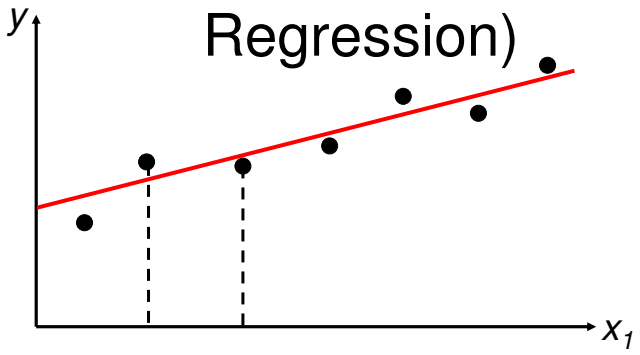
Neural Networks

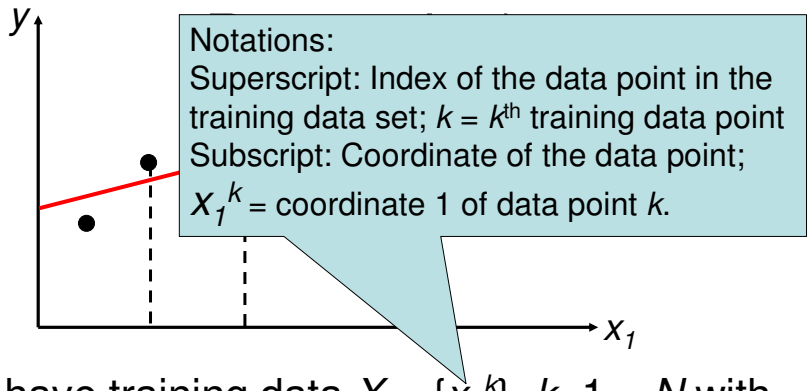
A Simple Problem (Linear Regression)



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

$$Y \approx w_o + w_1 x_1$$

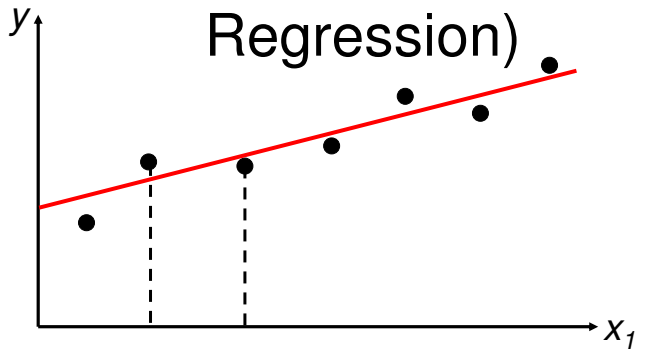
A Simple Problem (Linear



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

$$y^k \approx w_o + w_1 x_1^k$$

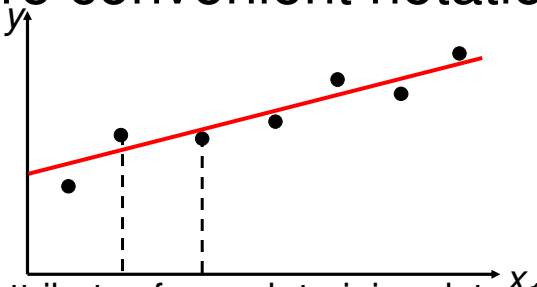
A Simple Problem (Linear Regression)



- It is convenient to define an additional "fake" attribute for the input data: $x_o = 1$
- We want to find the parameters that predict the output Y from the data X in a linear fashion:

$$y^k \approx w_o x_o^k + w_1 x_1^k$$

More convenient notations



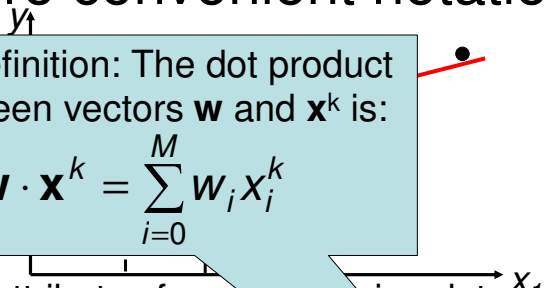
- Vector of attributes for each training data point:

$$\mathbf{x}^k = [x_o^k,..,x_M^k]$$

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

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

More convenient notations



By definition: The dot product between vectors $\mathbf{w}$ and $\mathbf{x}^k$ is:

$$\mathbf{w} \cdot \mathbf{x}^k = \sum_{i=0}^{M} w_i x_i^k$$

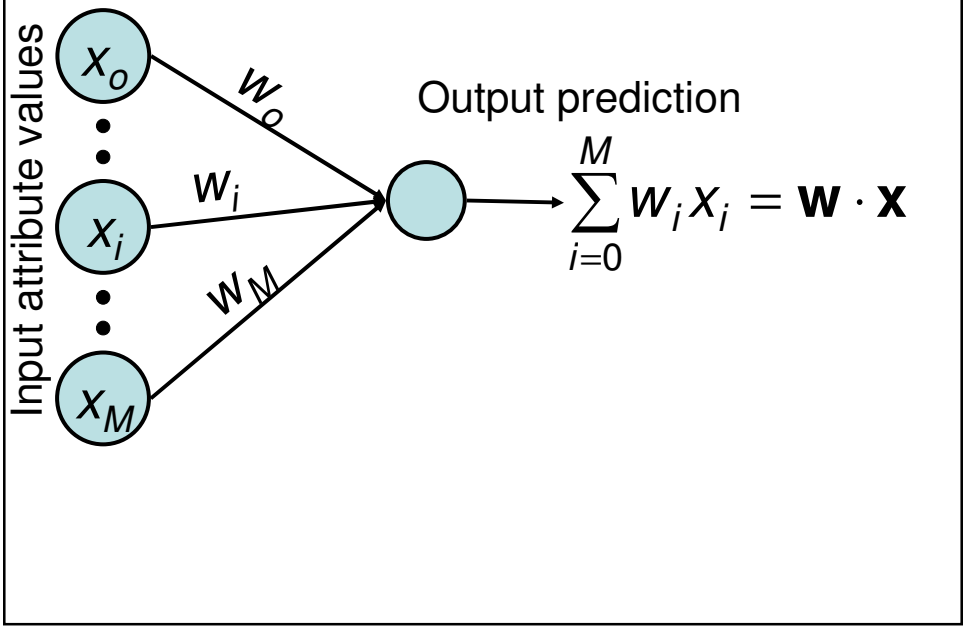
- Vector of attributes for ea ing data point:

$$\mathbf{x}^i = [x_o^i,..,$$

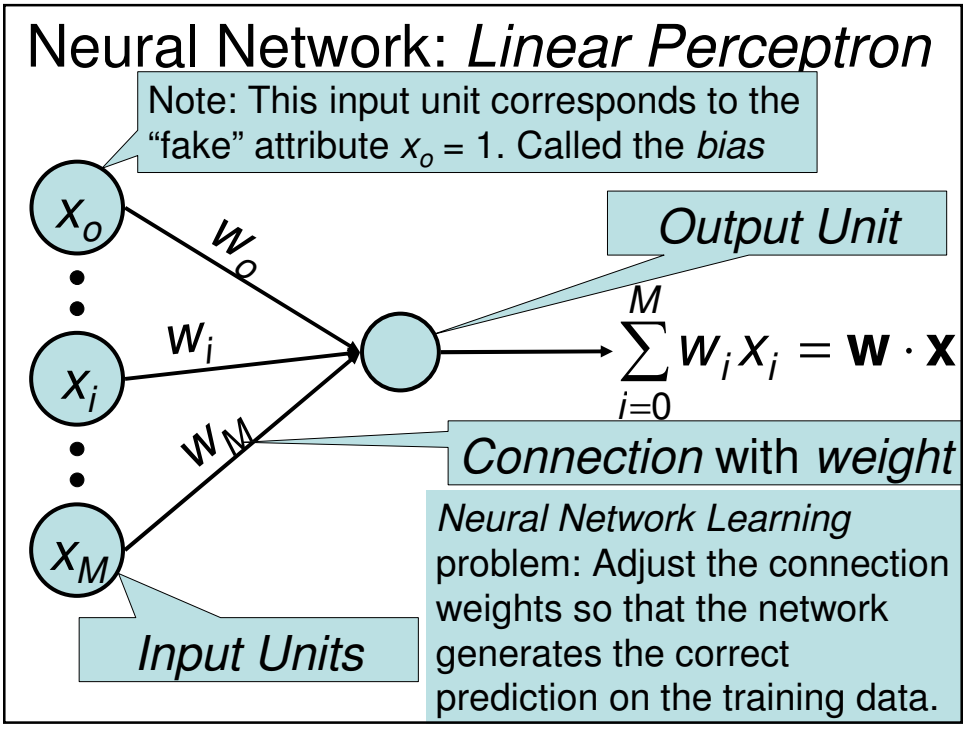
- We seek a vector of parameters: [w_o,..,w_M]
- Such that we have a linear relation be en prediction Y and attributes X:

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

Neural Network: *Linear Perceptron*



Neural Network: *Linear Perceptron*



Linear Regression: Gradient Descent

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

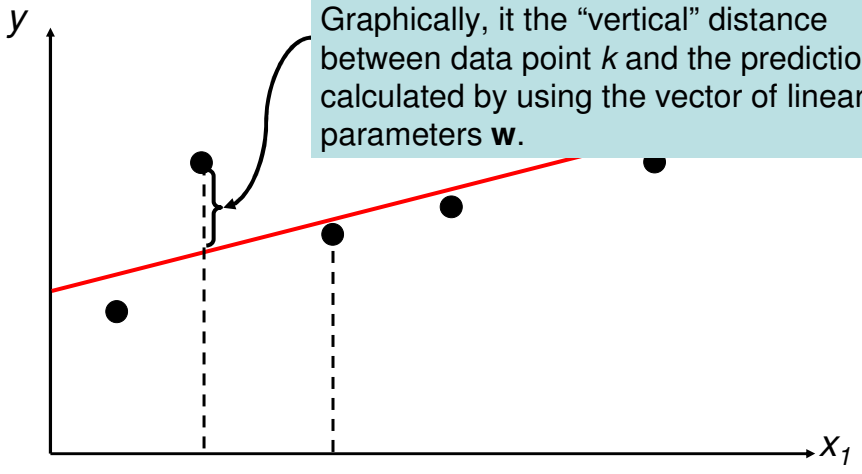
$$E = \sum_{k=1}^{N} \left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k) \right)^2$$

$$= \sum_{k=1}^{N} \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right)^2$$

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

Linear R

δ_k is the error between the input **x** and the prediction y at data point k.
Graphically, it the "vertical" distance between data point k and the prediction calculated by using the vector of linear parameters **w**.



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

Gradient Descent

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

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{N}\left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k)\right)x_i^k$$

$$= -2\sum_{k=1}^{N}\left(y^k - \mathbf{w}\cdot\mathbf{x}^k\right)x_i^k$$

$$= -2\sum_{k=1}^{N}\delta_k x_i^k$$

Gradient Descent

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

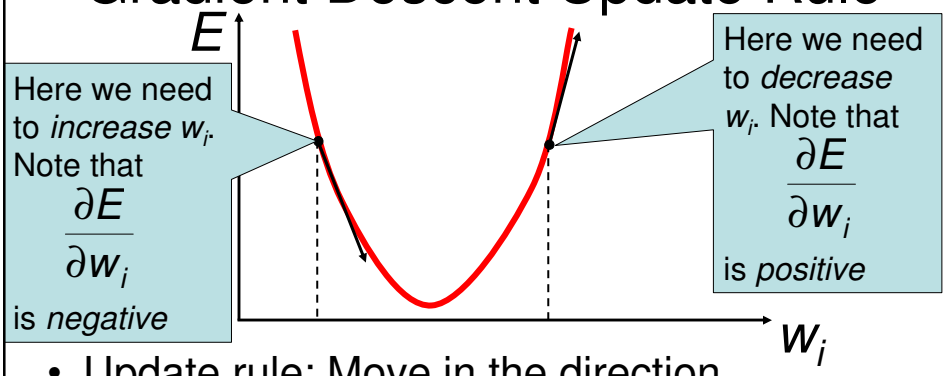
Note that the contribution of training data element number *k* to the overall gradient is $-\delta_k x_i^k$

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{N}\left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k)\right)x_i^k$$

$$= -2\sum_{k=1}^{N}\left(y^k - \mathbf{w}\cdot\mathbf{x}^k\right)x_i^k$$

$$= -2\sum_{k=1}^{N}\delta_k x_i^k$$

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:

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

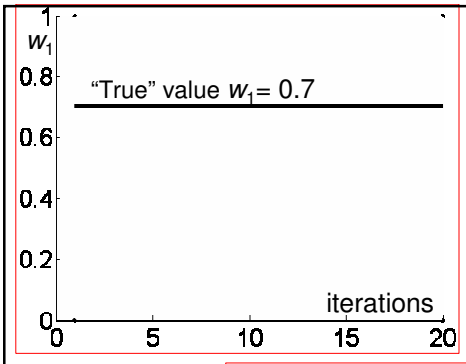
α is the learning rate.
α too small: May converge slowly and may need a lot of training examples
α too large: May change **w** too quickly and spend a long time oscillating around the minimum.

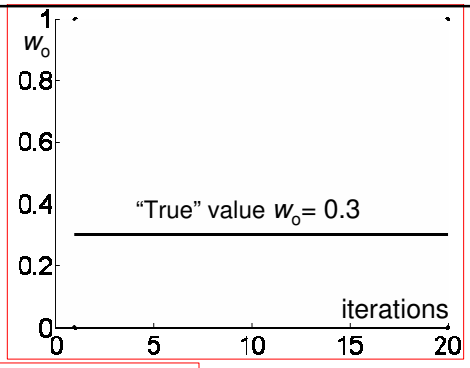
1. Compute error:

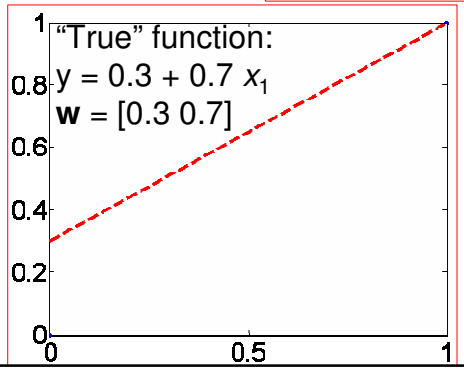
$$\delta_k \leftarrow y^k \cdot \mathbf{x}^k$$

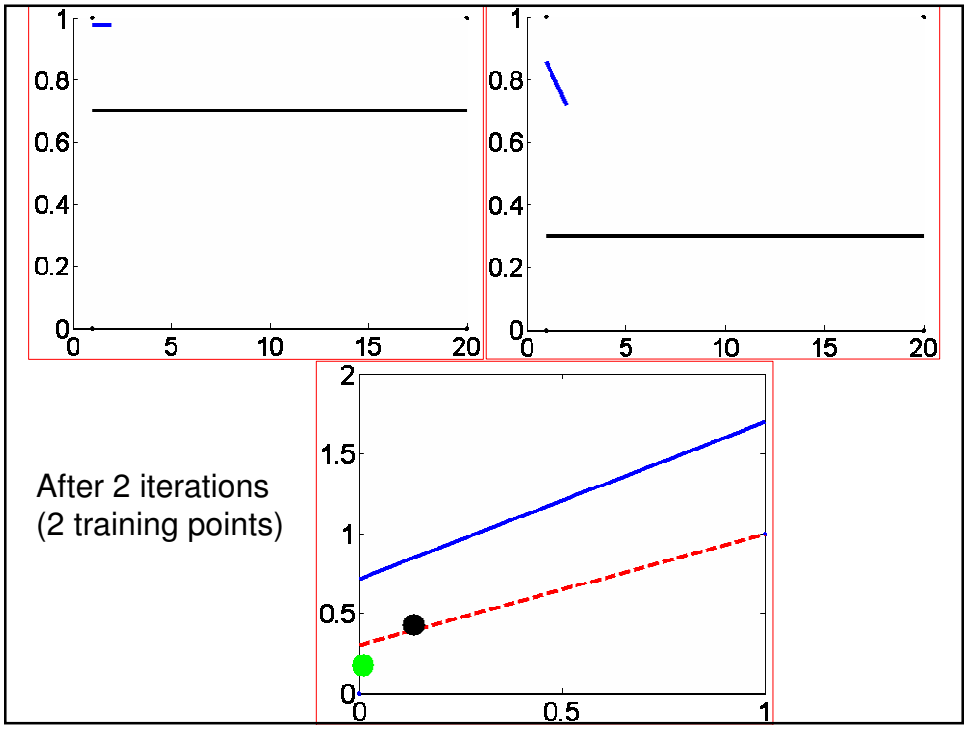
2. Update NN weights:

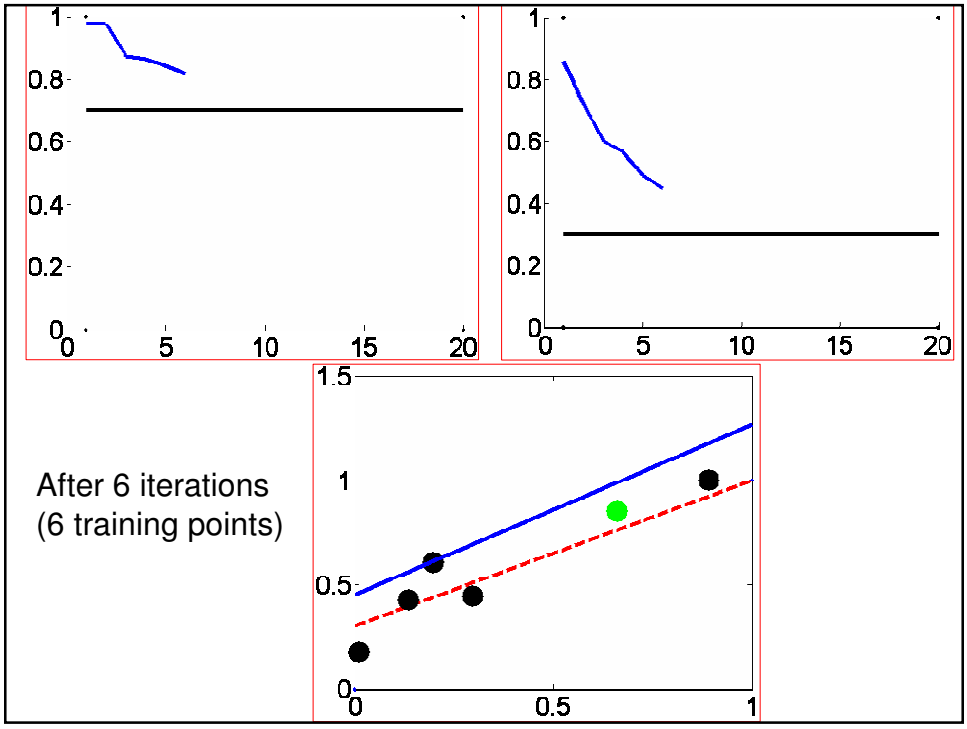
$$w_i \leftarrow w_i + \alpha \delta_k x_i^k$$

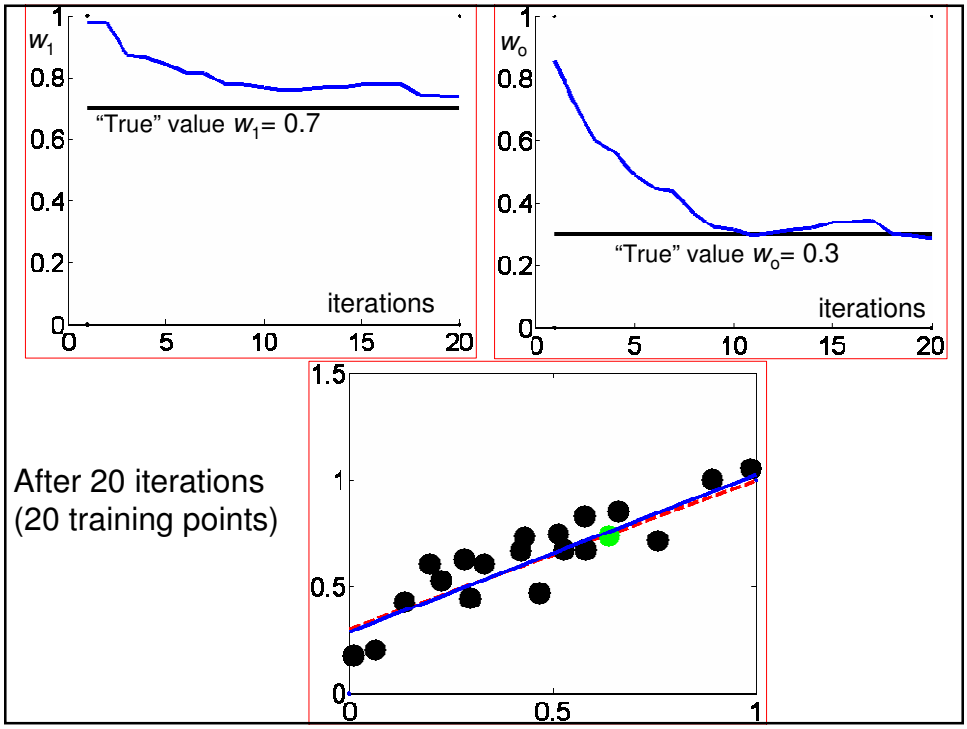







After 2 iterations
(2 training points)


After 6 iterations
(6 training points)

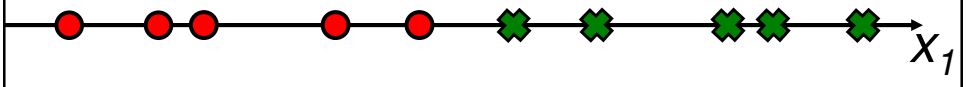


Perceptrons: Remarks

- Update has many names: delta rule, gradient rule, LMS rule…..
- Update is ***guaranteed*** to converge to the best linear fit (global minimum of *E*)
- Of course, there are more direct ways of solving the linear regression problem by using linear algebra techniques. It boils down to a simple matrix inversion (not shown here).
- In fact, the perceptron training algorithm can be much, much slower than the direct solution
- So why do we bother with this? The answer in the next few of slides…be patient

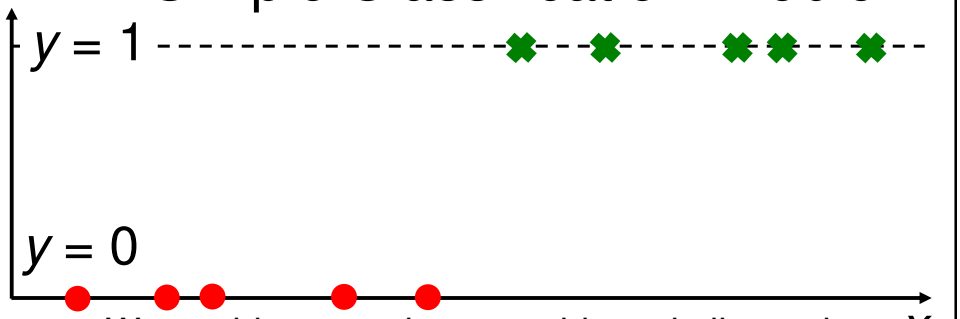
A Simple Classification Problem

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 and nearest-neighbors).

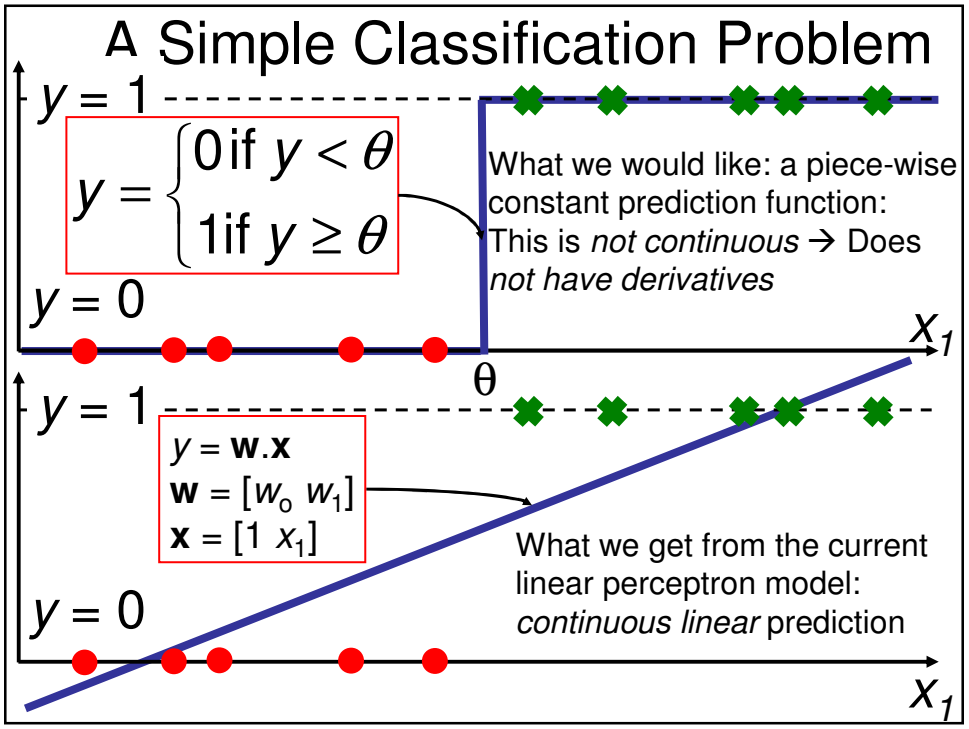
A Simple Classification Problem

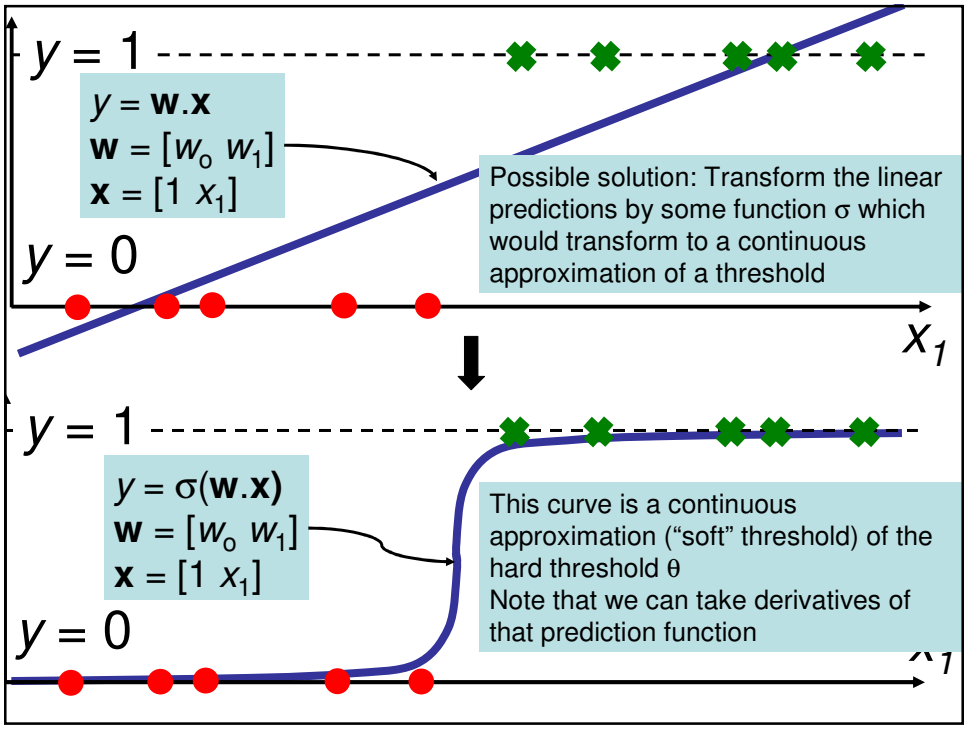


- We could convert it to a problem similar to the previous one by defining an output value y

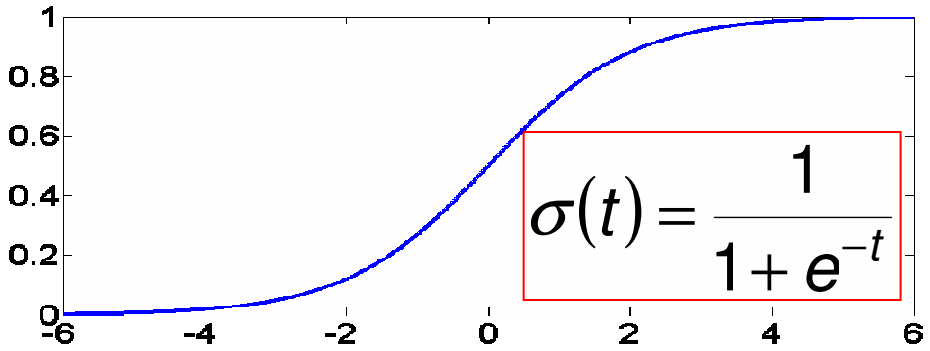
$$y = \begin{cases} 0 \text{ if in red class} \\ 1 \text{ if in green class} \end{cases}$$

- 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
y = 1
$y = \begin{cases} 0 \text{ if } y < \theta \\ 1 \text{ if } y \geq \theta \end{cases}$
What we would like: a piece-wise constant prediction function:
This is *not continuous* → Does *not have derivatives*
y = 0
x_1
θ
y = 1
y = **w**.**x**
w = [w_0 w_1]
x = [1 x_1]
What we get from the current linear perceptron model:
continuous linear prediction
y = 0
x_1


y = 1
y = **w**.**x**
w = [w_0 w_1]
x = [1 x_1]
Possible solution: Transform the linear predictions by some function σ which would transform to a continuous approximation of a threshold
y = 0
x_1
y = 1
y = σ(**w**.**x**)
w = [w_0 w_1]
x = [1 x_1]
This curve is a continuous approximation ("soft" threshold) of the hard threshold θ
Note that we can take derivatives of that prediction function
y = 0

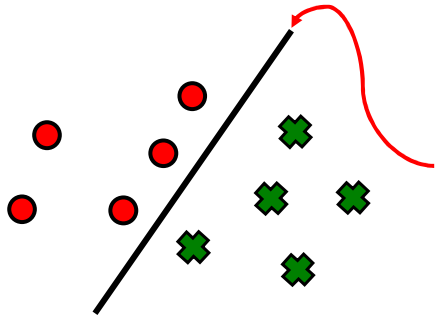
The Sigmoid Function



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

- Note: It is ***not*** important to remember the exact expression of σ (in fact, alternate definitions are used for σ). What is important to remember is that:
  - It is smooth and has a derivative σ' (exact expression is unimportant)
  - It approximates a hard threshold function at $x = 0$

Generalization to *M* Attributes



A linear separation is parameterized like a line:

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

- 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

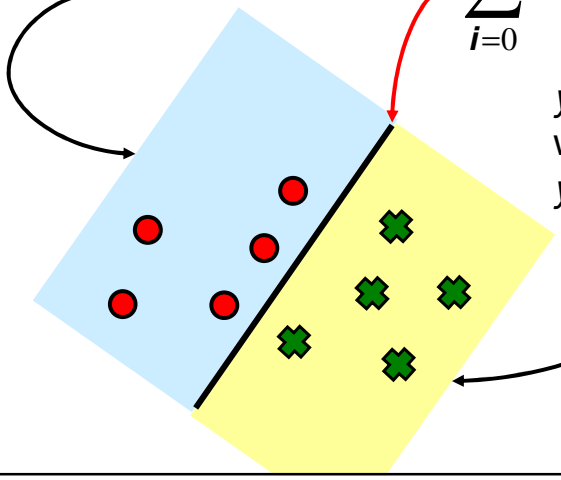
Generalization to *M* Attributes

y = 0 in this region, we can approximate *y* by σ(**w**.**x**) ≈ 0

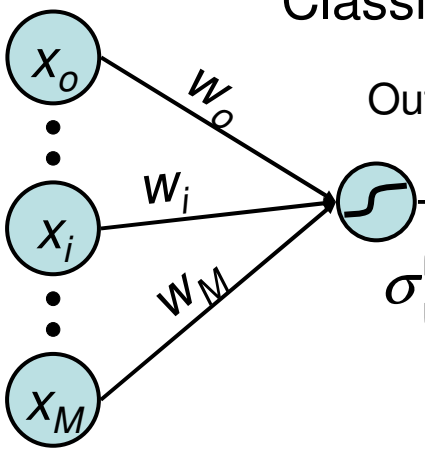
A linear separation is parameterized like a line:

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

y = 1 in this region, we can approximate *y* by σ(**w**.**x**) ≈ 1



Single Layer Network for Classification

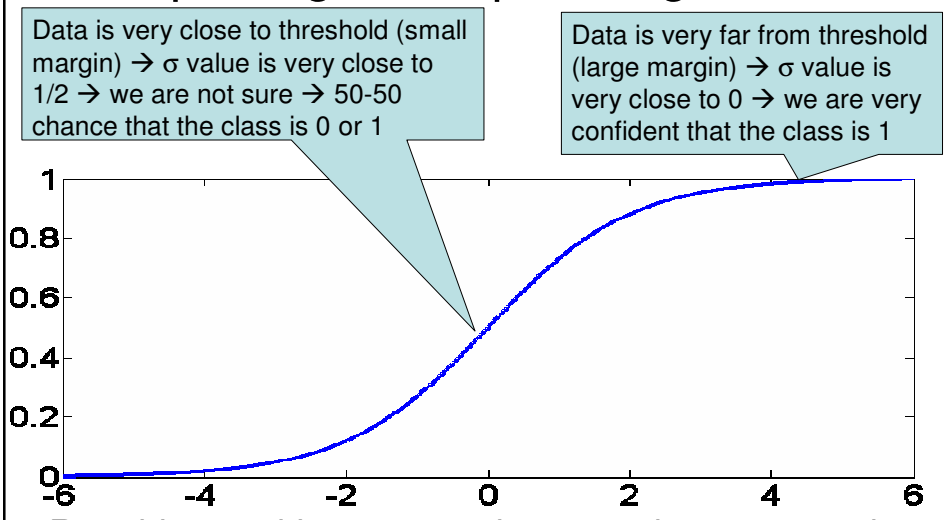


Output prediction

$$\sigma\left(\sum_{i=0}^{M} w_i x_i\right) = \sigma(\mathbf{w} \cdot \mathbf{x})$$

- Term: Single-layer Perceptron

Interpreting the Squashing Function



- Roughly speaking, we can interpret the output as how confident we are in the classification: Prob(y=1|x)

Training

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

1. Compute error:

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

2. Update NN weights:

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

Note: It is exactly the same as before, except for the additional complication of passing the linear output through σ

- Given input tra… ata x^k with corresponding va…

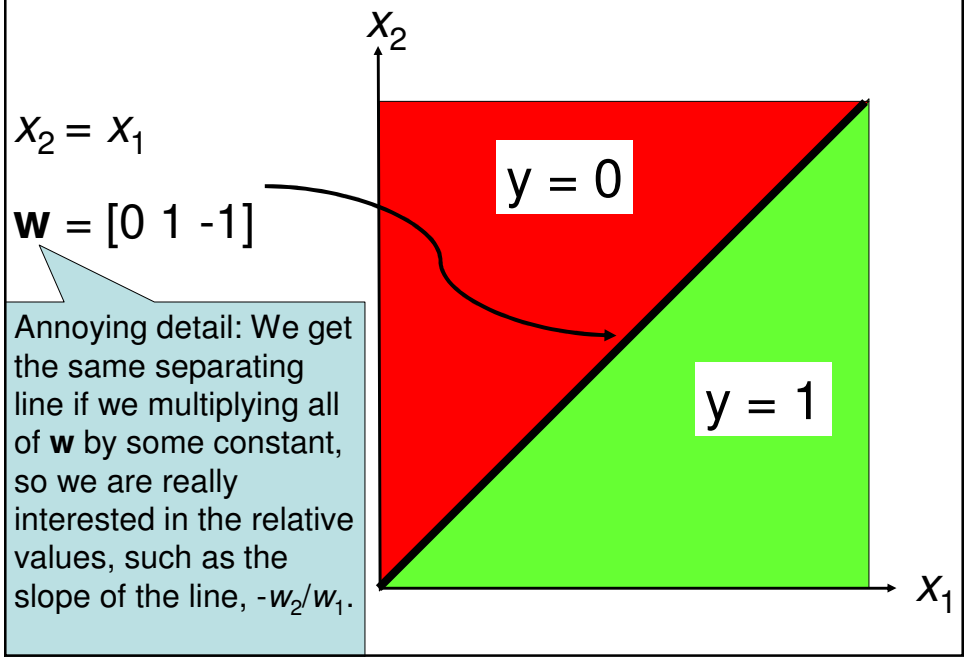
1. Compute error:

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

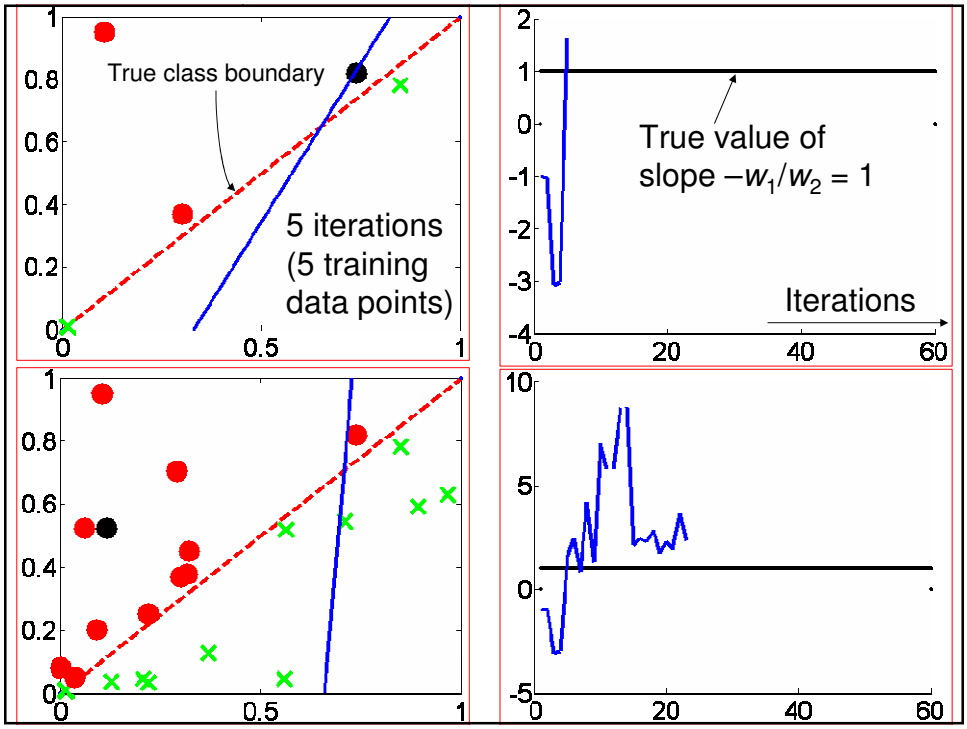
This formula derived by direct application of the chain rule from calculus

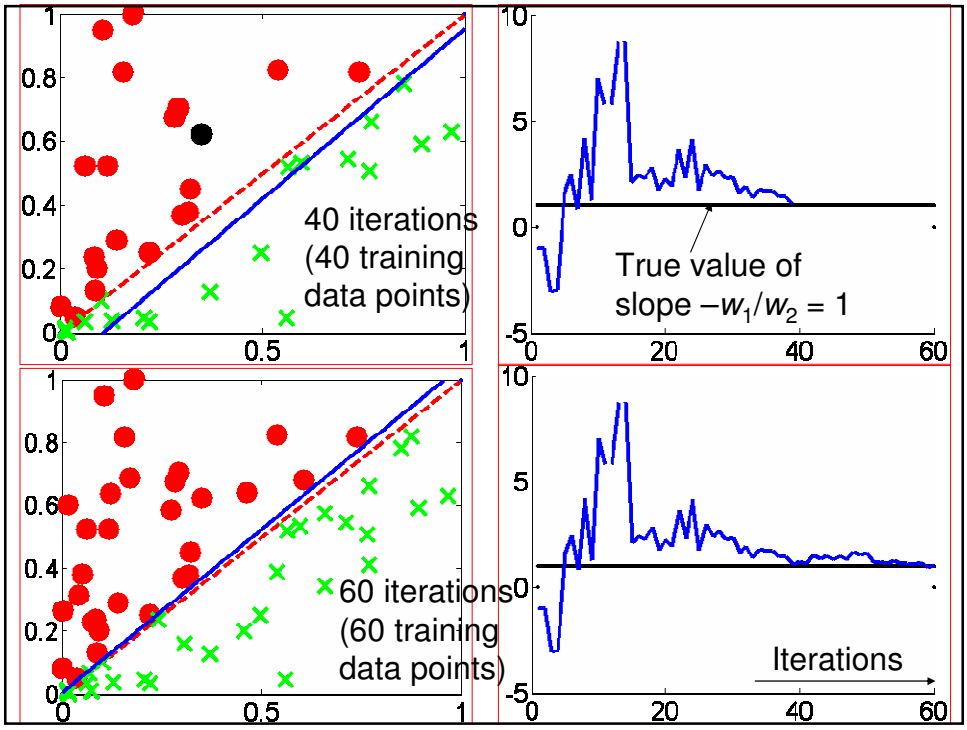
$$w_i \leftarrow w_i + \alpha \delta_k x_i^k \sigma'\left(\mathbf{w} \cdot \mathbf{x}^k\right)$$

Example



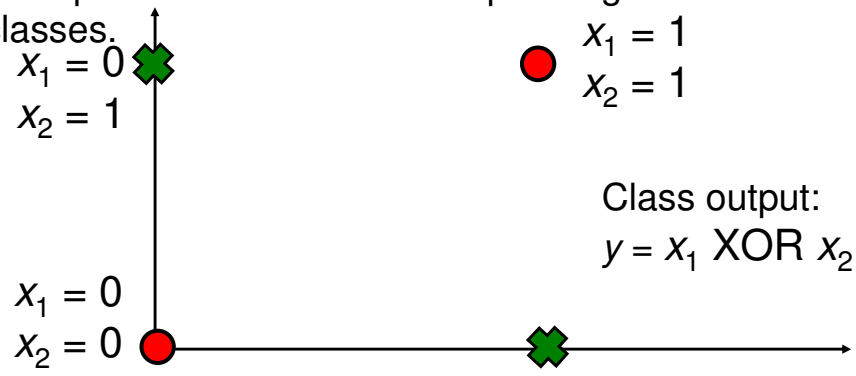
Annoying detail: We get the same separating line if we multiplying all of **w** by some constant, so we are really interested in the relative values, such as the slope of the line, $-w_2/w_1$.


True class boundary
5 iterations
(5 training
data points)
True value of
slope $-w_1/w_2 = 1$
Iterations

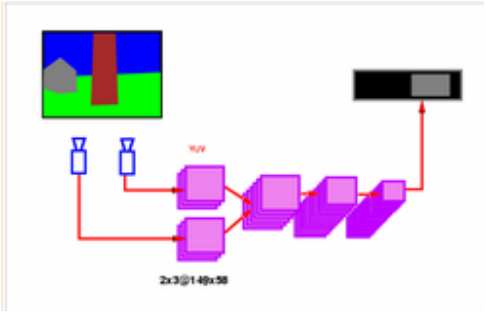

40 iterations
(40 training
data points)
True value of
slope $-w_1/w_2 = 1$
60 iterations
(60 training
data points)
Iterations

Single Layer: Remarks

- Good news: Can represent any problem in which the decision boundary is *linear*.
- Bad news: *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.

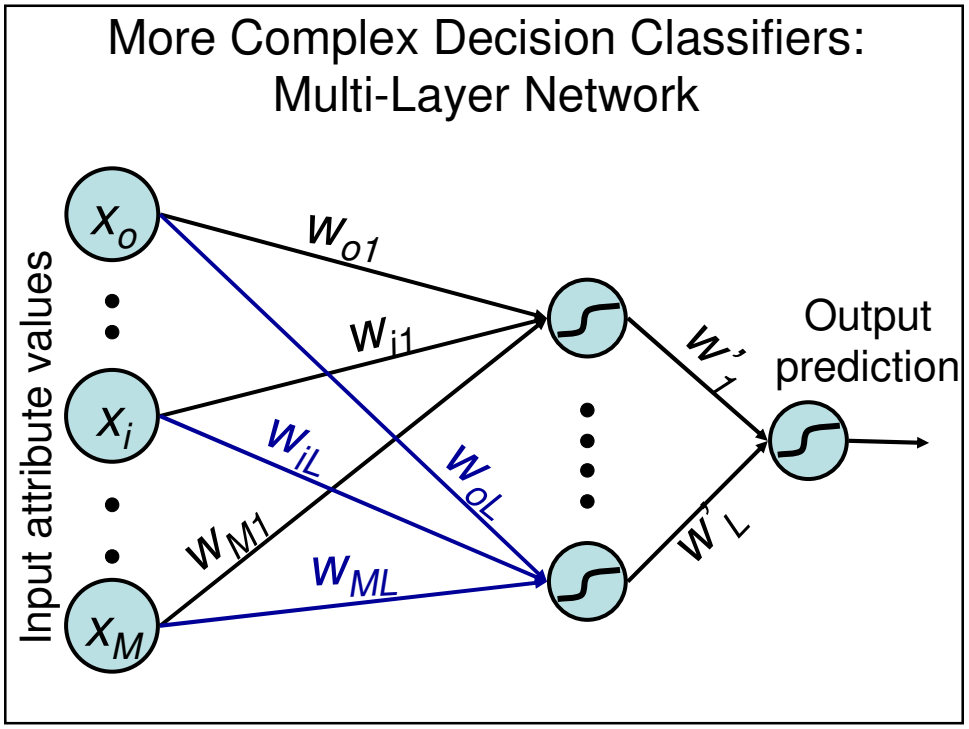


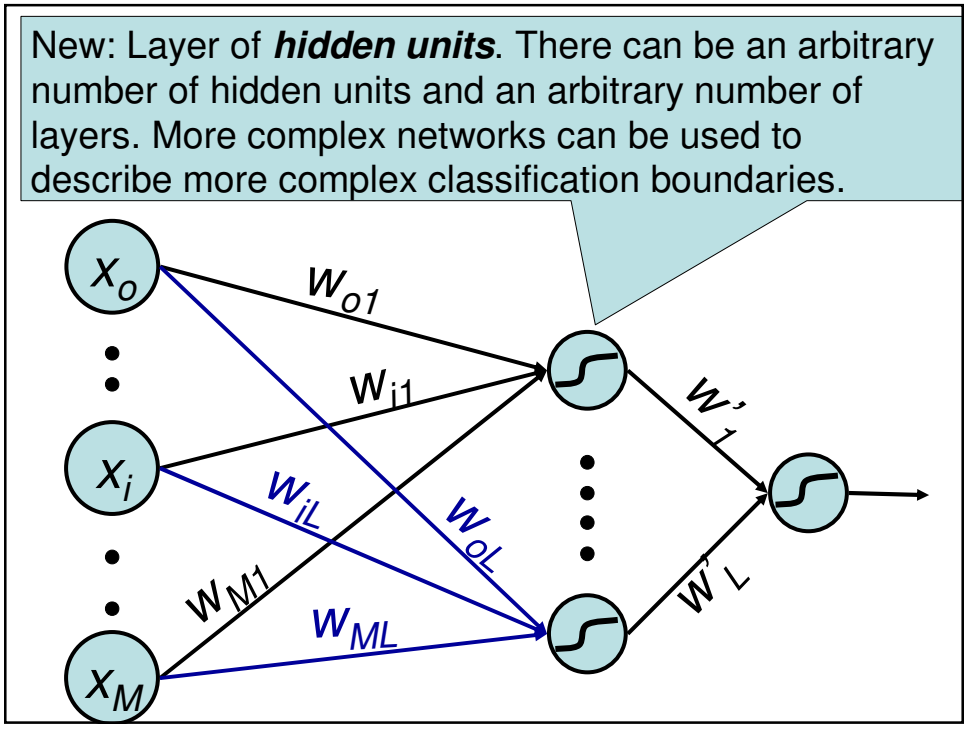
Why we need more complex models than just "linear" (and why we need to suffer through a lot more slides)



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.)

From 2004: http://www.cs.nyu.edu/~yann/research/dave/index.html


More Complex Decision Classifiers:
Multi-Layer Network
Input attribute values
x_o
x_i
x_M
w_{o1}
w_{i1}
w_{iL}
w_{oL}
w_{M1}
w_{ML}
w'_1
w'_L
Output prediction


New: Layer of ***hidden units***. There can be an arbitrary number of hidden units and an arbitrary number of layers. More complex networks can be used to describe more complex classification boundaries.
x_o
x_i
x_M
w_{o1}
w_{i1}
w_{iL}
w_{oL}
w_{M1}
w_{ML}
w'_1
w'_L

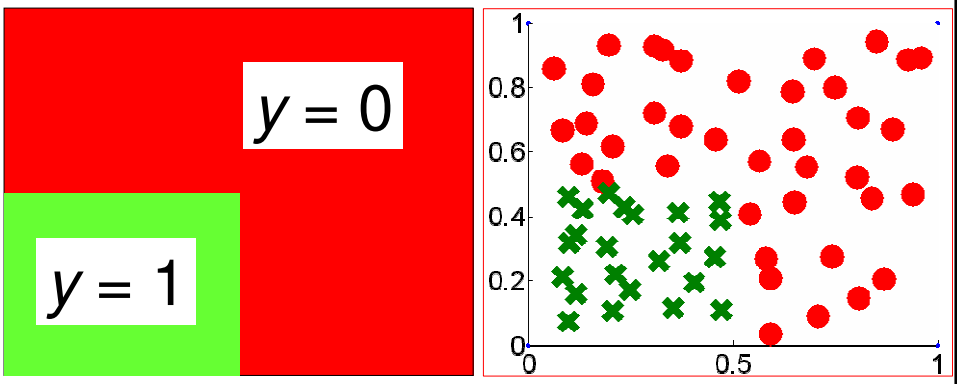
Training: Backpropagation

- The analytical expression of the network's output becomes totally unwieldy (something like:

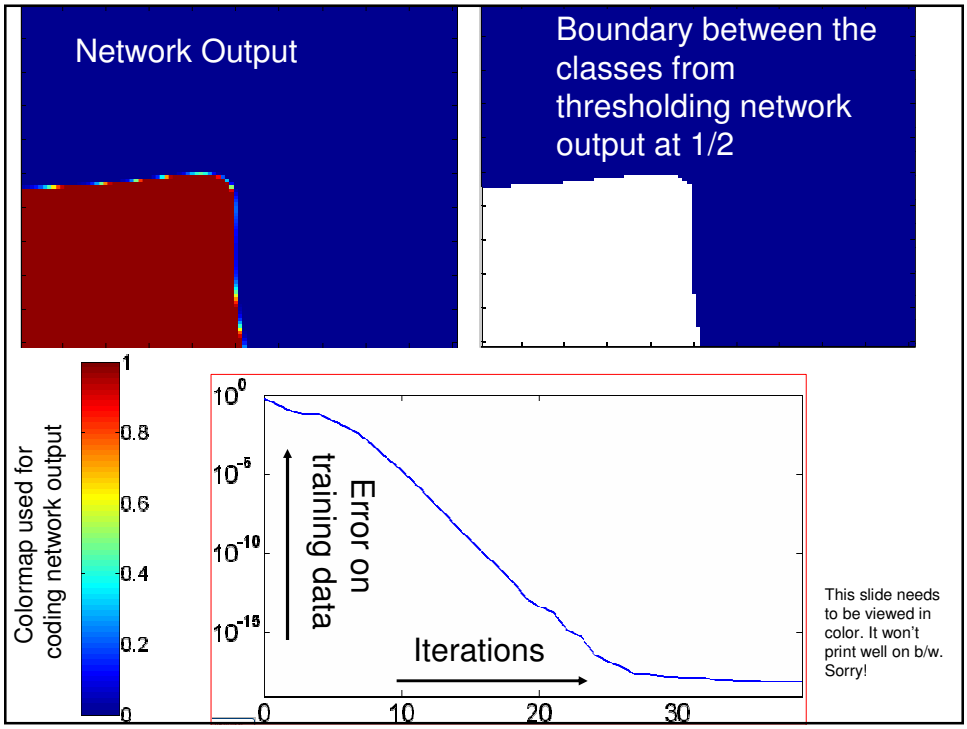
$$\text{output} = \sigma\left(\sum_{m=1}^{L} w'_m \sigma\left(\sum_{j=0}^{M} w_{jm} x_j\right)\right)$$

- Things get worse as we can use an arbitrary number of layers with an arbitrary number of hidden units
- The point is that we can still take the derivatives of this monster expression with respect to all of the weights and adjust them to do gradient descent based on the training data
- Fortunately, there is a mechanical way of propagating the errors (the δ's) through the network so that the weights are correctly updated. So we never have to deal directly with these ugly expressions.
- This is called *backpropagation*. Backpropagation implements the gradient descent shown earlier in the general case of multiple layers

(Contrived) Example



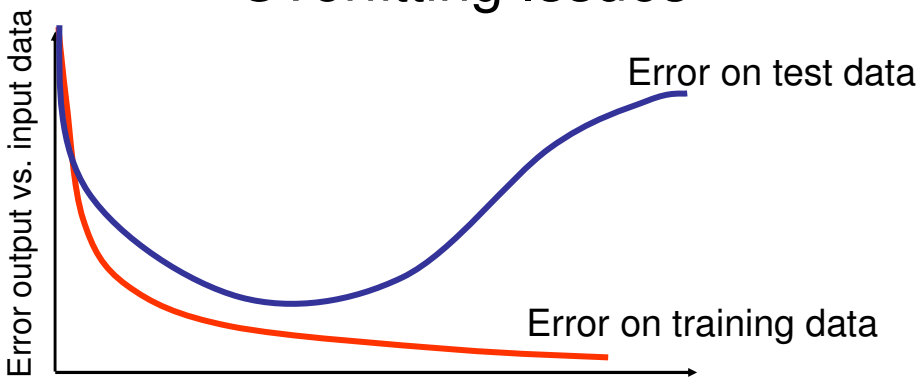
Classes cannot be separated by a linear boundary
Let's try: 2-layer network, 4 hidden units



Multi-Layer Networks: Key Results

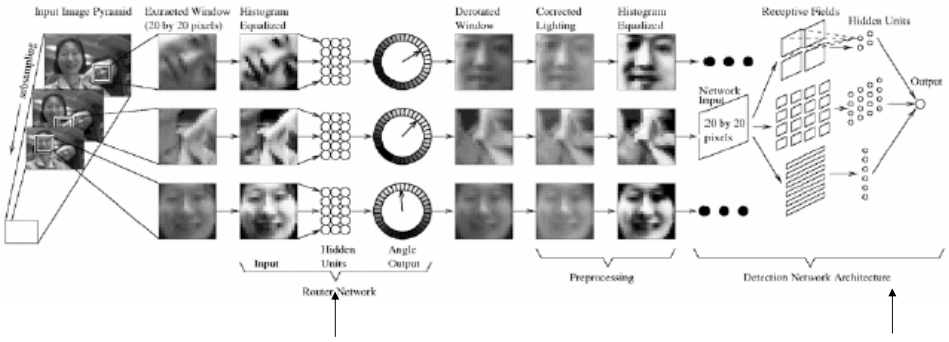
- *Good:* Multi-layer networks can represent ***any*** arbitrary decision boundary. We are no longer limited to linear boundaries.
- *Bad:* Unfortunately, there is ***no*** guarantee at all regarding convergence of training procedure.
- More complex networks (more hidden units and/or more layers) can 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 test data is minimum

Real Example

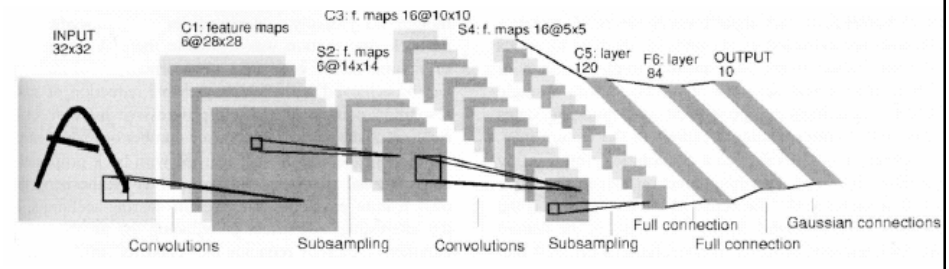


f(x) = orientation of template sampled at 10° intervals

f(x) = face detection posterior distribution

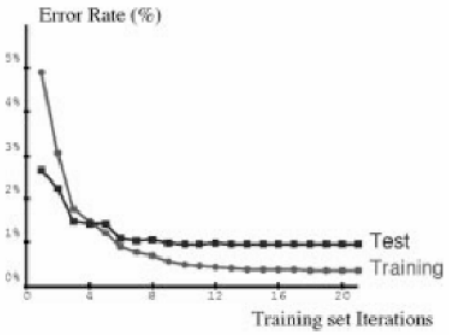
H. Rowley, S. Baluja, and T. Kanade. Rotation Invariant Neural Network-Based Face Detection. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, June, 1998.

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



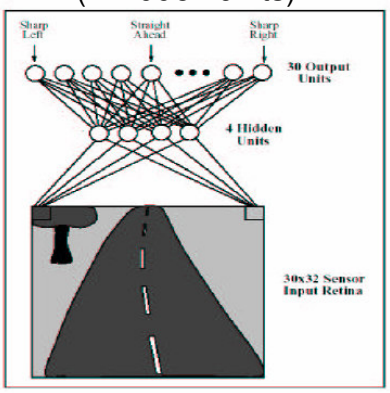
Very low error rate (<< 1%

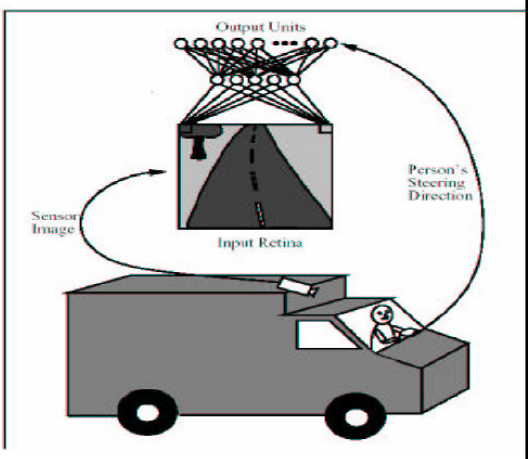
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998.

http://yann.lecun.com/exdb/lenet/

Real Example

network with 1 layer
(4 hidden units)



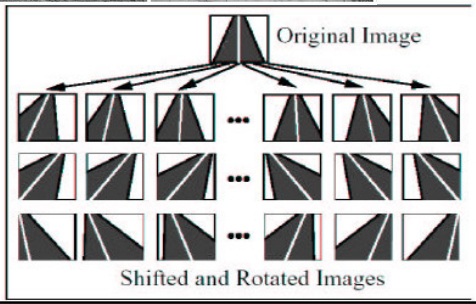


- Learns to drive on roads
- Demonstrated at highway speeds over 100s of miles

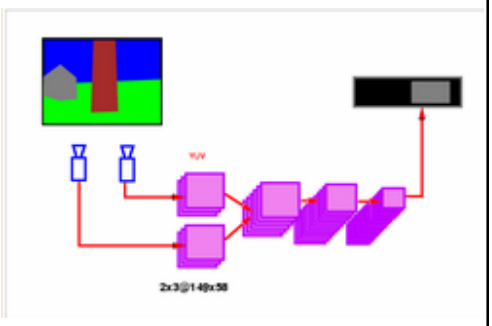
D. Pomerleau. *Neural network perception for mobile robot guidance.* Kluwer Academic Publishing, 1993.

Training data:
Images + corresponding steering angle

Important:
Conditioning of training data to generate new examples → avoids overfitting



Real Example



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.)

From 2004: http://www.cs.nyu.edu/~yann/research/dave/index.html

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 transfer 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