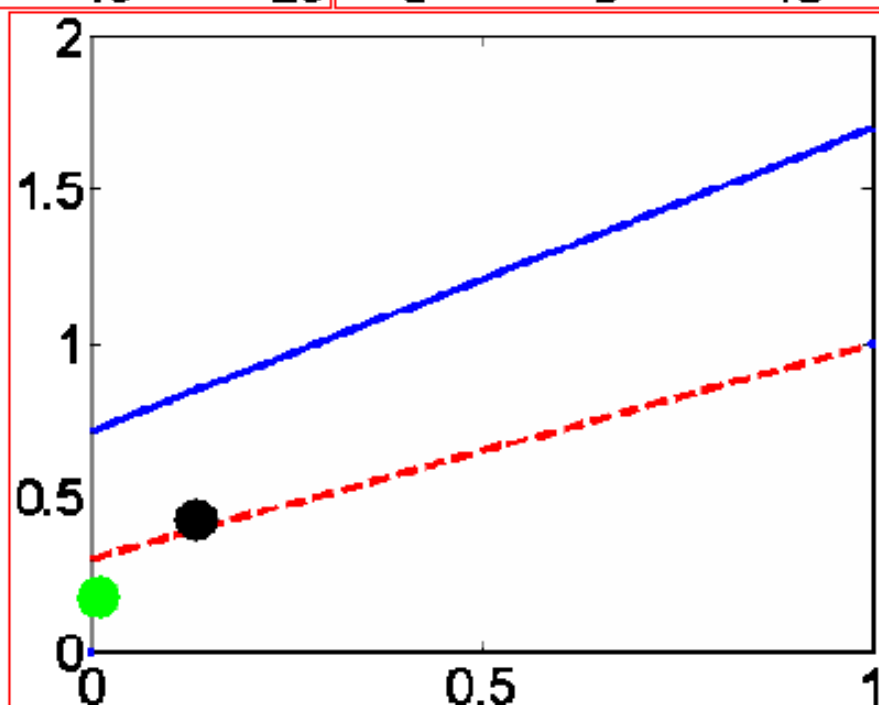
2. Update NN weights:

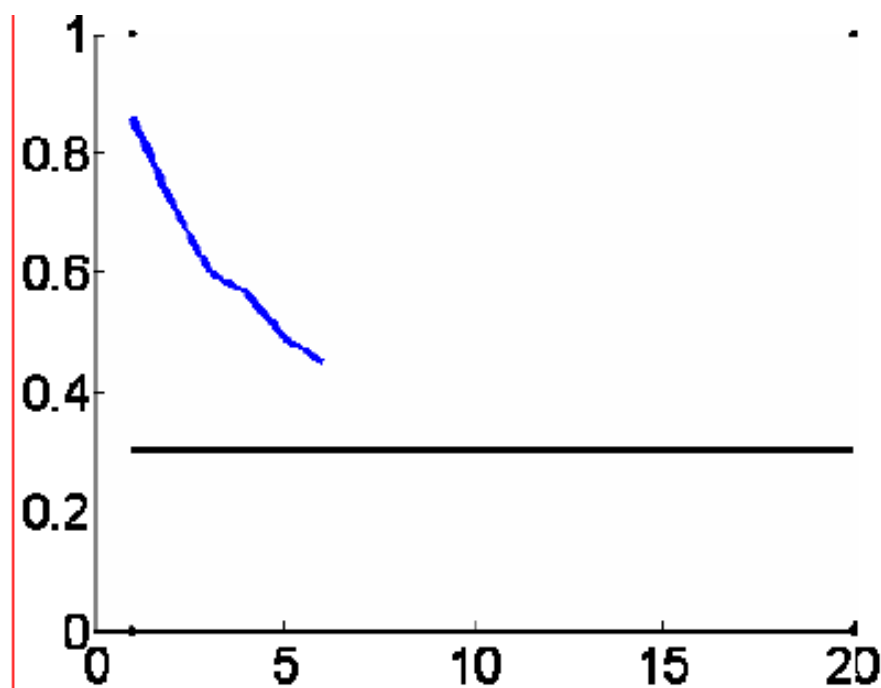
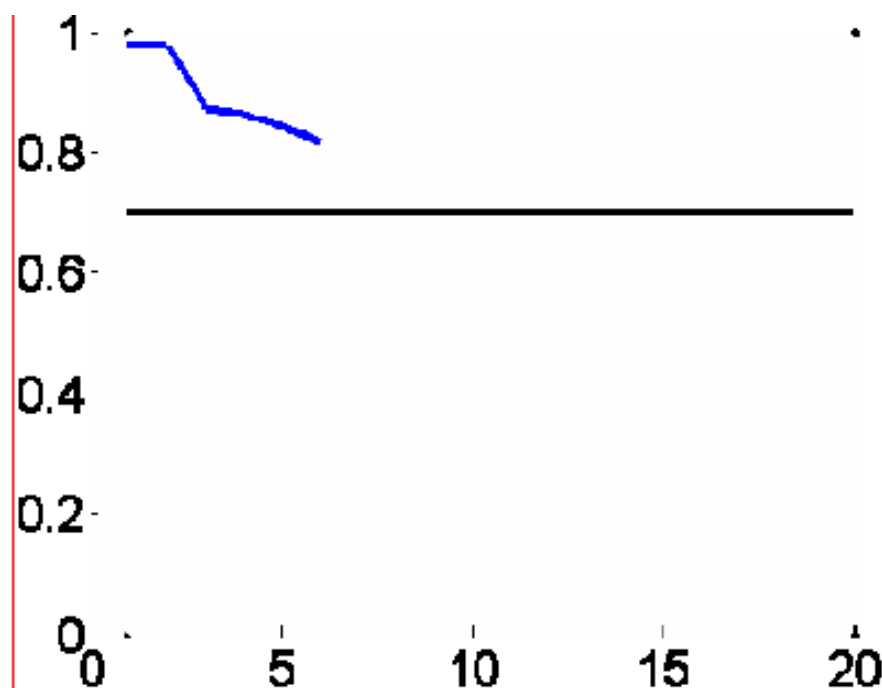
$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \alpha \delta_k x_i^k$$



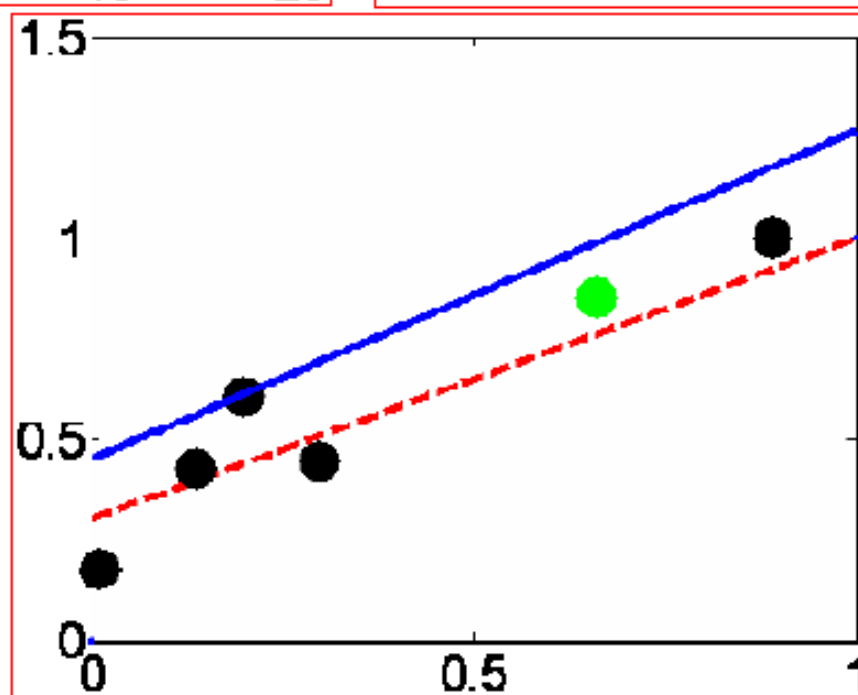


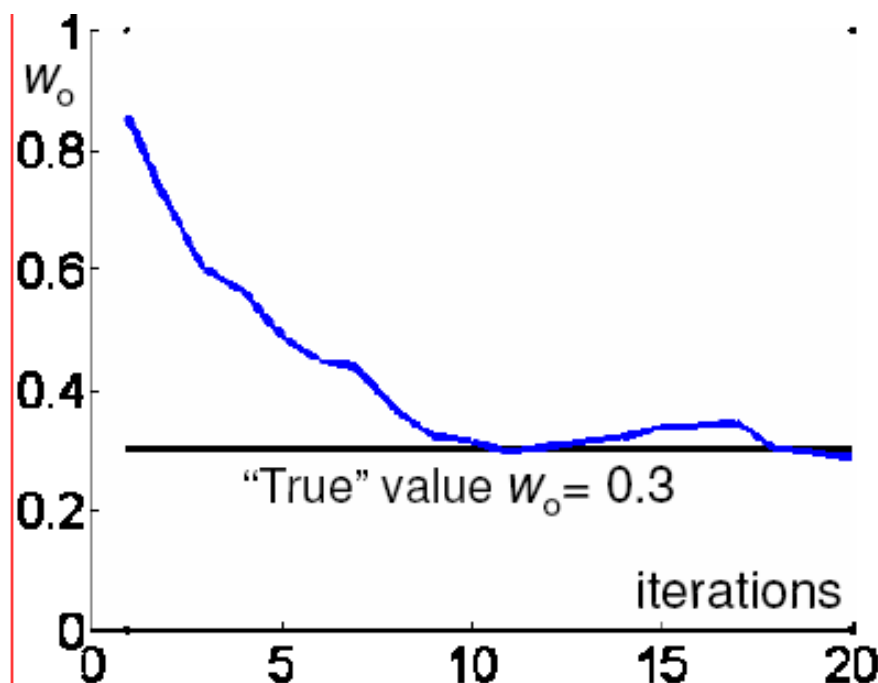
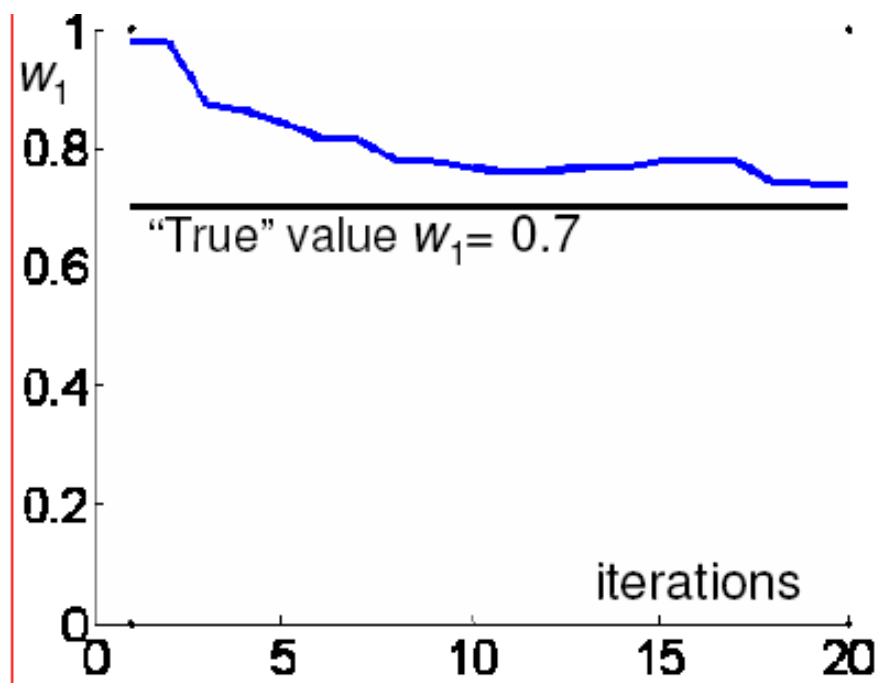
After 2 iterations
(2 training points)



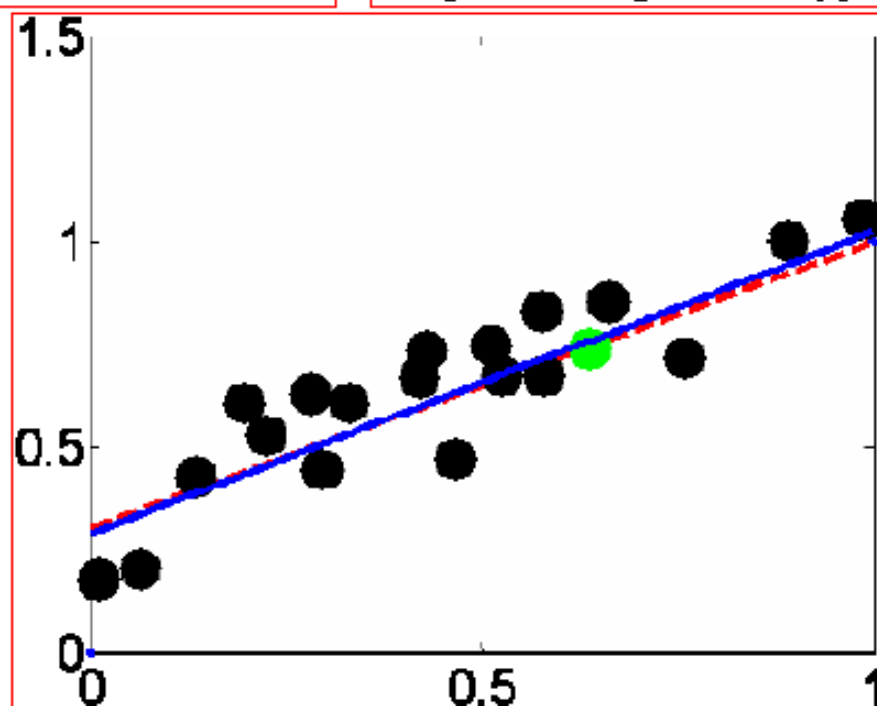


After 6 iterations
(6 training points)

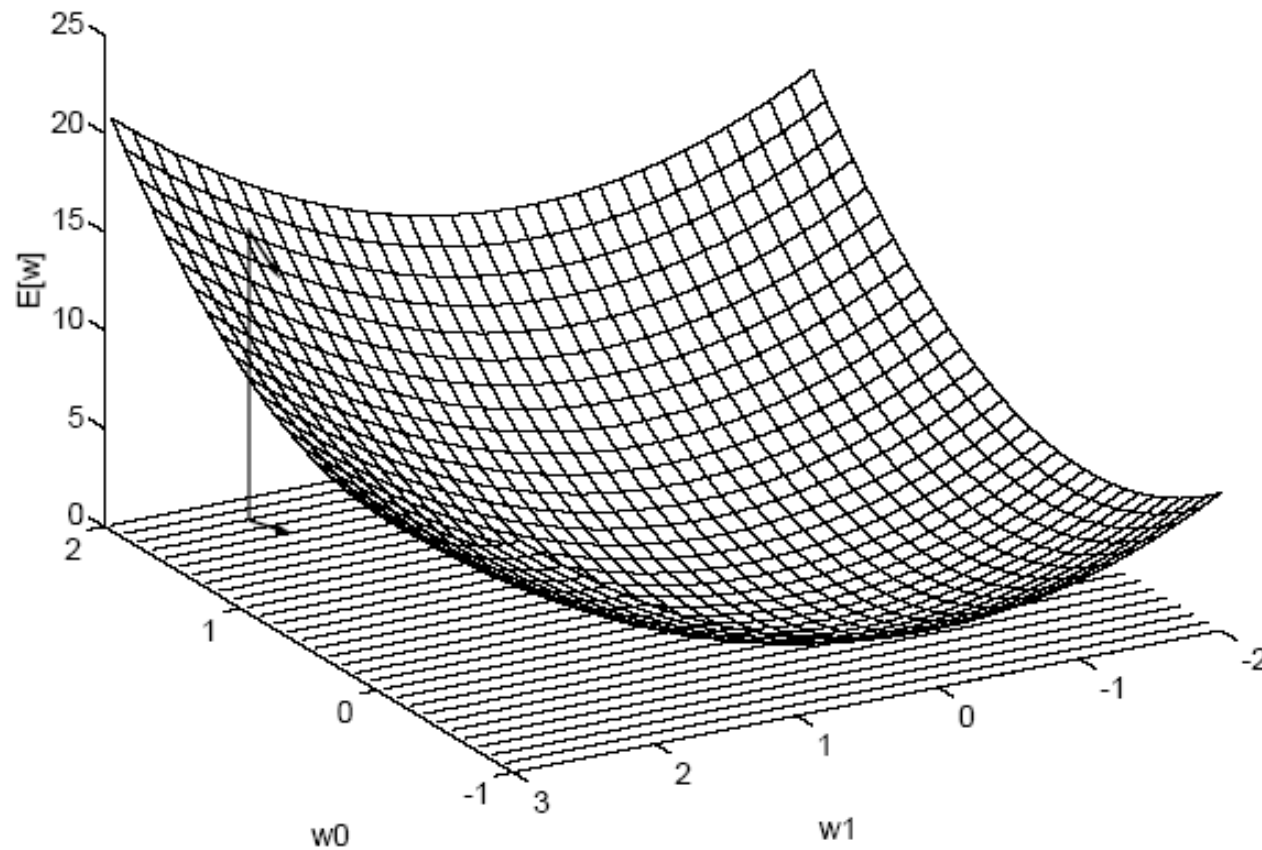




After 20 iterations
(20 training points)



Gradient Descent Through Weight Space

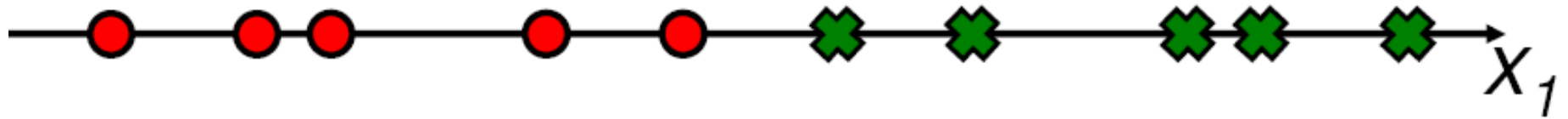


Perceptrons: Remarks

- Update has many names: delta rule, gradient rule, least mean squares (LMS) rule.....
- Update is ***guaranteed*** to converge to the best linear fit (global minimum of E)
- Of course, there are other ways of solving the linear regression problem by using linear algebra techniques. It boils down to a simple matrix inversion (not shown here).

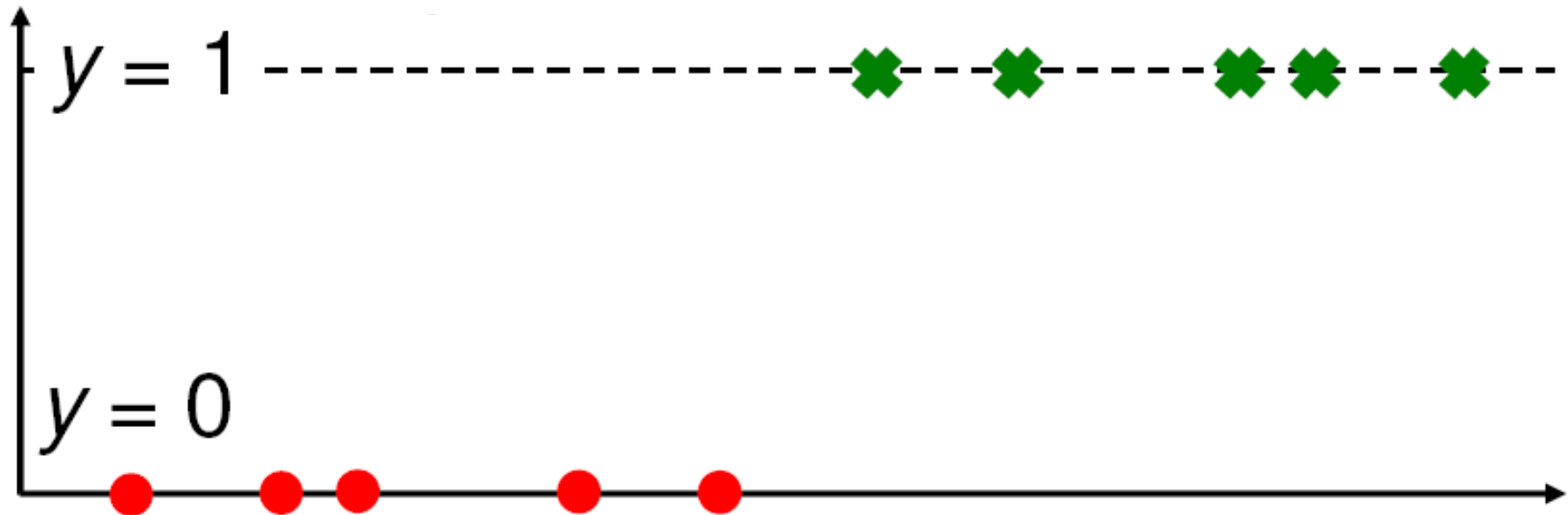
A Simple Classification Problem

Training data:



- Suppose that we have one attribute x_1
- Suppose that the data is in two classes (red dots and green dots)
- Given an input value x_1 , we wish to predict the most likely class (note: Same problem as the one we solved with decision trees).

A Simple Classification Problem

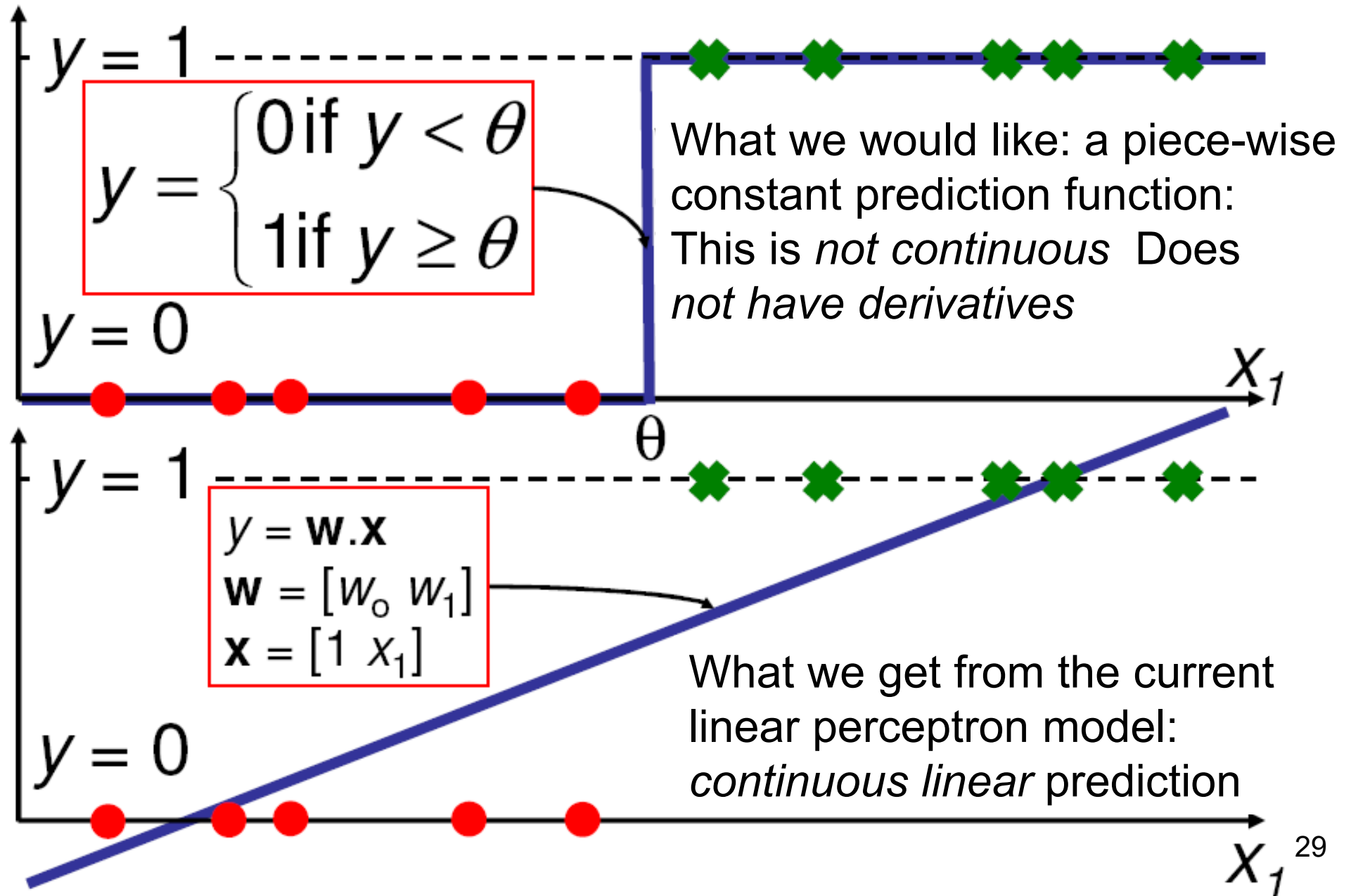


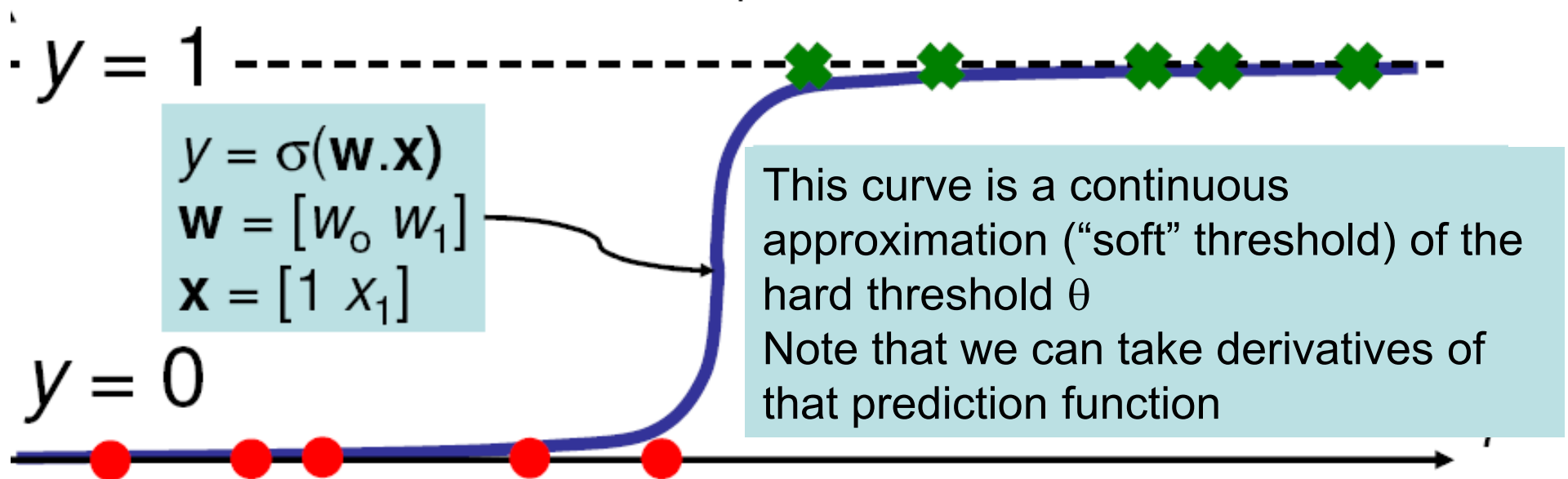
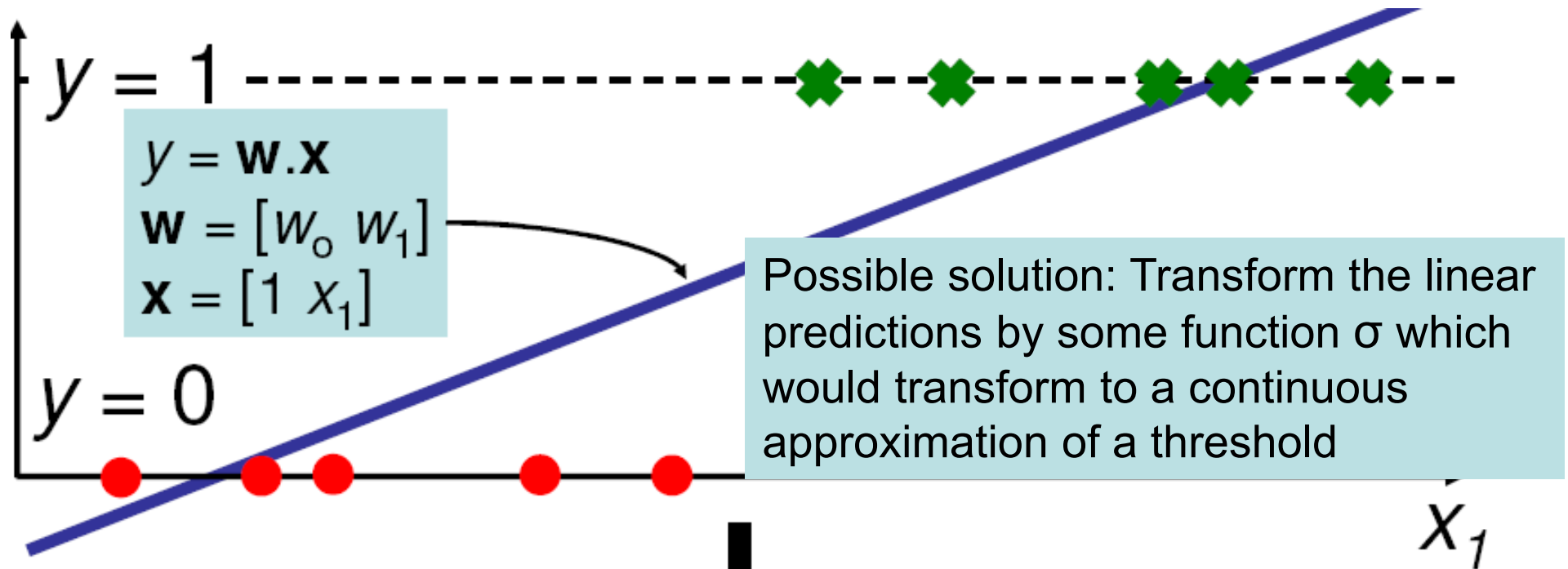
- We could convert it to a problem similar to the previous one by defining an output value y

$$y = \begin{cases} 0 & \text{if in red class} \\ 1 & \text{if in green class} \end{cases}$$

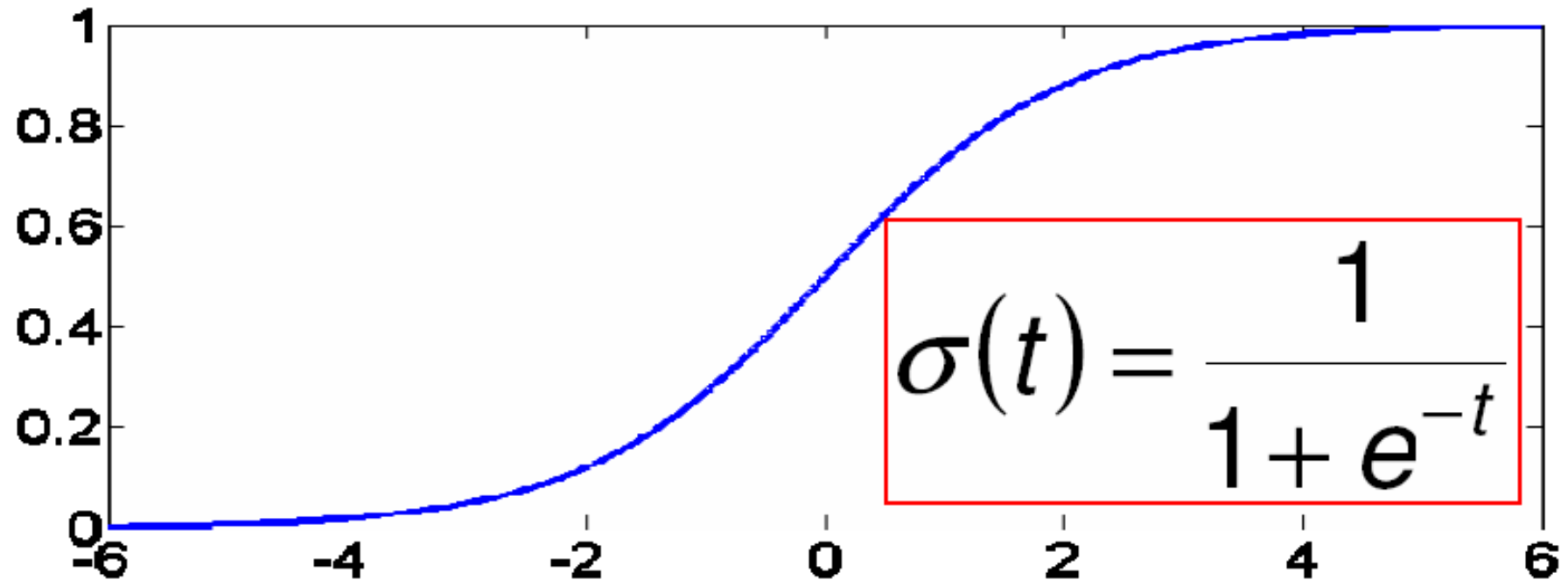
- The problem now is to learn a mapping between the attribute x_1 of the training examples and their corresponding class output y

A Simple Classification Problem



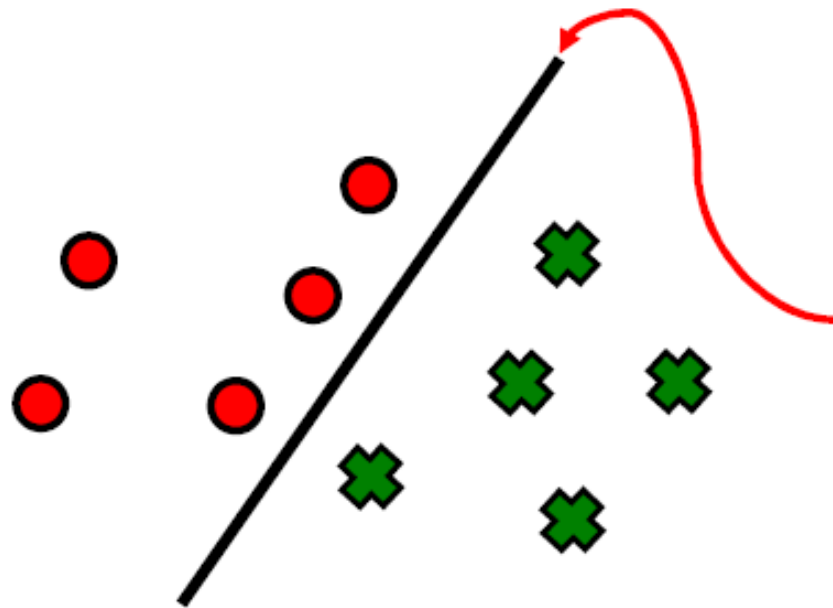


The Sigmoid Function



- It is important to remember that:
 - σ is the activation function
 - σ is smooth and has a derivative $\sigma' = \sigma(1 - \sigma)$
 - σ approximates a hard threshold function at $x = 0$

Generalization to M Attributes



A linear separation is parameterized like a line:

$$\sum_{i=0}^M \mathbf{w}_i \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x} = 0$$

- Two classes are **linearly separable** if they can be separated by a linear combination of the attributes:
 - Threshold in 1-d
 - Line in 2-d
 - Plane in 3-d
 - Hyperplane in M -d

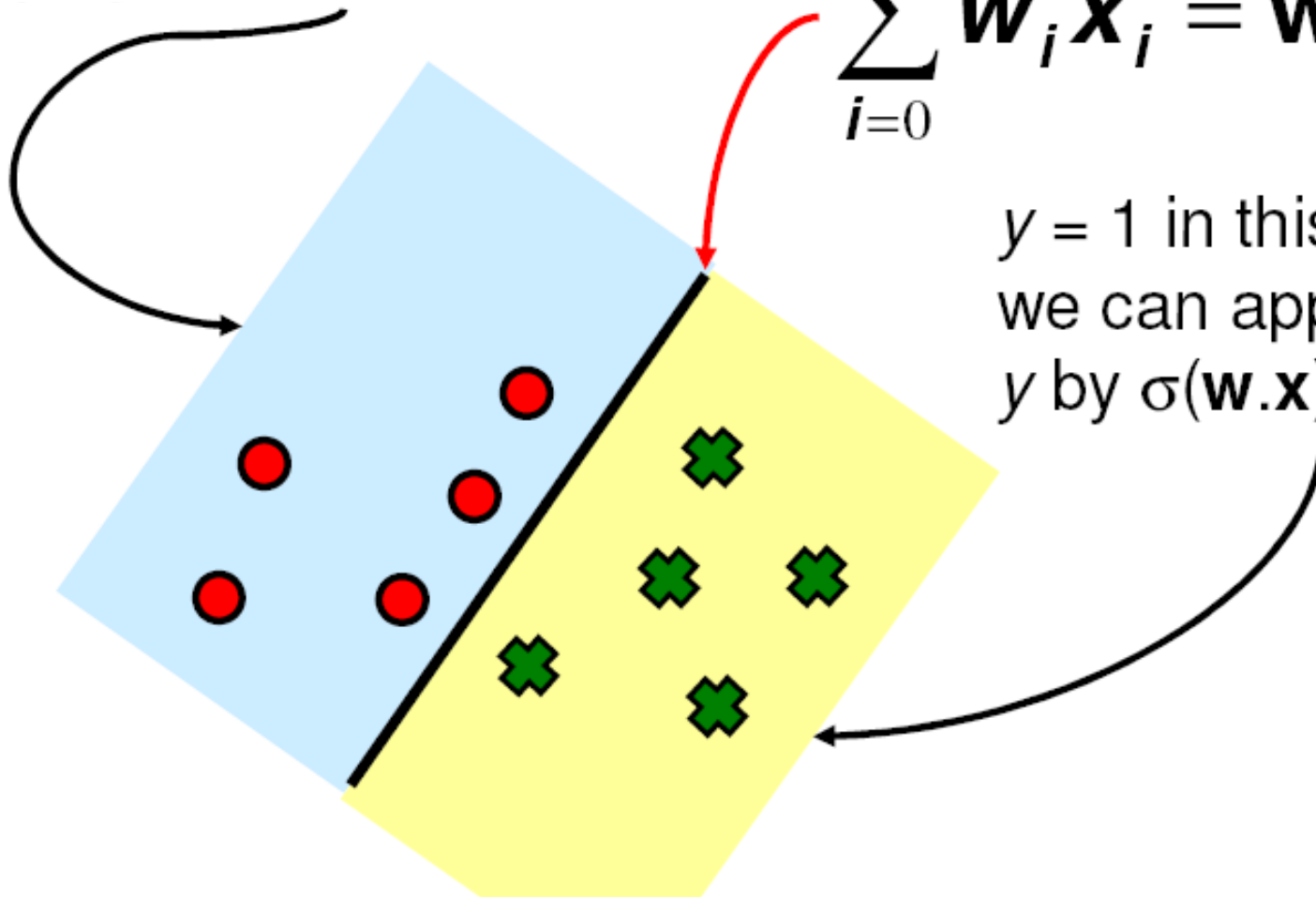
Generalization to M Attributes

$y = 0$ in this region,
we can approximate
 y by $\sigma(\mathbf{w} \cdot \mathbf{x}) \approx 0$

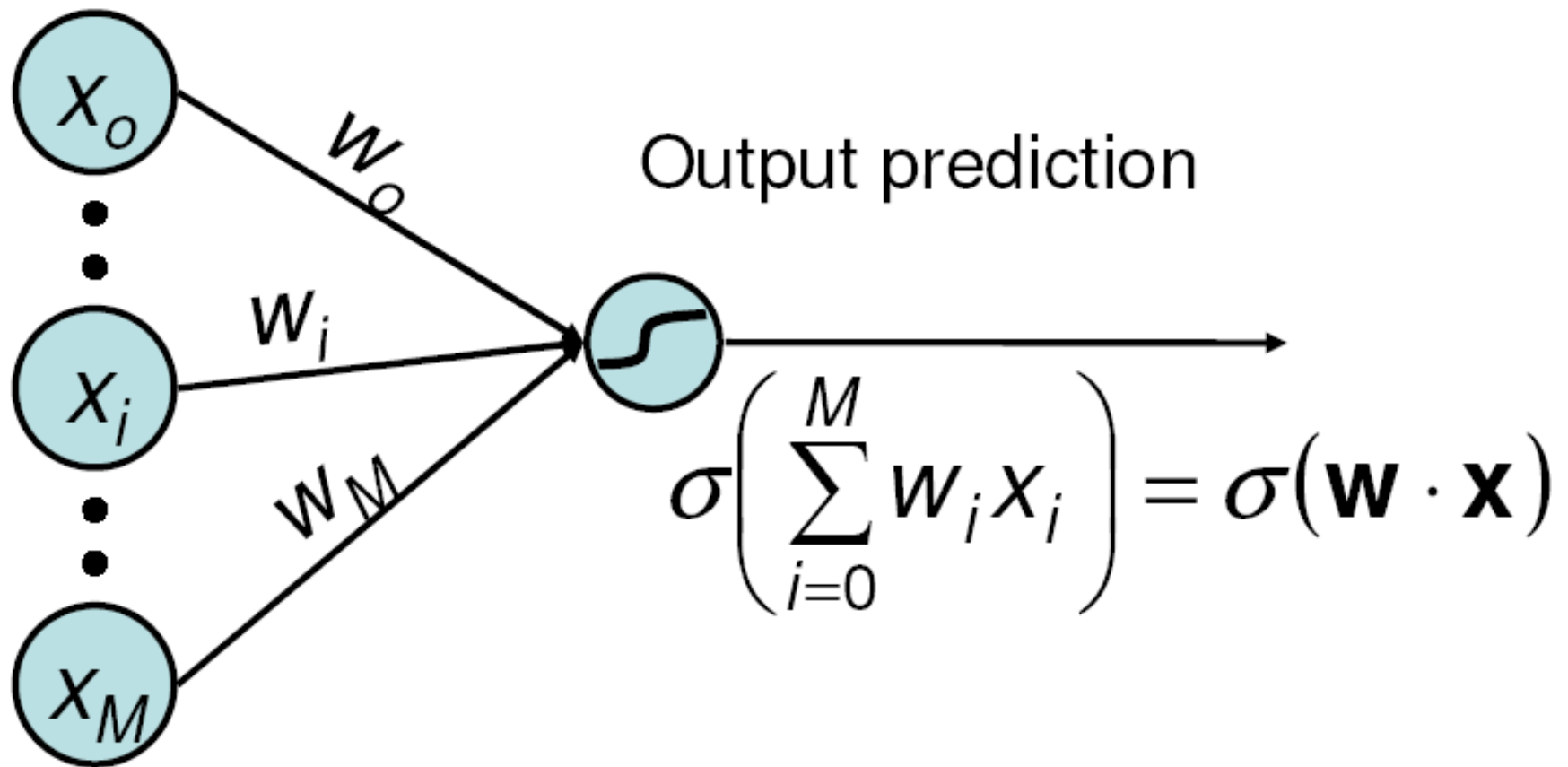
A linear separation is
parameterized like a line:

$$\sum_{i=0}^M \mathbf{w}_i \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x} = 0$$

$y = 1$ in this region,
we can approximate
 y by $\sigma(\mathbf{w} \cdot \mathbf{x}) \approx 1$



Perceptron for Classification



Training

- Given input training data x^k with corresponding value y^k

1. Compute error:

$$\delta_k \leftarrow y^k - \sigma(\mathbf{w} \cdot \mathbf{x}^k)$$

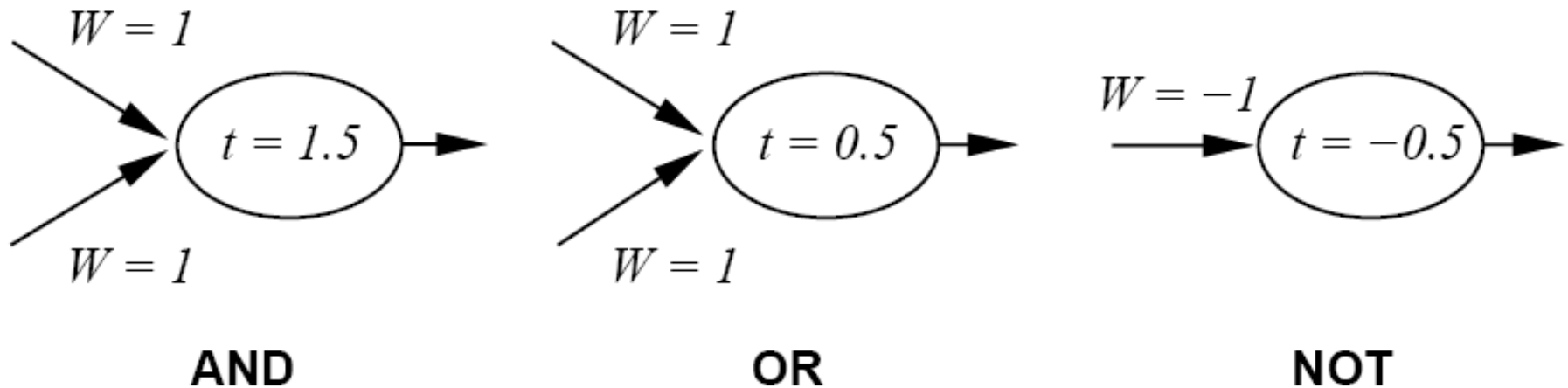
2. Update NN weights:

$$w_i \leftarrow w_i + \alpha \delta_k x_i^k \sigma'(\mathbf{w} \cdot \mathbf{x}^k)$$

Expressiveness of Perceptrons

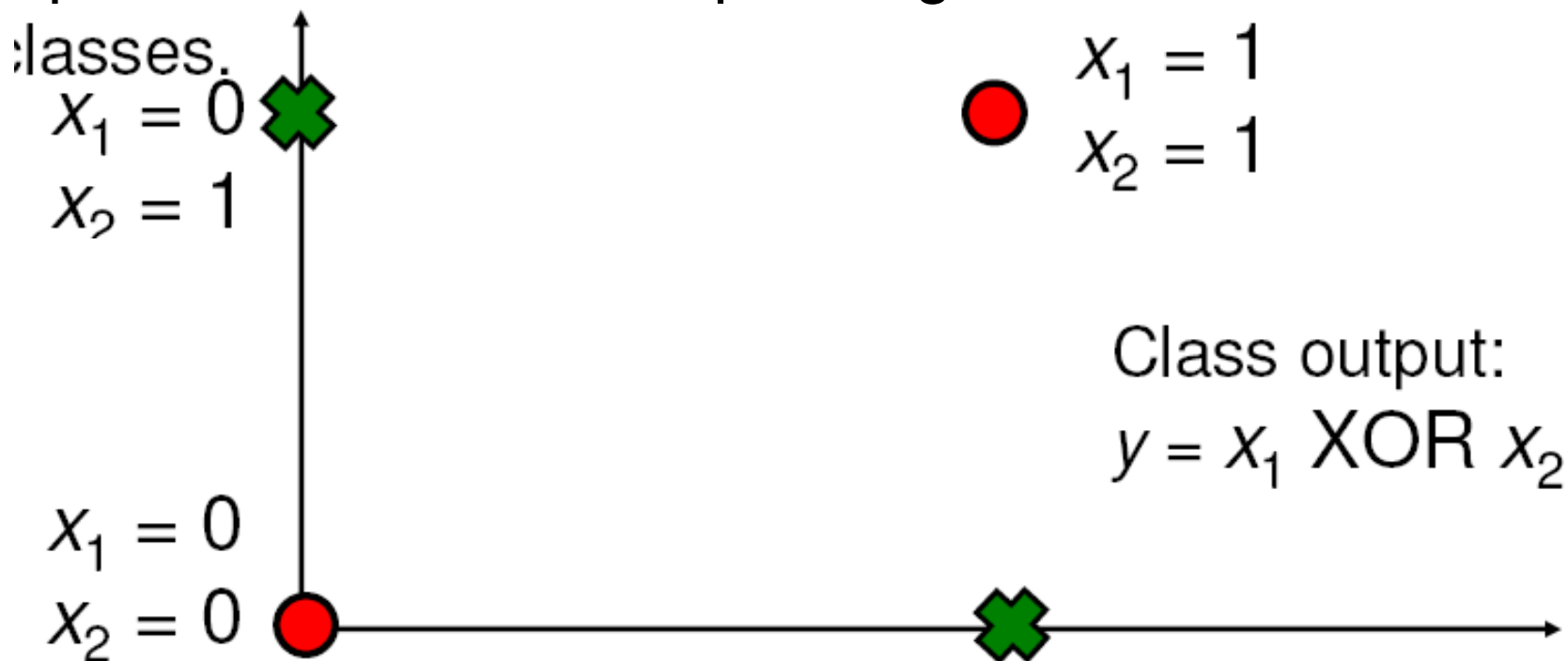
- Can represent any problem in which the decision boundary is *linear*

E.g., Can represent boolean AND, OR, and NOT



Expressiveness of Perceptrons

- But ... *NO* guarantee if the problem is not linearly separable
- Canonical example: Learning the XOR function from example There is no line separating the data in 2 classes.

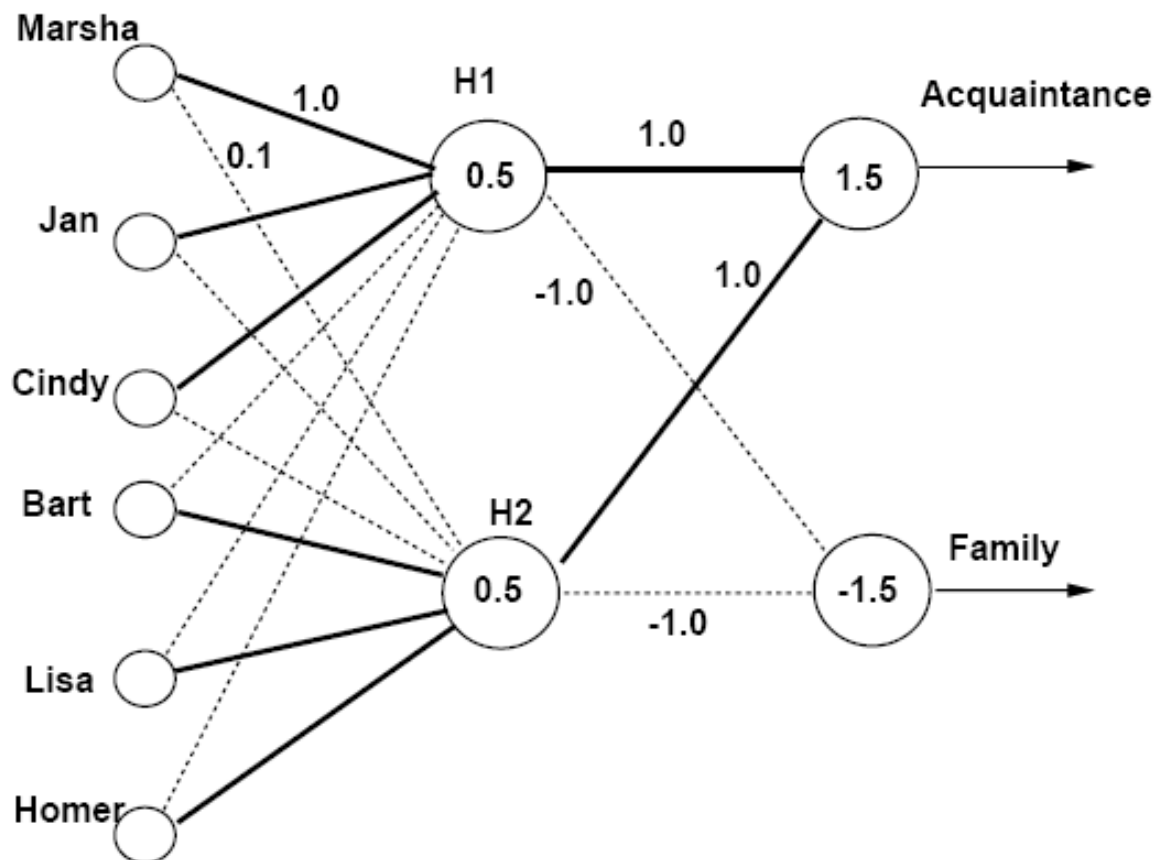


- How do we know whether our data is linearly separable?

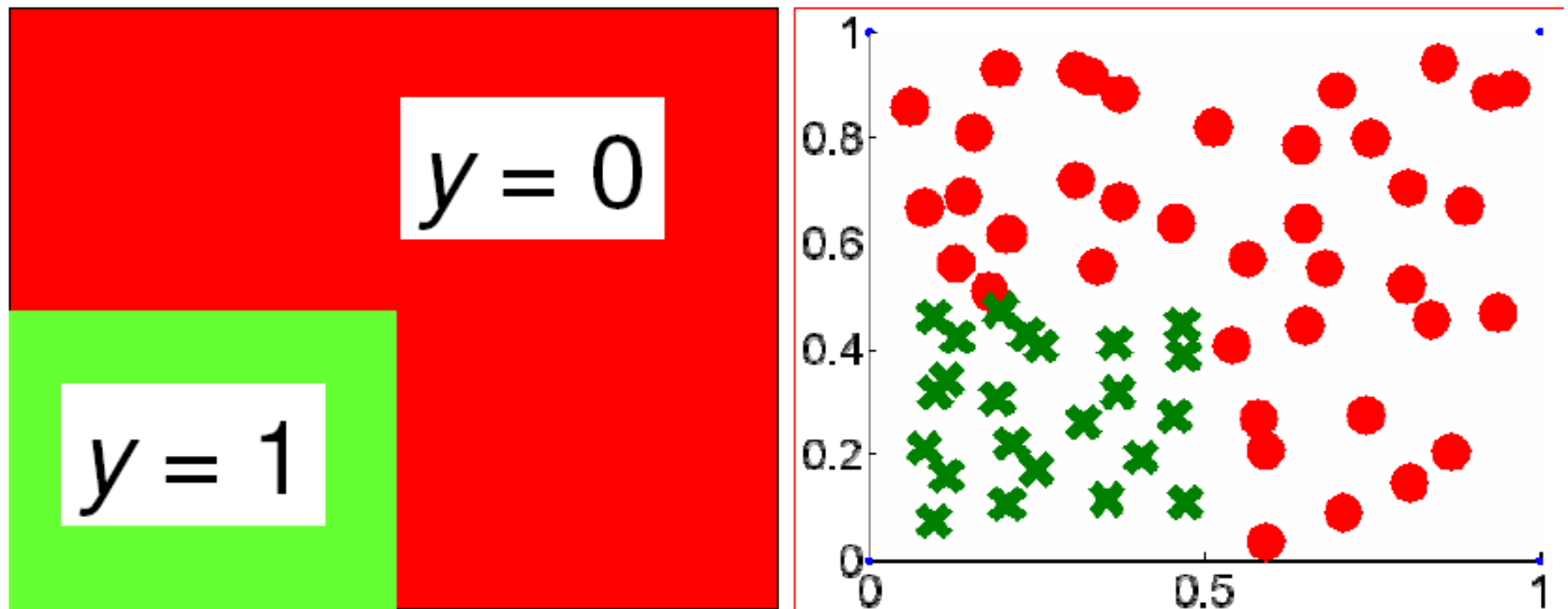
Use **linear programming**

Complex Feedforward Nets for Classification

Feedforward, layered, fully connected

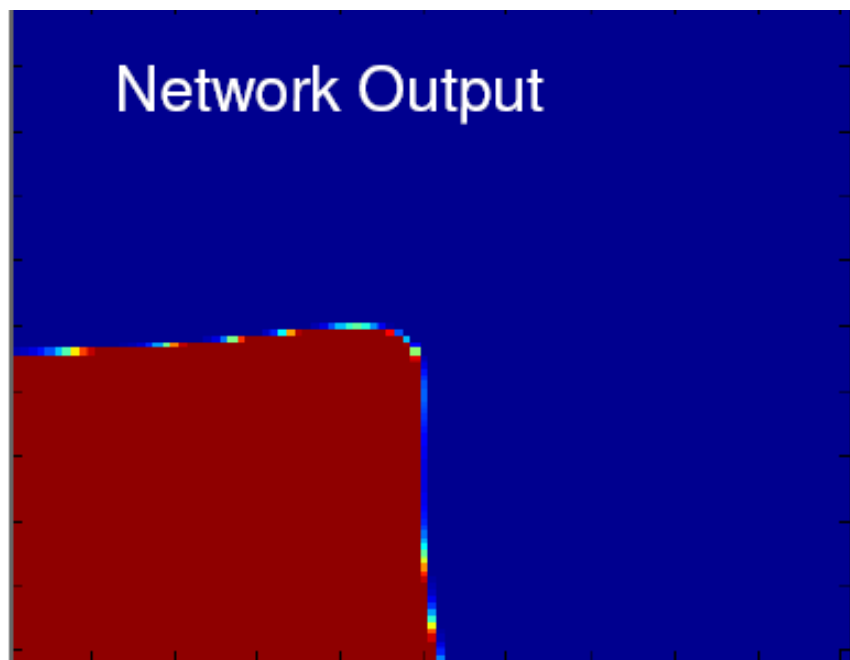


Example

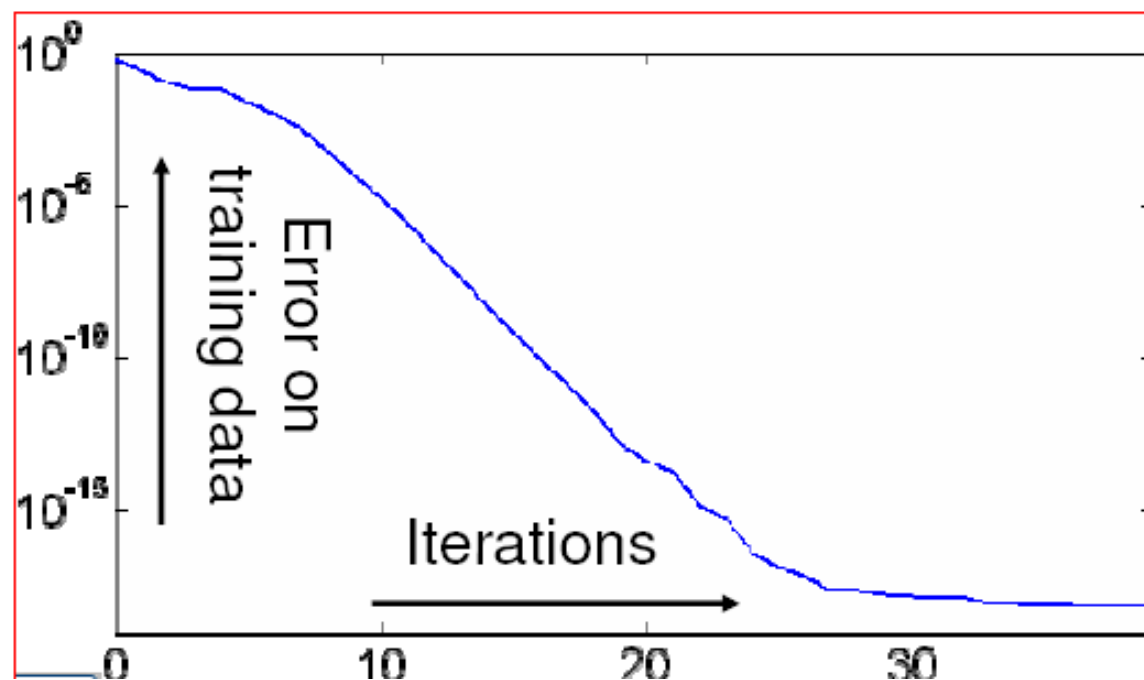
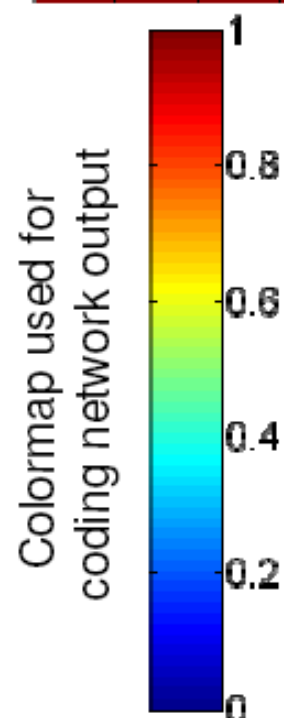
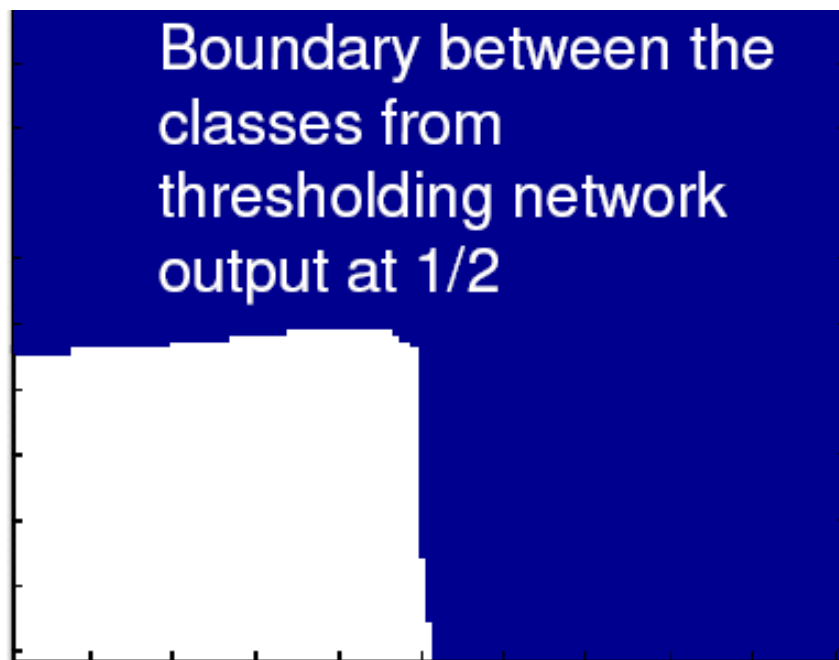


Classes cannot be separated by a linear boundary
Let's try: 2-layer network, 4 hidden units

Network Output



Boundary between the
classes from
thresholding network
output at 1/2



Backpropagation

Note that in the perceptron case, we looked at the output value, compared it to the desired value and changed the weights accordingly.

We want to do something similar in the multi-layer case but one difficulty is to determine the error in the outputs of the hidden units.

Luckily, we can approximate those errors by “backpropagating” the final output error.

Backpropagation Procedure

Initialize weights. Until performance is satisfactory*,

1. Present all training instances. For each one,
 - (a) Calculate actual output. (forward pass)
 - (b) Compute the weight changes. (backward pass)
 - i. Calculate error at output nodes. Compute adjustment to weights from hidden layer to output layer accordingly.
 - ii. Calculate error at hidden layer. Compute adjustment to weights from initial layer to hidden layer accordingly.
 - iii. Change the weights.

The Backpropagation Algorithm

Initialize the weights to small numbers. Until satisfied, do

For each training example, do

Input the training example to the network and compute outputs

For each output unit k

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

For each hidden unit h

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

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

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

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

Hidden Units

- **Hidden units** are nodes that are situated between the input nodes and the output nodes.
- Hidden units allow a network to learn non-linear functions.
- Hidden units allow the network to represent combinations of the input features.
- Given too many hidden units, a neural net will simply memorize the input patterns.
- Given too few hidden units, the network may not be able to represent all of the necessary generalizations.

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete, real-valued, or a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

More on Backpropagation

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

Expressive Capabilities of Neural Nets

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:

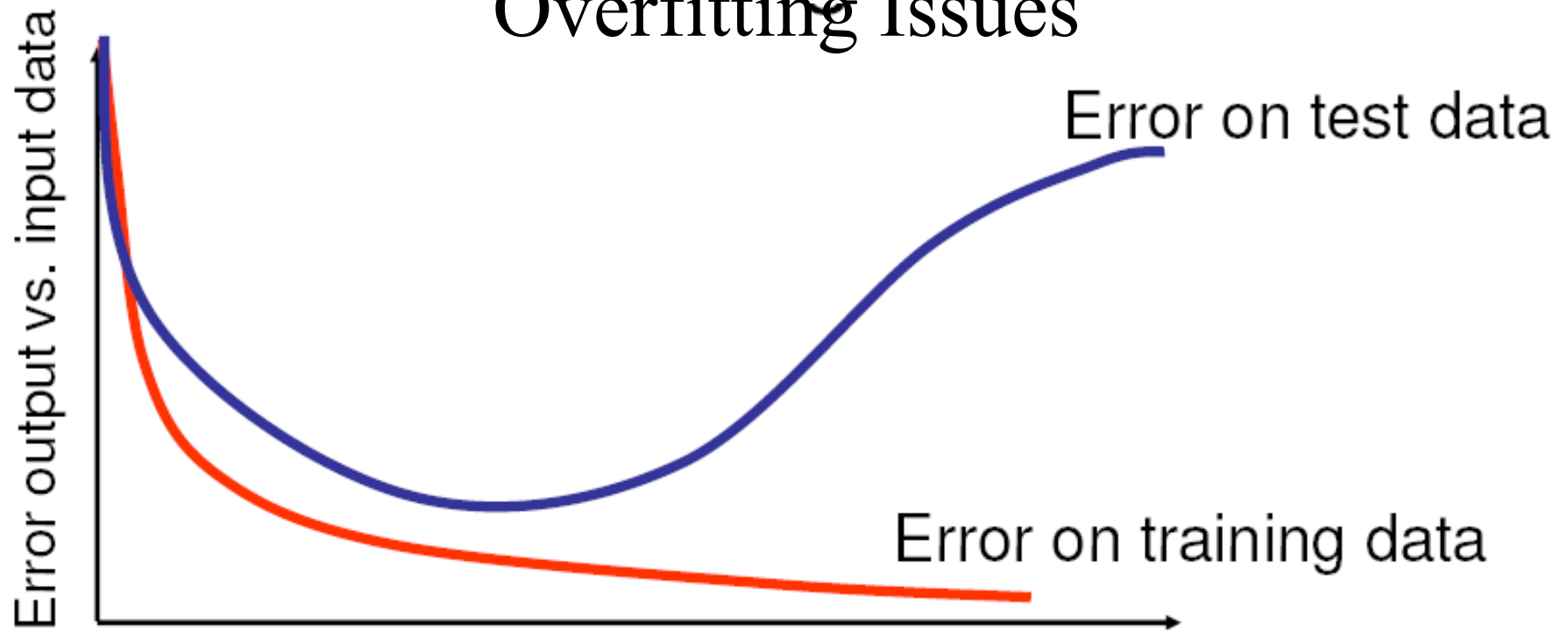
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Multi-Layer Networks: Key Results

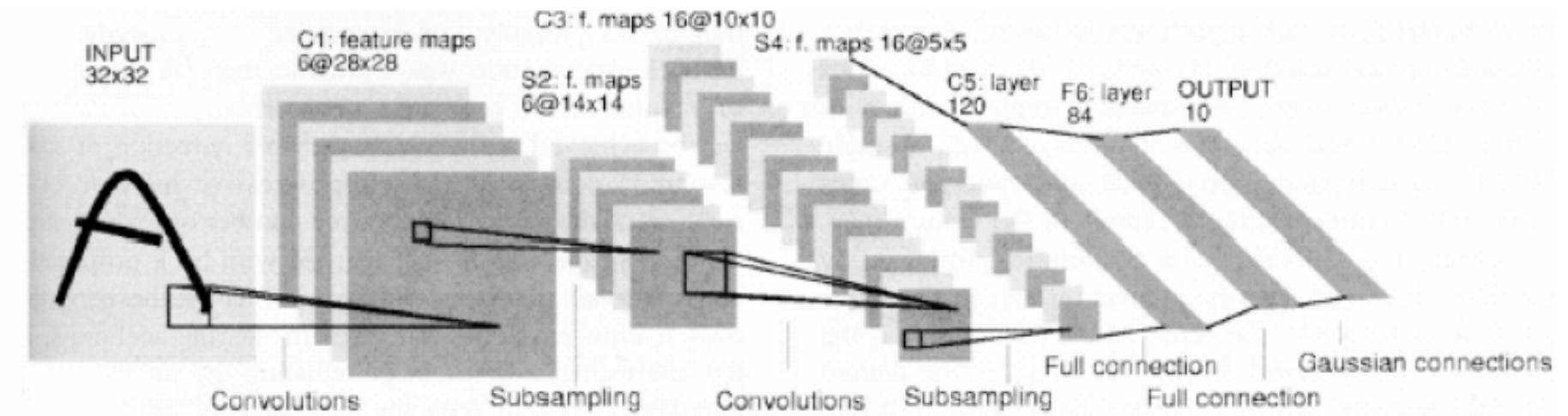
- *Good:* Multi-layer networks can represent **any** arbitrary decision boundary. We are no longer limited to linear boundaries.
- *Bad:* Unfortunately, there is **no** guarantee at all regarding convergence of training procedure.
- More complex networks (more hidden units and/or more layers) can represent more complex boundaries, but beware of overfitting → A complex enough network can always fit the training data!
- In practice: Training works well for reasonable designs of the networks for specific problems.

Overfitting Issues



- NNs have the same overfitting problem as any of the other techniques
- This is addressed essentially in the same way:
 - Train on *training data set*
 - At each step, evaluate performance on *an independent validation test set*
 - Stop when the error on the validation data is minimum

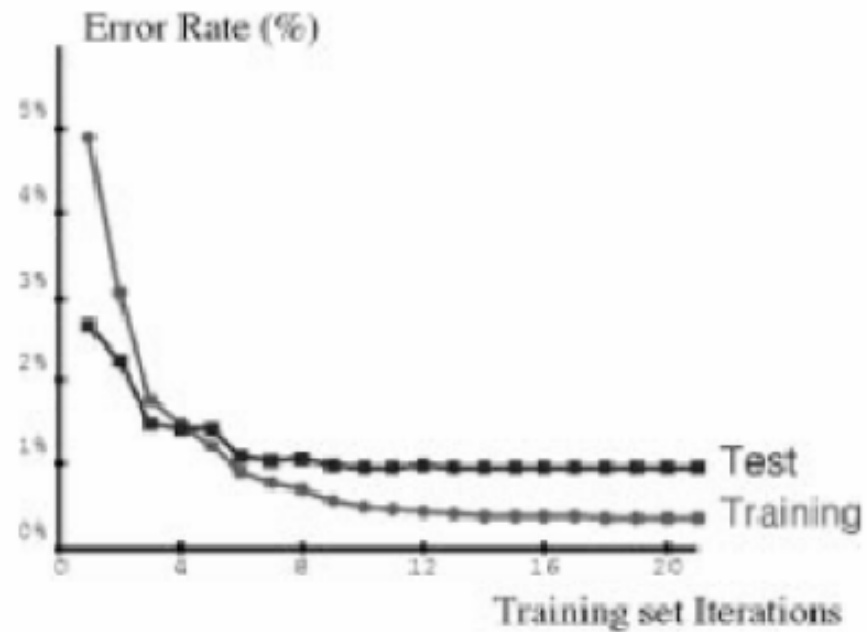
Real Example



- Takes as input image of handwritten digit
- Each pixel is an input unit
- Complex network with many layers
- Output is digit class
- Tested on large (50,000+) database of handwritten samples
- Real-time
- Used commercially

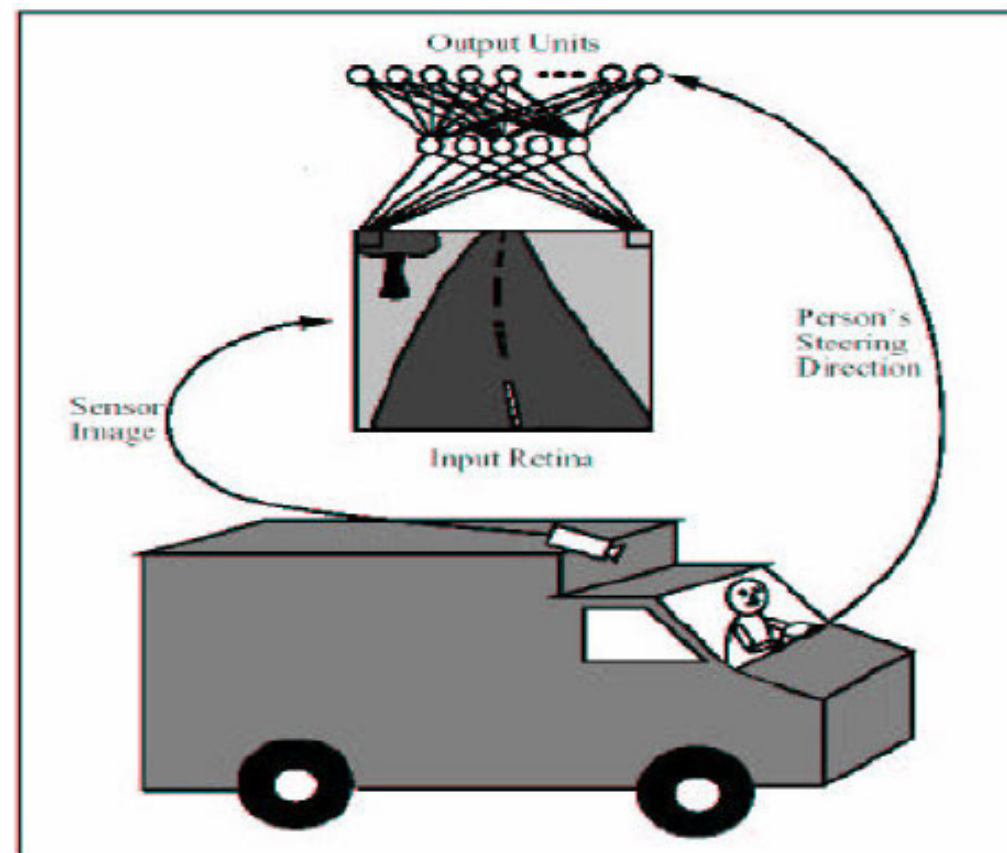
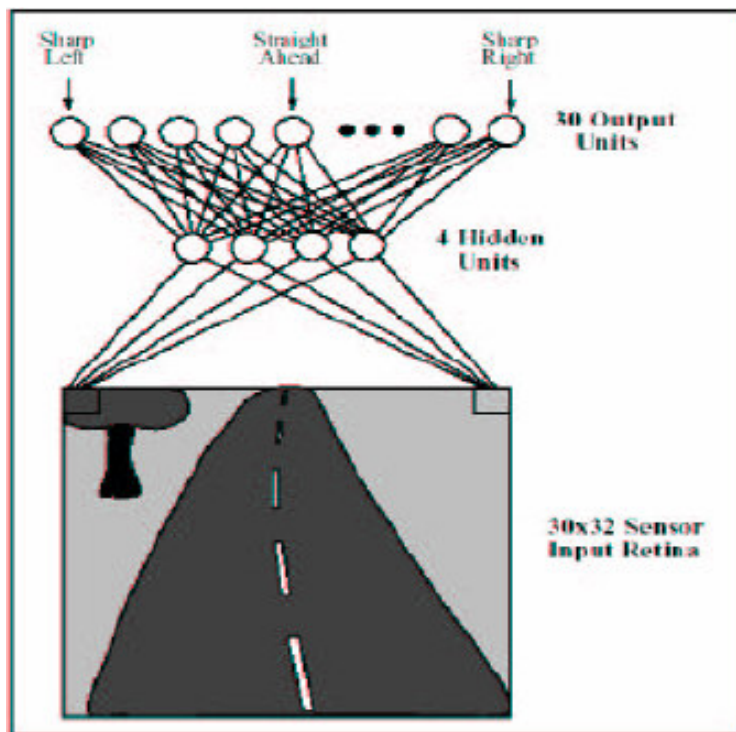
Handwriting Recognition

3 6 8 1 7 9 6 6 9 1
6 7 5 7 8 6 3 4 8 5
2 1 7 9 7 1 2 8 4 5
4 8 1 9 0 1 8 8 9 4
7 6 1 8 6 4 1 5 6 0
7 5 9 2 6 5 8 1 9 7
2 2 2 2 2 3 4 4 8 0
0 2 3 8 0 7 3 8 5 7
0 1 4 6 4 6 0 2 4 3
7 1 2 8 9 6 9 8 6 1



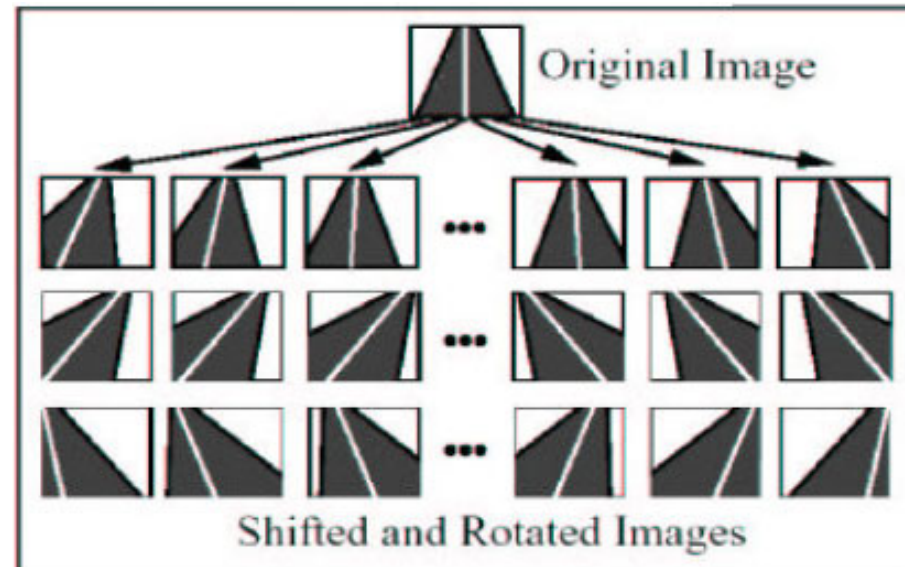
Very low error rate ($\ll 1\%$)

Autonomous Driving

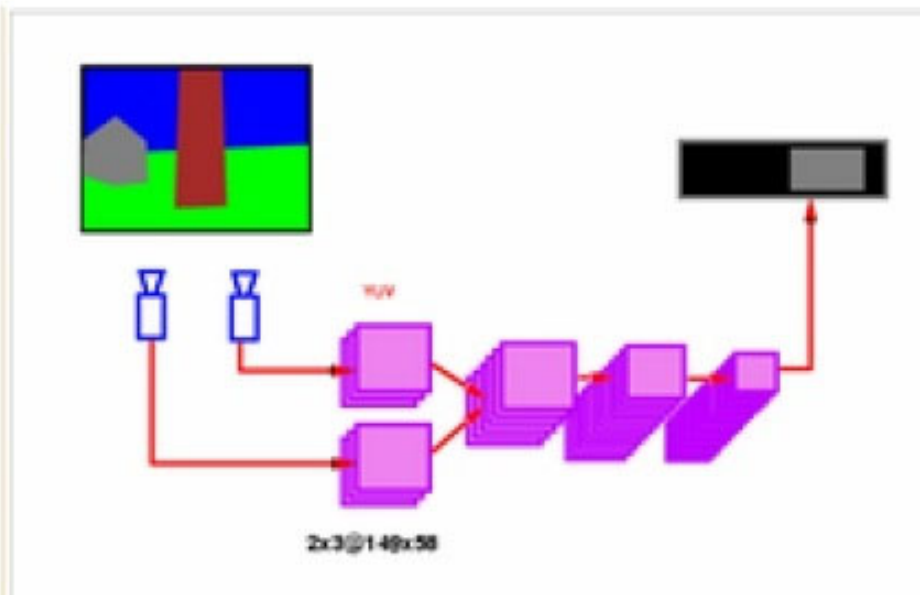


- Learns to drive on roads
- Demonstrated at highway speeds over 100s of miles

Training data:
Images +
corresponding
steering angle



Robot Navigation



Learns to avoid obstacles using cameras

Input: All the pixels from the images from 2 cameras are input units

Output: Steering direction

Network: Many layers (3.15 million connections, and 71,900 parameters!!)

Training: Trained on 95,000 frames from human driving (30,000 for testing)

Execution: Real-time execution on input data (10 frames/sec. approx.)

Summary

- Neural networks used for
 - Approximating y as function of input x (regression)
 - Predicting (discrete) class y as function of input x (classification)
- Key Concepts:
 - Difference between linear and sigmoid outputs
 - Gradient descent for training
 - Backpropagation for general networks
 - Use of validation data for avoiding overfitting
- Good:
 - “simple” framework
 - Direct procedure for training (gradient descent...)
 - Convergence guarantees in the linear case
- Not so good:
 - Many parameters (learning rate, etc.)
 - Need to design the architecture of the network (how many units? How many layers? What activation function at each unit? Etc.)
 - Requires a substantial amount of engineering in designing the network
 - Training can be very slow and can get stuck in local minima