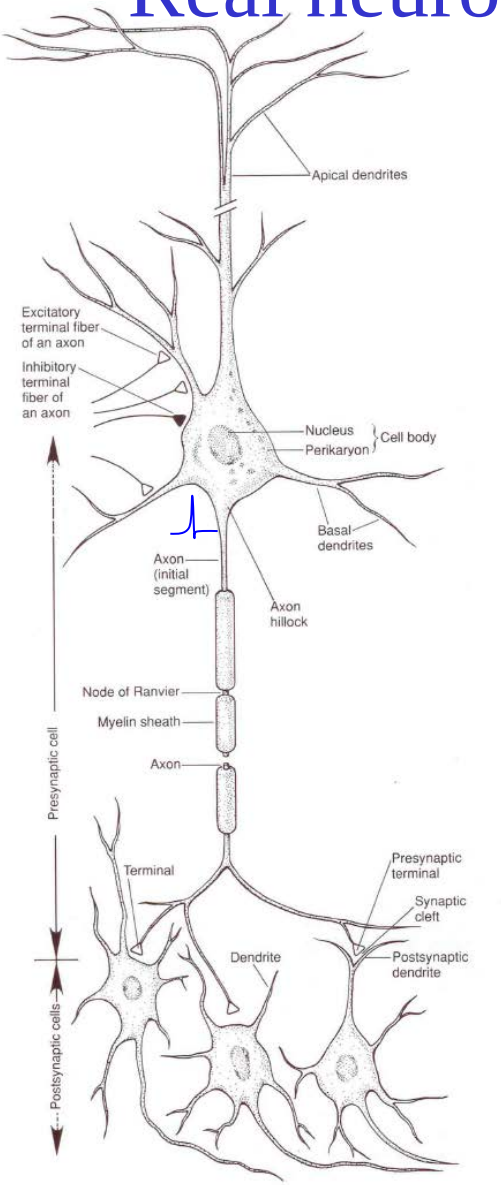
CSE 5526: Introduction to Neural Networks

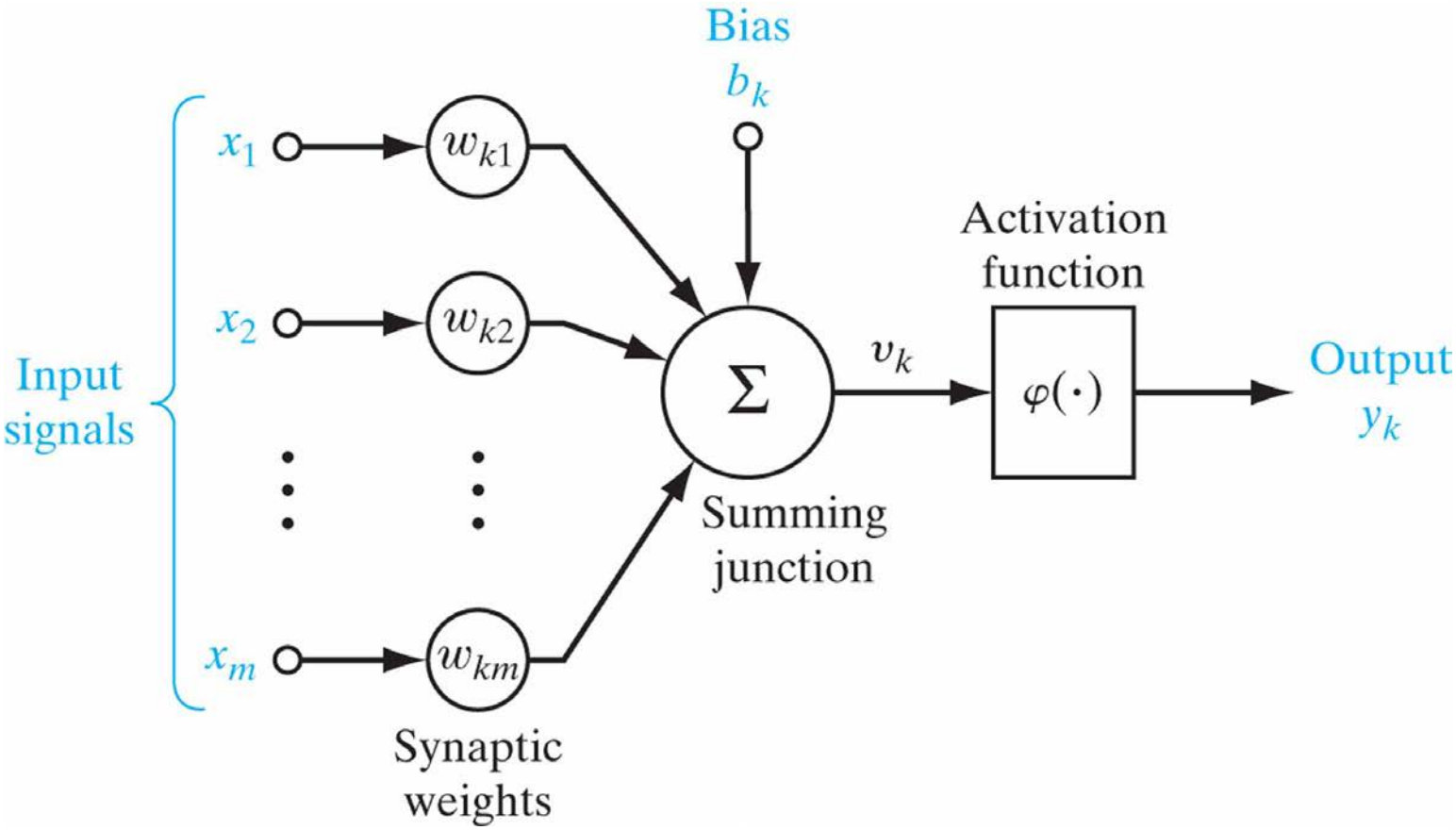
Review to date

Real neurons have three main parts



- Cell body (~50μm)
  - Initiates action potential
- Axon (0.2-20μm)
  - Transmits signal to up to 1000 other neurons
  - Insulated by myelin sheath
  - Up to 1m long
- Dendrites: receive signals
  - Synapse: junction to another neuron's axon

This model approximates the neural firing rate



McCulloch-Pitts neuron model

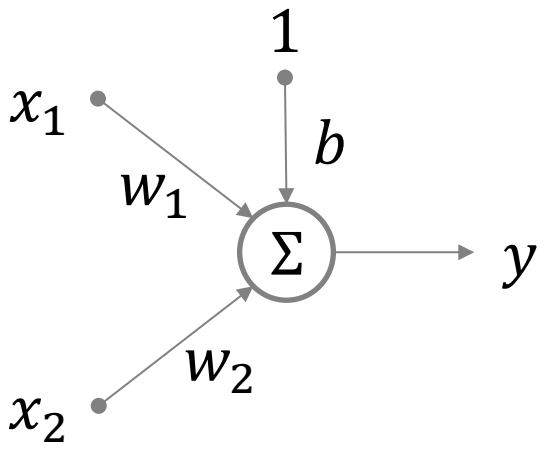
$x_i \in \{-1, 1\}$ Bipolar input

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ -1 & \text{if } v < 0 \end{cases}$$

A form of signum (sign) function

$$y = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

M-P neurons can implement any logic function



$$y = w_1 x_1 + w_2 x_2 + b$$

x_1	x_2	x_1 AND x_2	x_1 OR x_2	NOT x_1
-1	-1	-1	-1	1
-1	1	-1	1	1
1	-1	-1	1	-1
1	-1	1	1	-1

	x_1 AND x_2	x_1 OR x_2	NOT x_1
w_1	1	1	-1
w_2	1	1	0
b	-0.5	0.5	0

M-P neurons have a linear decision boundary

- Can we visualize the decision the perceptron would make in classifying every potential point?
- Yes, it is called the discriminant function

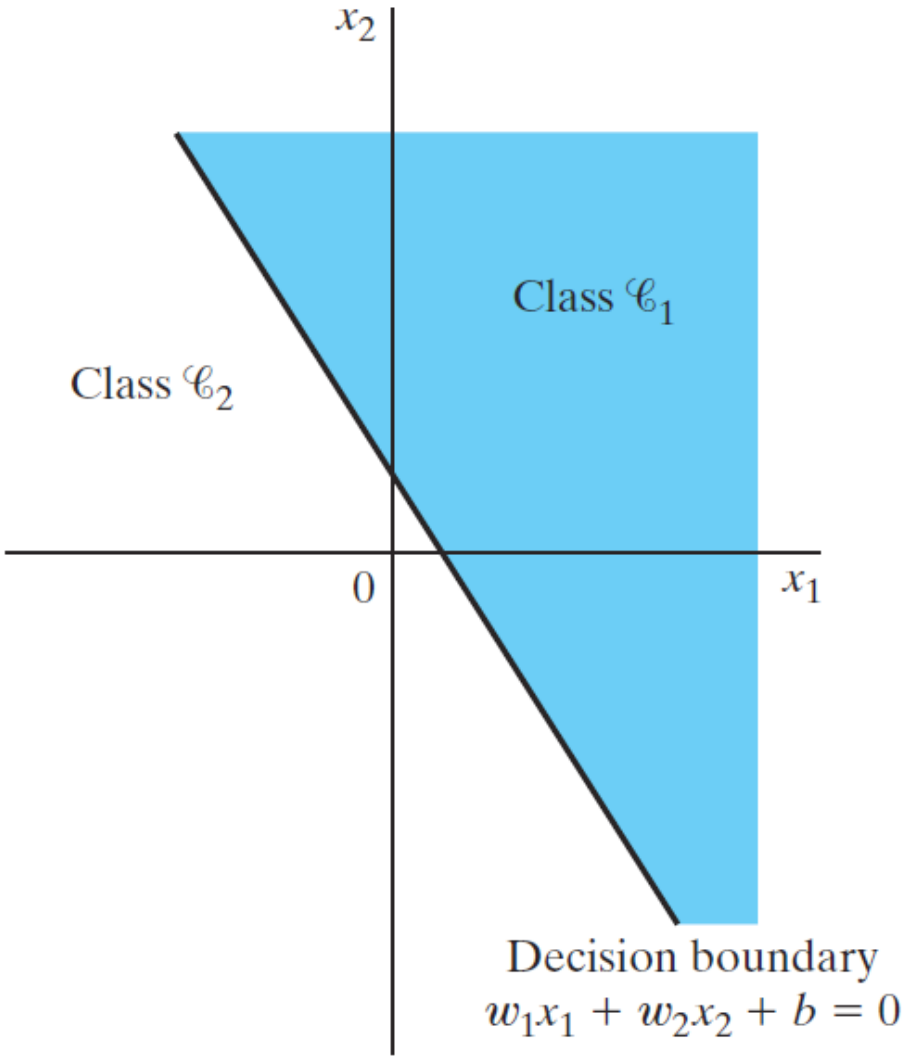
$$g(x) = x^T w = \sum_{i=0}^{m} w_i x_i$$

- What is the boundary between the two classes like?

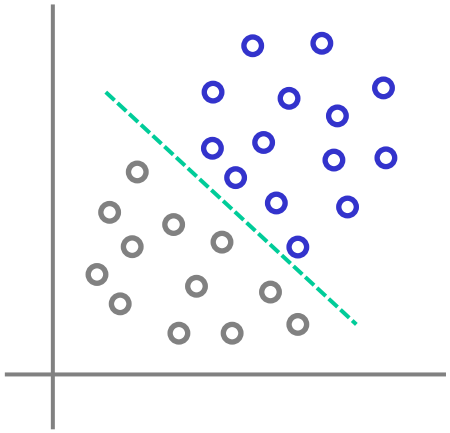
$$g(x) = x^T w = 0$$

- This is a linear function of x

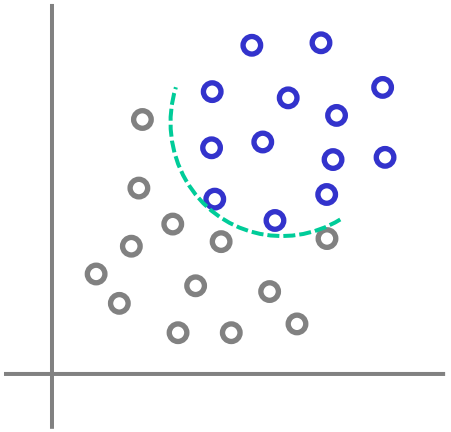
M-P neurons have a linear decision boundary



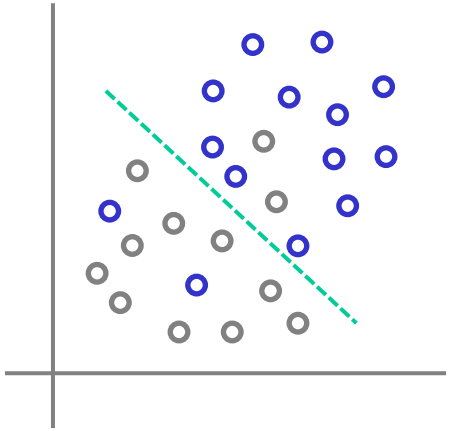
Linear decision functions can't solve all classification problems



Linearly separable



Non-linearly separable



Not separable

Distinction depends on "scale" of classifier

Perceptron algorithm learns weights from data

- Learn parameters *w* from examples (x_p, d_p)
- In an online fashion, i.e., one point at a time
- Adjust weights as necessary, i.e. when incorrect
- Adjust weights to be more like d=1 points and more like negative d=-1 points

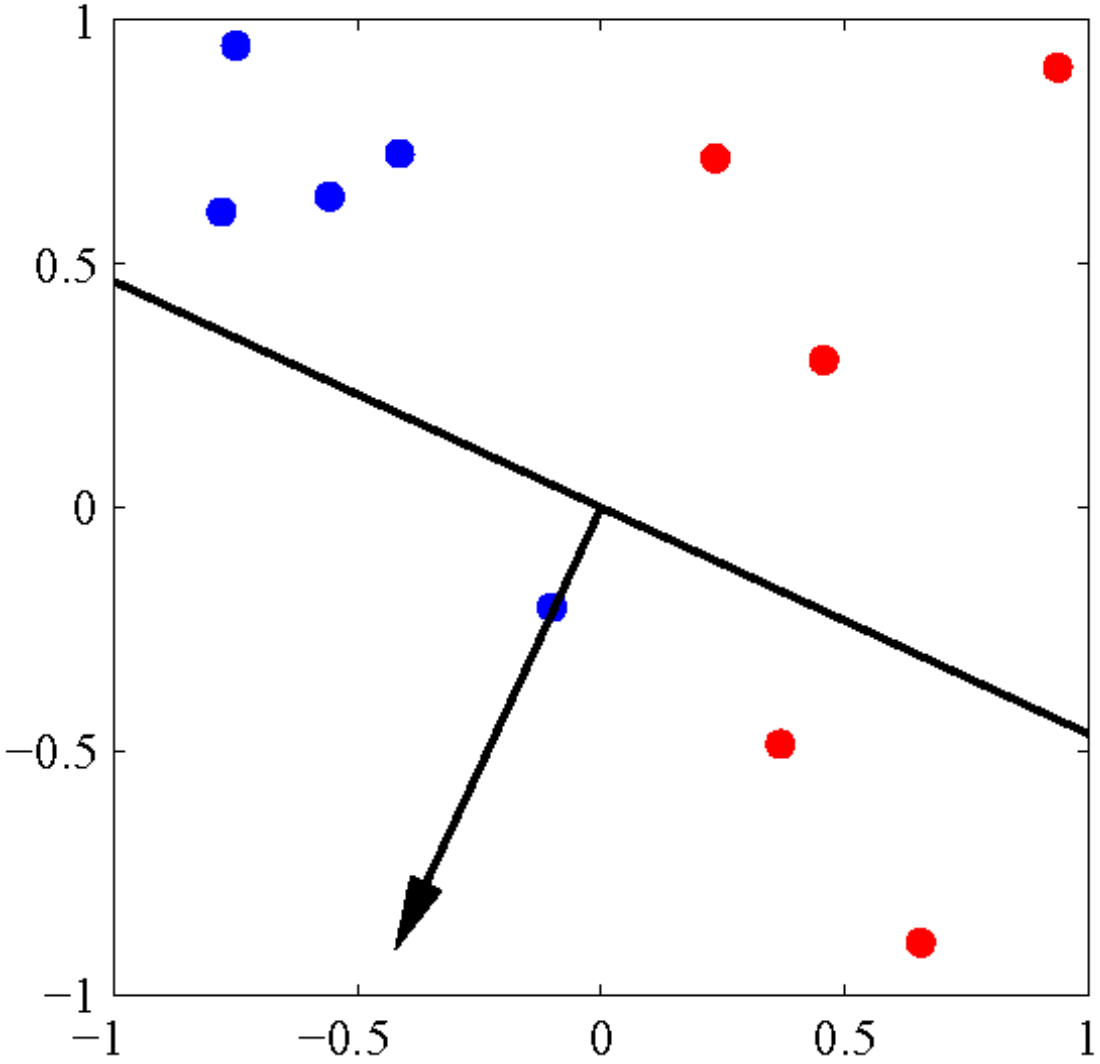
Perceptron algorithm learns weights from data

$$w(n+1) = w(n) + \Delta w(n)$$

$$= w(n) + \eta[d(n) - y(n)]x(n)$$

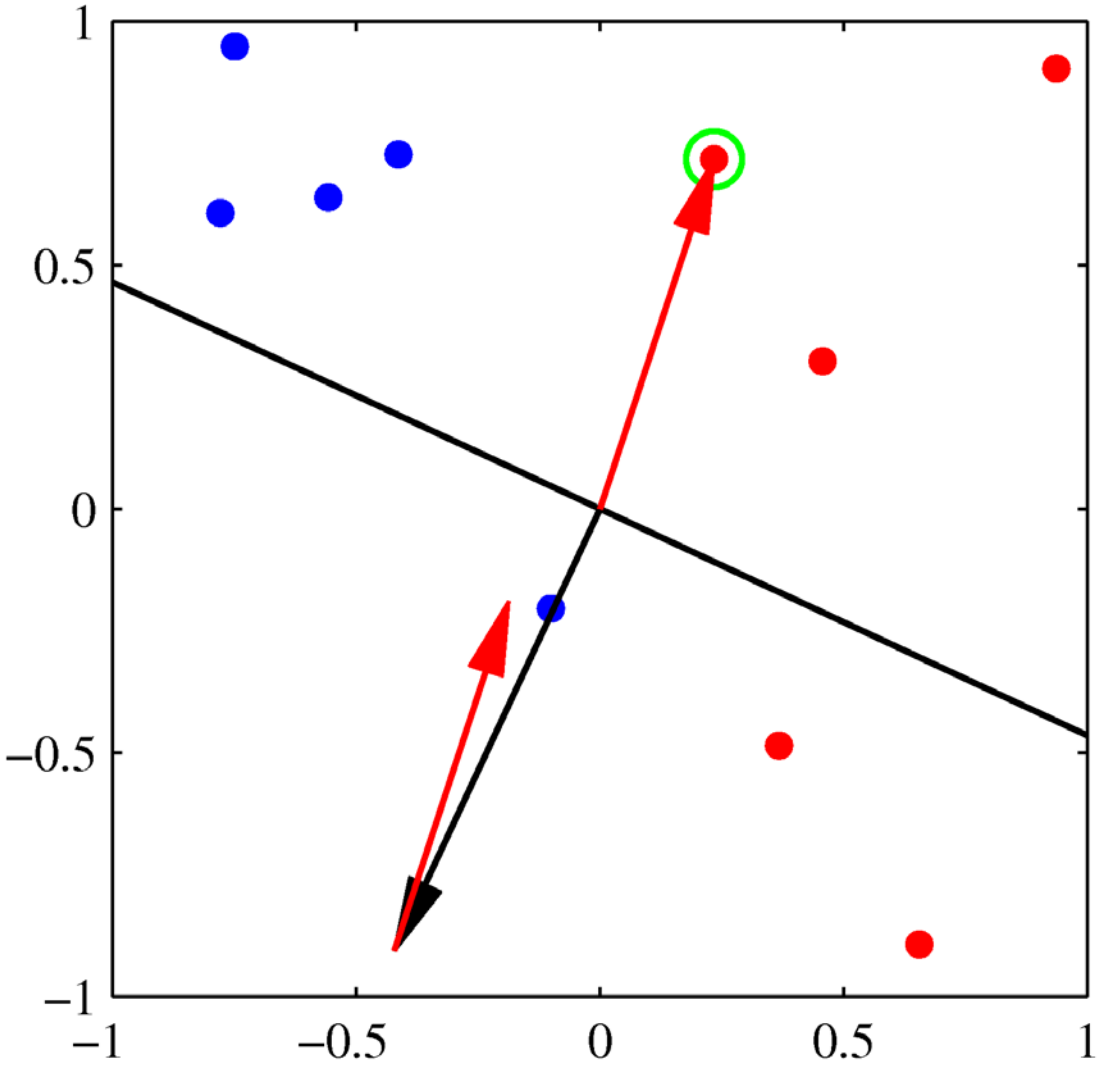
- n: iteration number, iterating over points in turn
- η: step size or learning rate, = 1 WLOG
- Only updates w when y(n) is incorrect

Visualization of perceptron learning



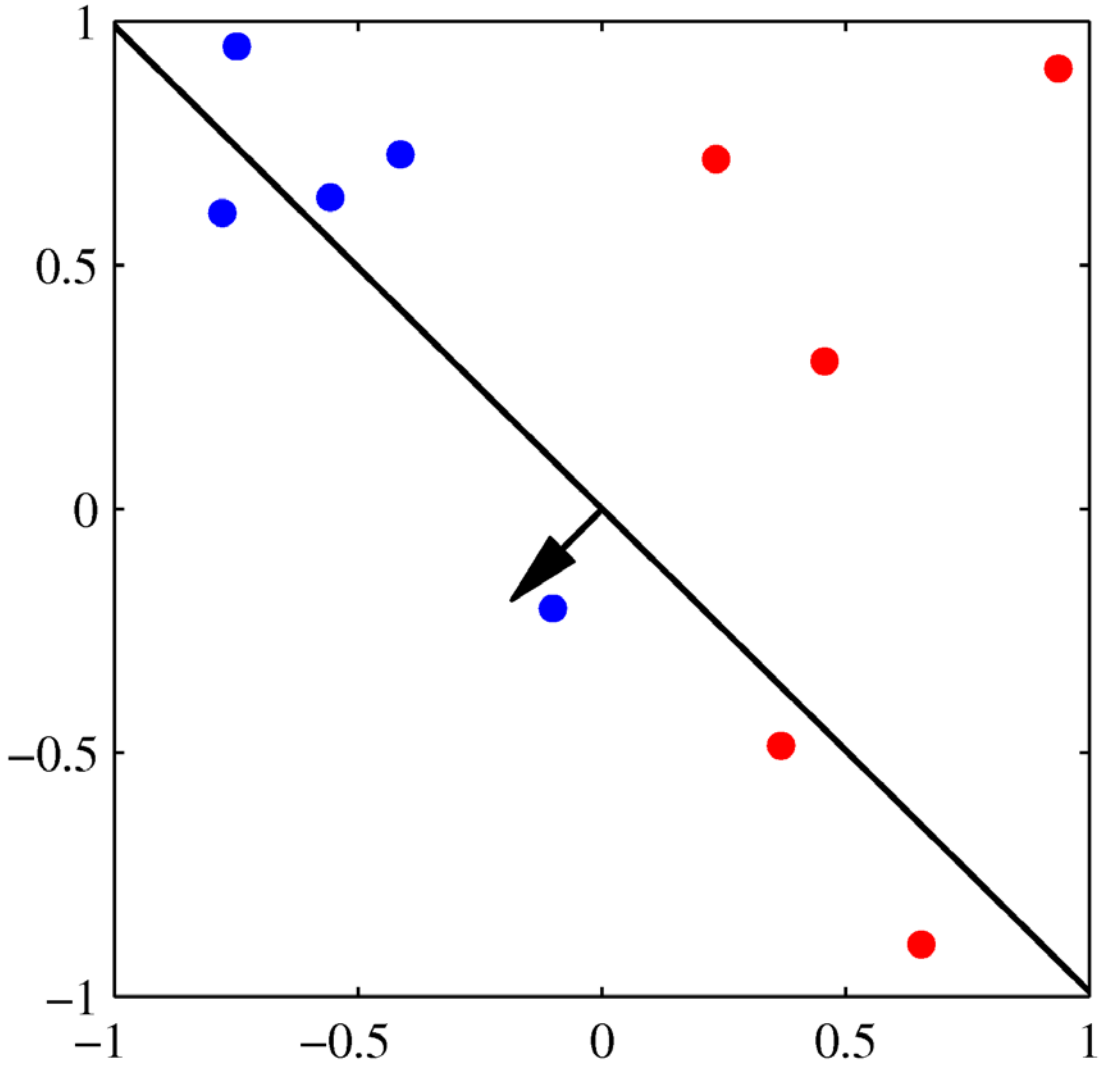
From Bishop (2006)

Visualization of perceptron learning



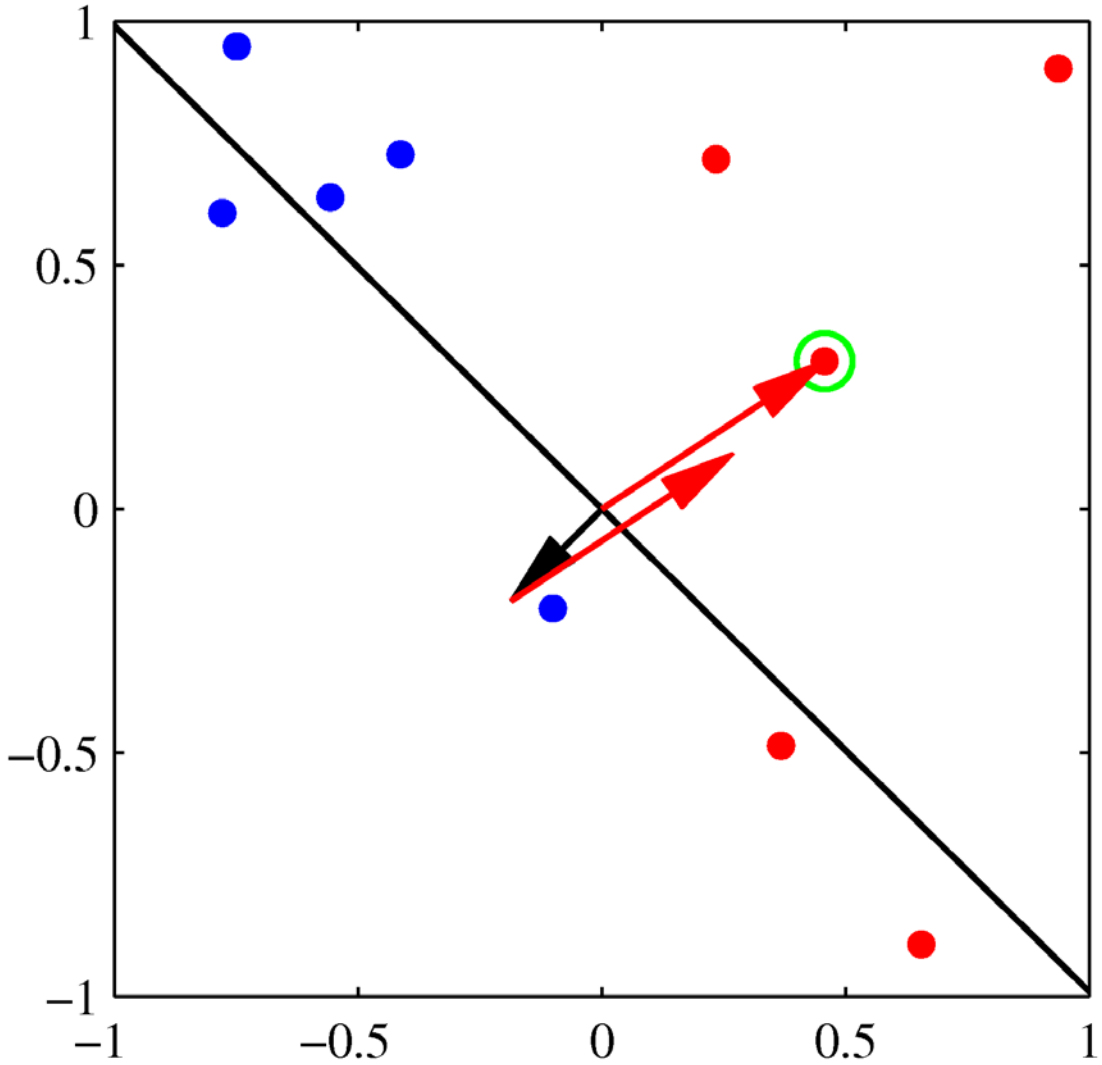
From Bishop (2006)

Visualization of perceptron learning



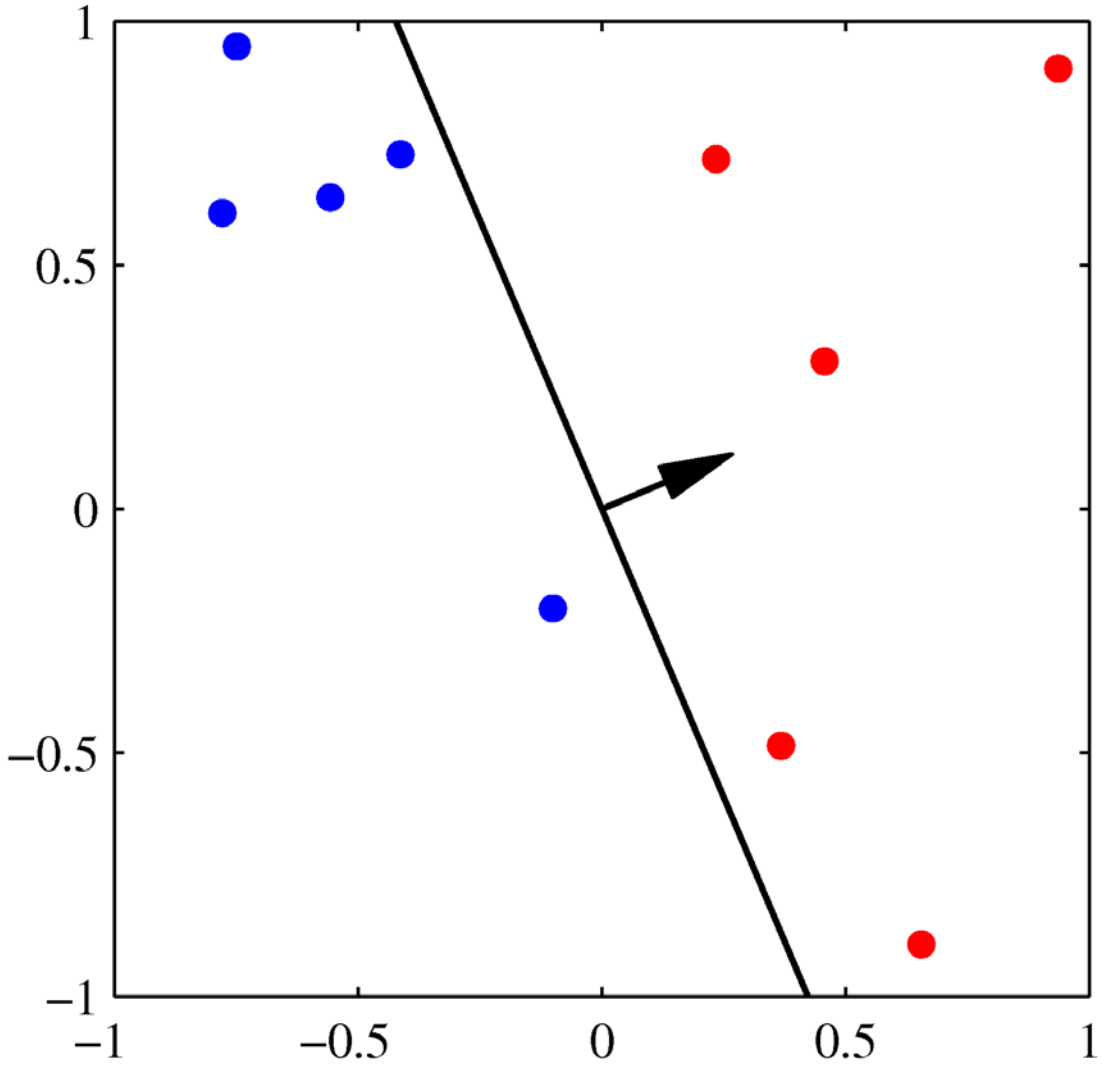
From Bishop (2006)

Visualization of perceptron learning



From Bishop (2006)

Visualization of perceptron learning



From Bishop (2006)

Summary of perceptron learning algorithm

- Definition
  - $w(n)$: (m+1)-by-1 weight vector (including bias) at step n
- Inputs
  - $x(n)$: n^{th} (m+1)-by-1 input vector with first element = 1
  - $d(n)$: n^{th} desired response
- Initialization: set w(0) = 0
- Repeat until no points are mis-classified
  - Compute response: $y(n) = \text{signum}\{w(n)^T x(n)\}$
  - Update: $w(n+1) = w(n) + [d(n) - y(n)]x(n)$

Perceptron learning can be interpreted as gradient descent

- Consider the total amount by which a neuron mis-classifies all of the points

$$E(w) = -\sum_p \left(d_p - y_p\right) w^T x_p$$

- Then the gradient of this WRT w is

$$\nabla_w E(w) = -\sum_p \left(d_p - y_p\right) x_p$$

- So the gradient descent update is

$$w(n+1) = w(n) - \eta \nabla_w E = w(n) + \left(d_p - y_p\right) x_p$$

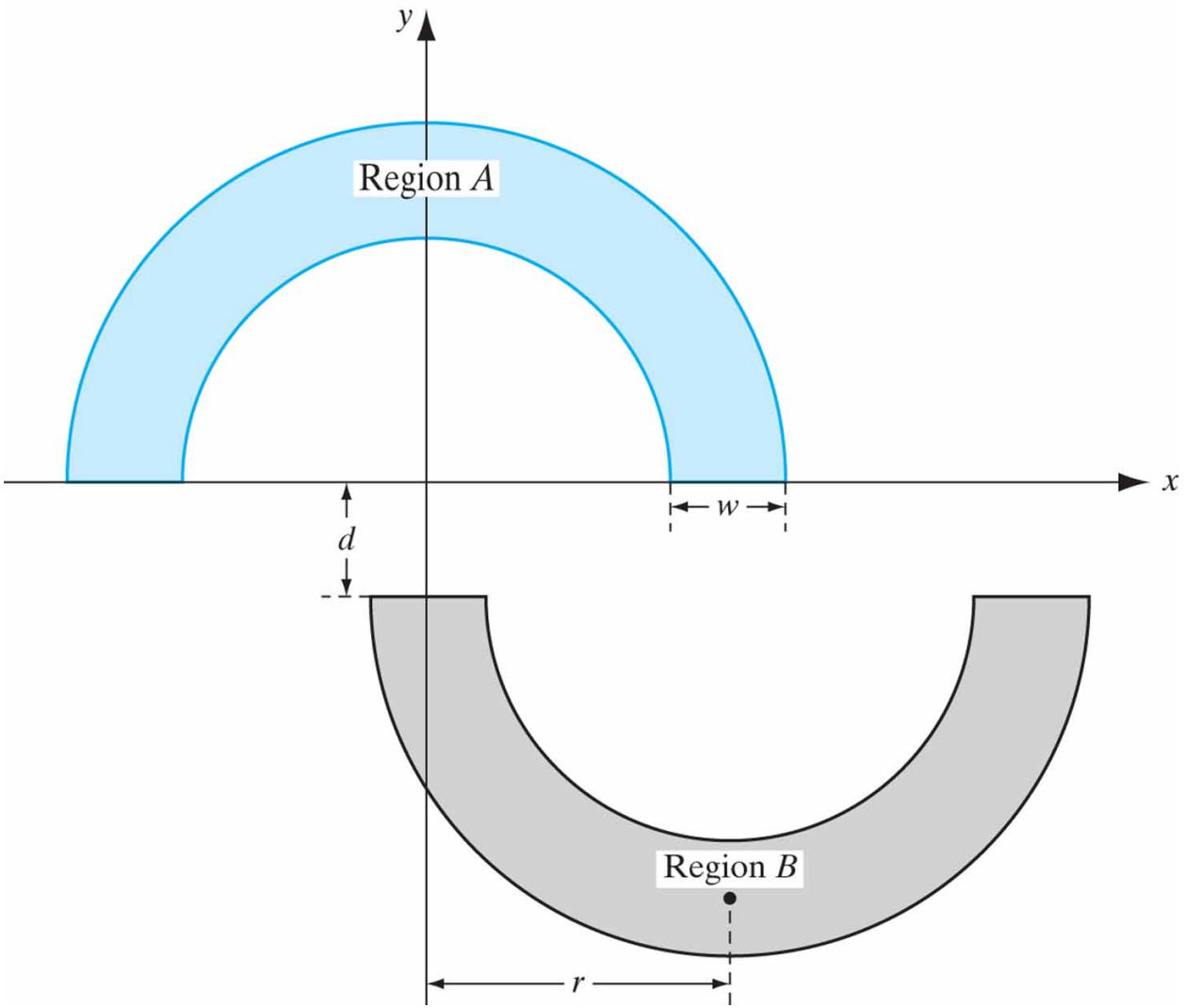
Perceptron convergence theorem

- Theorem:
  - Assume that there exists some unit vector w_0 and some α such that $d(n)w_0^T x(n) \geq \alpha$
    - i.e., the data are linearly separable
  - Assume also that there exists some R such that $\|x(n)\| = \sqrt{x(n)^T x(n)} \leq R \quad \forall n$
    - i.e., the data lie within a sphere of radius R
  - Then the perceptron algorithm makes at most $\frac{R^2}{\alpha^2}$ errors
    - i.e., it converges in at most $\frac{R^2}{\alpha^2}$ iterations

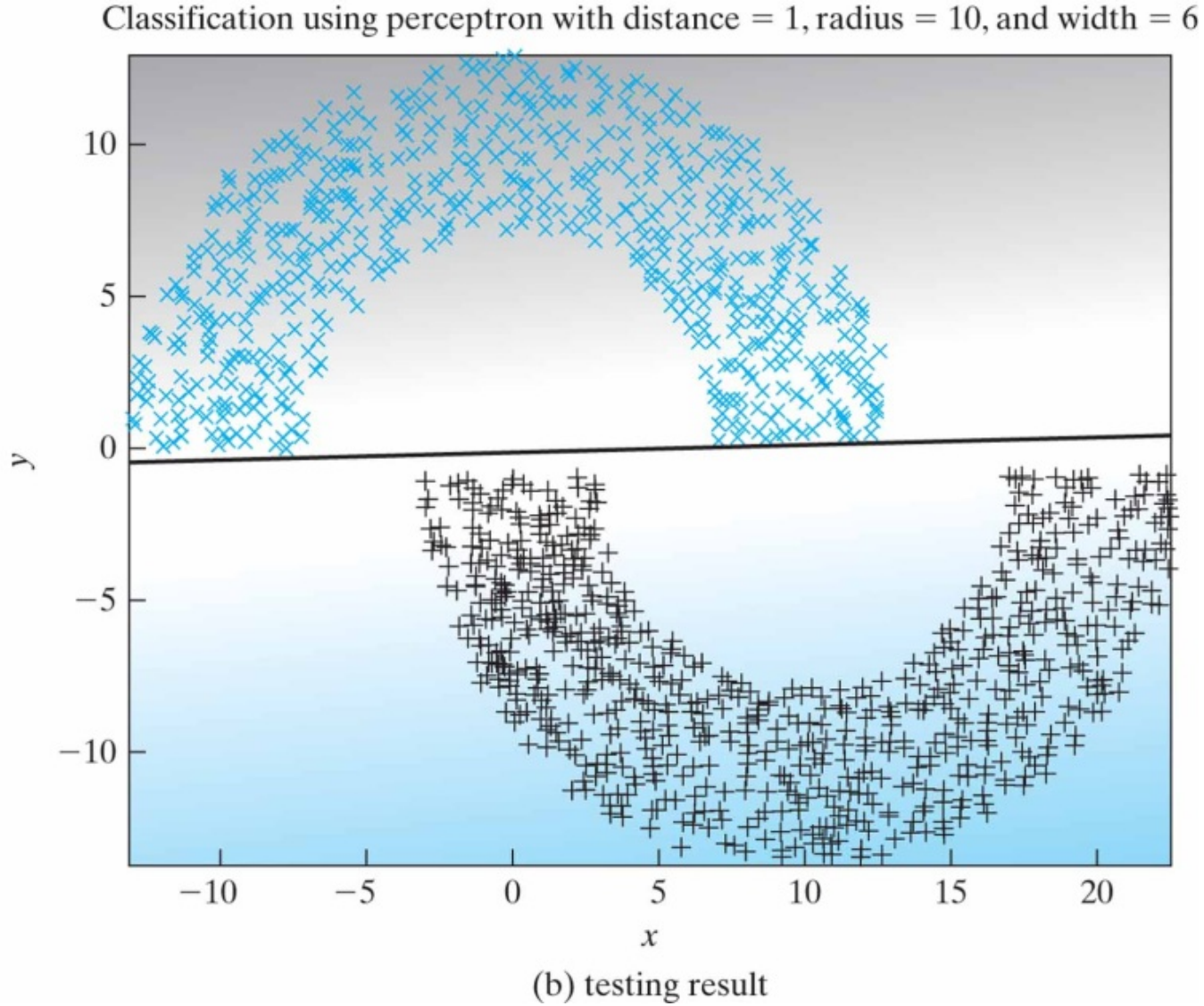
Perceptron convergence proof sketch

- Define w_k as the parameter vector when the algorithm makes its k^{th} error (note $w_1 = 0$)
- First show $k\alpha \leq \|w_{k+1}\|$ by induction
  - The weight vector grows in length proportionally with k
  - because of the separability of the data
- Second show $\|w_{k+1}\|^2 \leq kR^2$ by induction
  - But it can grow no faster than $\sqrt{k}$
  - because of the radius of the data
- Then it follows that $k \leq R^2/\alpha^2$
  - The perceptron makes a finite number of errors

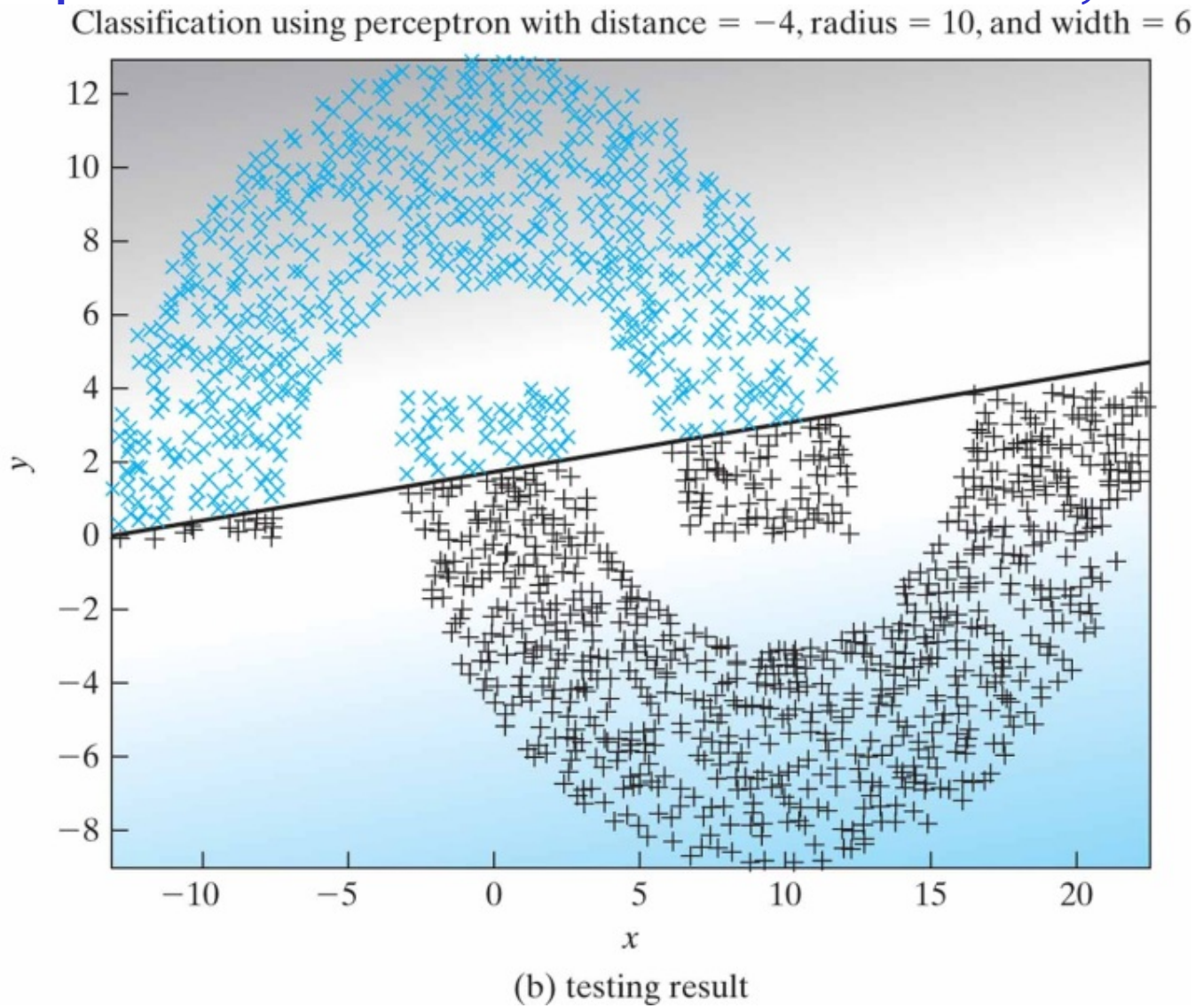
The double-moon classification problem



Perceptron learns double-moon, $d = 1$



Perceptron does not learn double-moon, d = -4



Linear regression has a closed-form solution

- Predict desired output d_p
- As a linear function of observations, $\boldsymbol{x}_p$

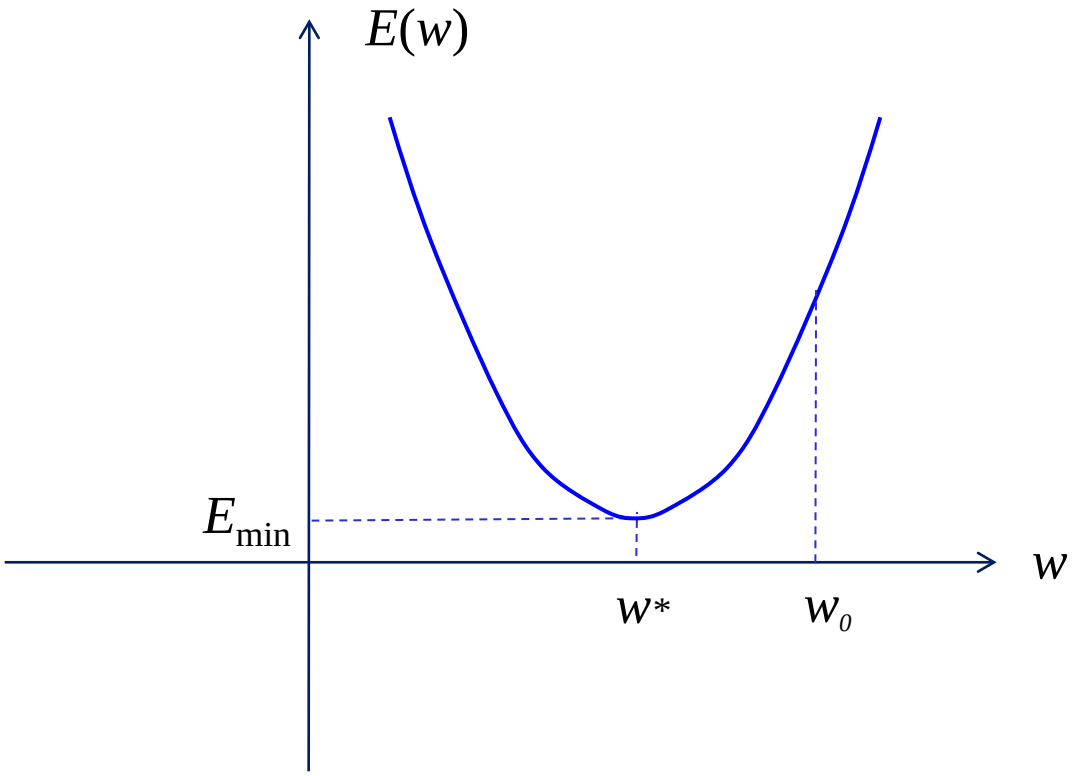
$$y_p = \boldsymbol{w}^T \boldsymbol{x}_p$$

- Find parameters $\boldsymbol{w}$ that minimize the mean square error of the predictions

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_p \left(d_p - y_p\right)^2$$

- Set gradient of error WRT $\boldsymbol{w}$ to 0
  - Solve for $\boldsymbol{w}$ analytically

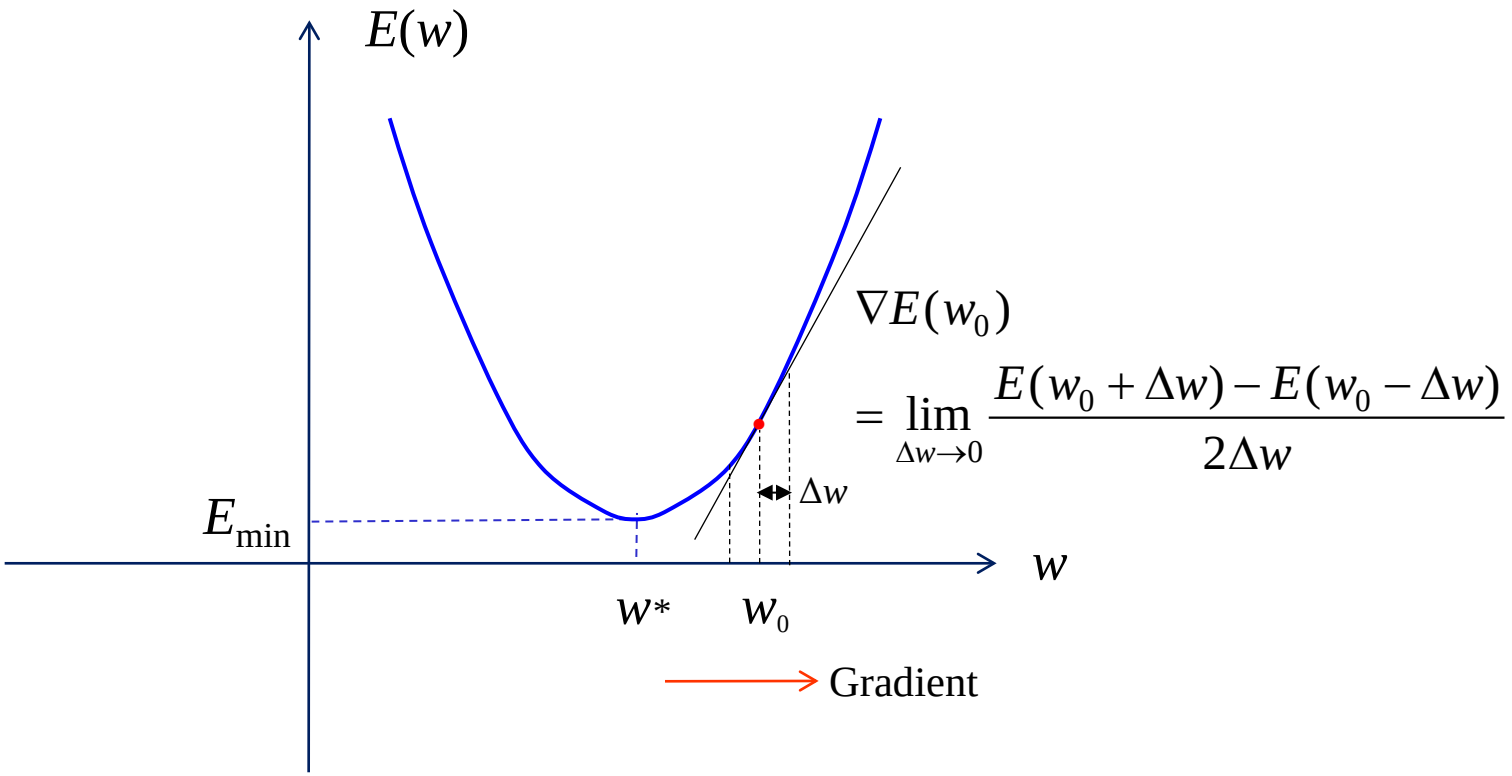
The mean square error defines a parabolic cost function



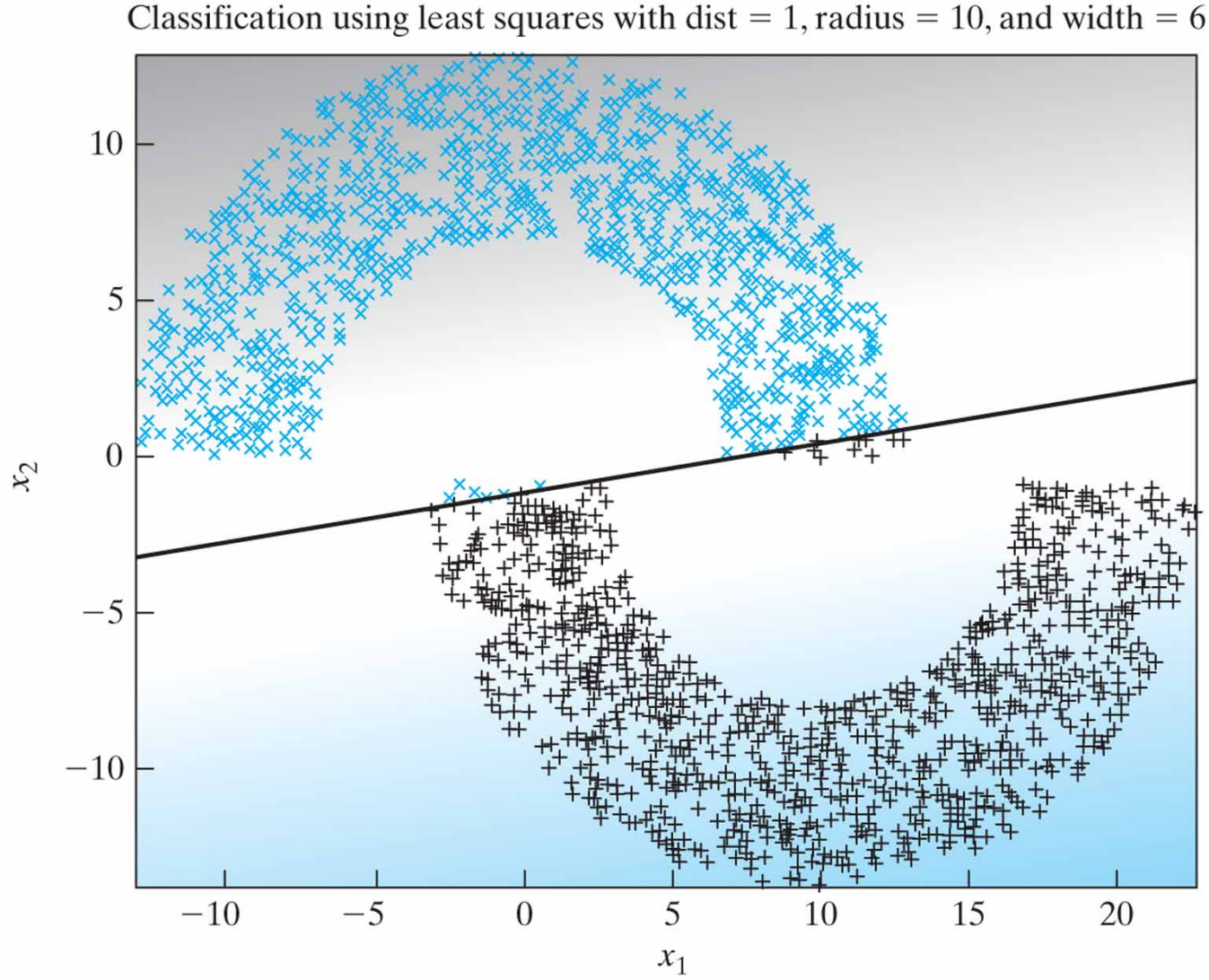
Optimal parameters can be found via search

- Often there is no closed form solution for
$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = 0$$
- We can still use the gradient in a numerical solution
- This is called gradient descent
$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \eta \nabla_{\boldsymbol{w}} E(\boldsymbol{w})$$
- At the minimum of $E(\boldsymbol{w})$, the gradient is 0
  - And $\boldsymbol{w}$ stays constant because $\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - 0$

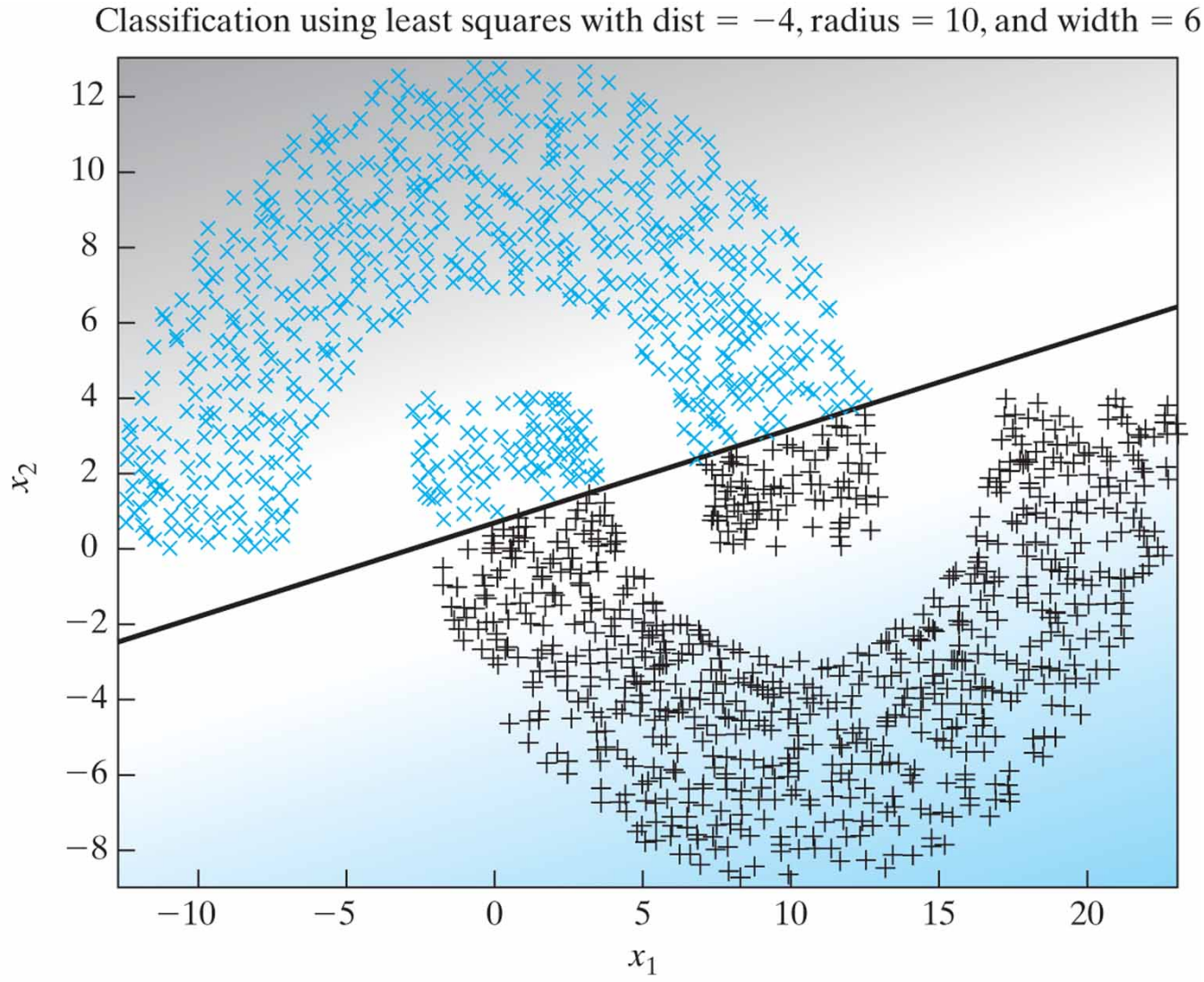
The gradient is the slope and direction of steepest ascent of the error function



Least squares classification works pretty well for double-moon, d = 1



Least squares classification works less well for double-moon, d = -4



LMS algorithm solves least squares on-line

- Stochastic gradient descent solution to linear regression is called the LMS algorithm
- Minimizes the error on one data point at a time

$$2E_p(w) = e_p^2(\boldsymbol{w}) = \left(d_p - y_p\right)^2 = \left(d_p - \boldsymbol{w}^T\boldsymbol{x}_p\right)^2$$

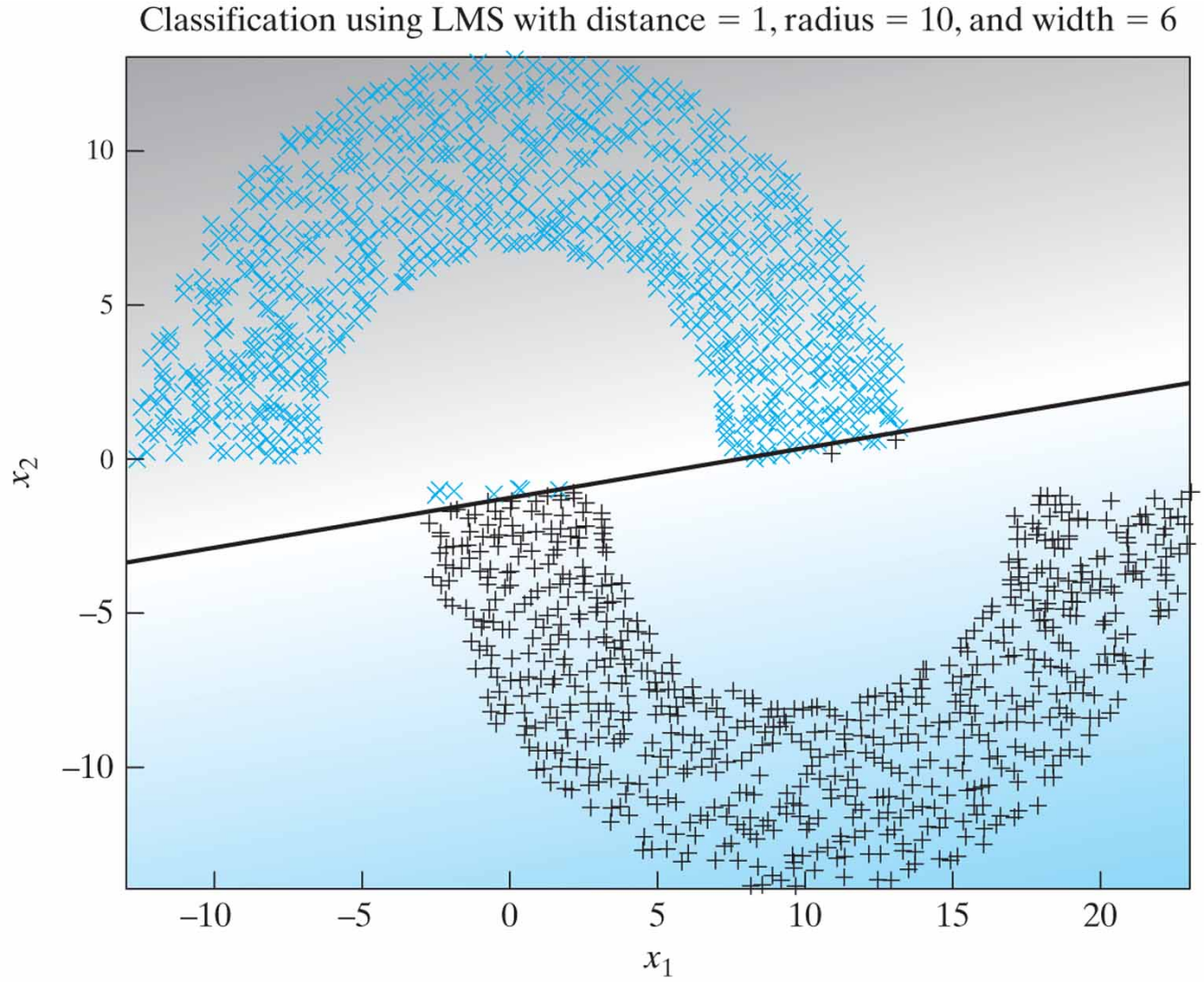
- The gradient is

$$\nabla_{\boldsymbol{w}} E_p(\boldsymbol{w}) = -\left(d_p - \boldsymbol{w}^T\boldsymbol{x}_p\right)\boldsymbol{x}_p = -e_p(\boldsymbol{w})\boldsymbol{x}_p$$

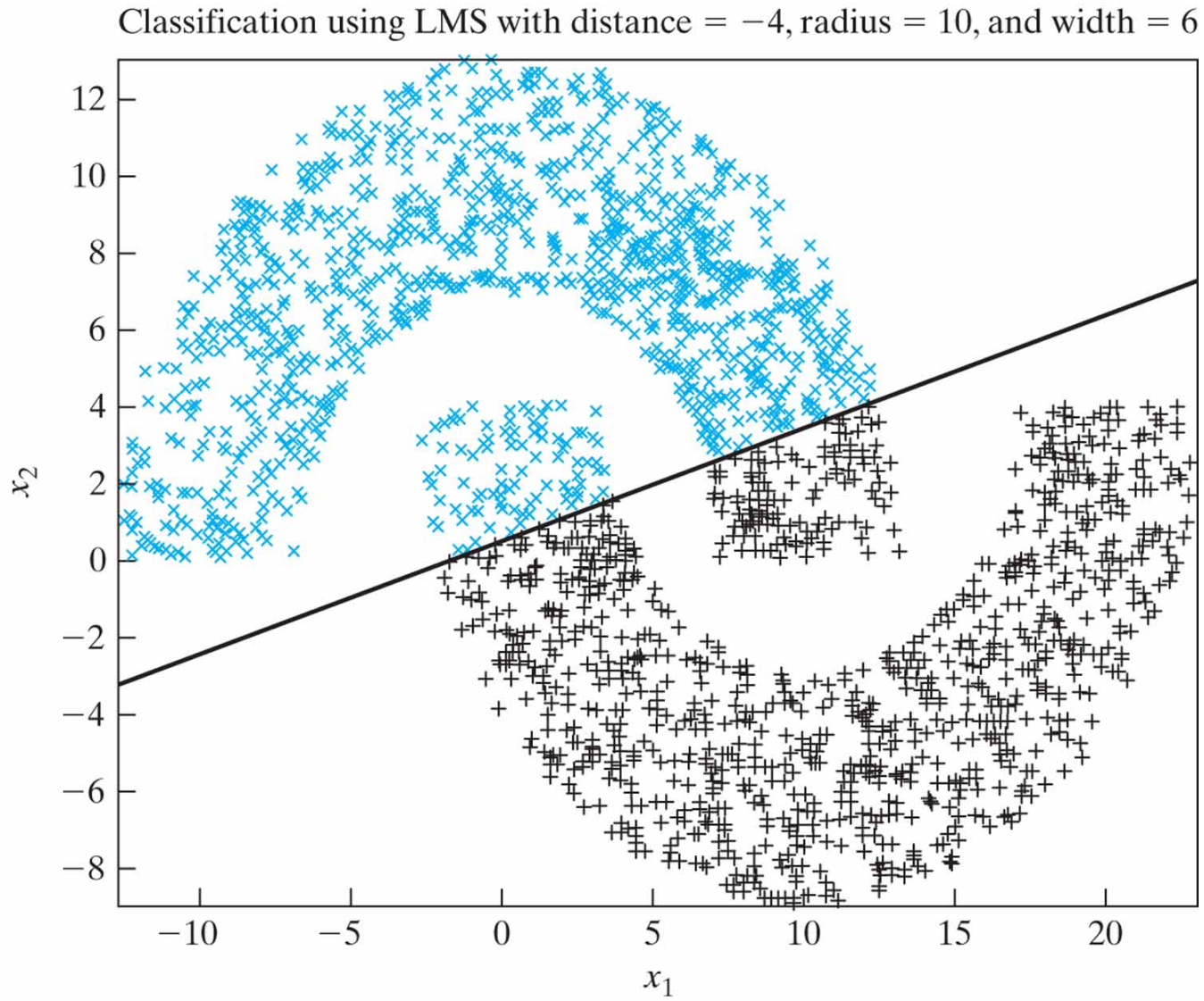
- So the LMS update is

$$\begin{aligned}\boldsymbol{w}(n+1) &= \boldsymbol{w}(n) - \eta\nabla_{\boldsymbol{w}} E_p(\boldsymbol{w}) \\ &= \boldsymbol{w}(n) + \eta e_p\,(\boldsymbol{w})\boldsymbol{x}_p\end{aligned}$$

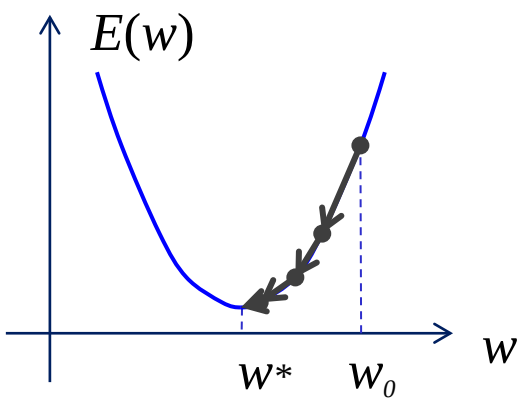
LMS achieves the least squares solution for double-moon, d = 1



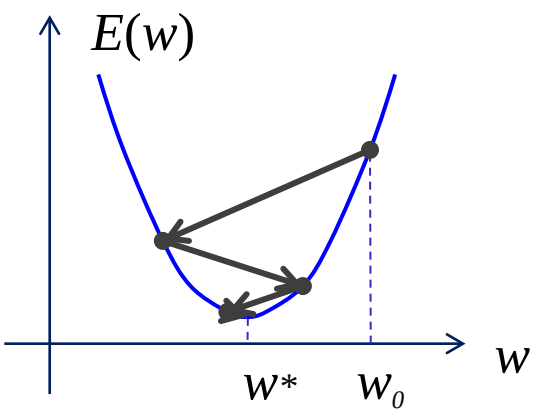
LMS achieves the least squares solution for double-moon, d=-4



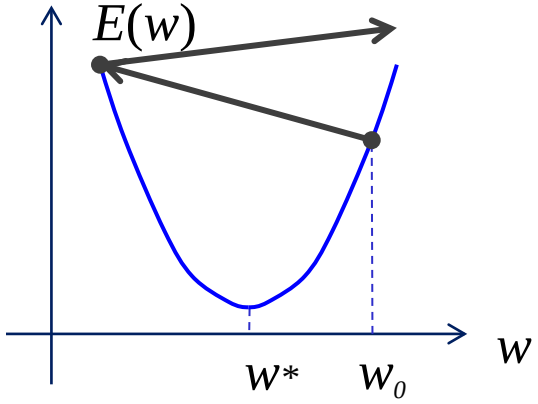
The optimal learning rate for a parabola is the reciprocal of the second derivative



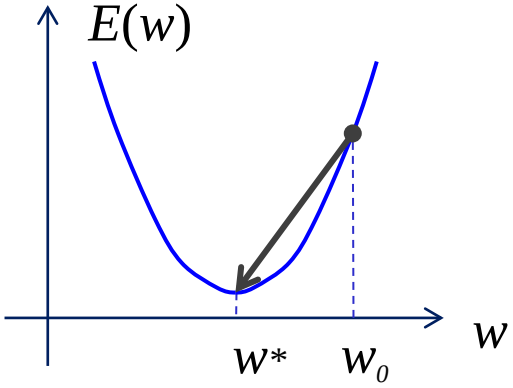
$\eta < (E''(w))^{-1}$



$\eta > (E''(w))^{-1}$

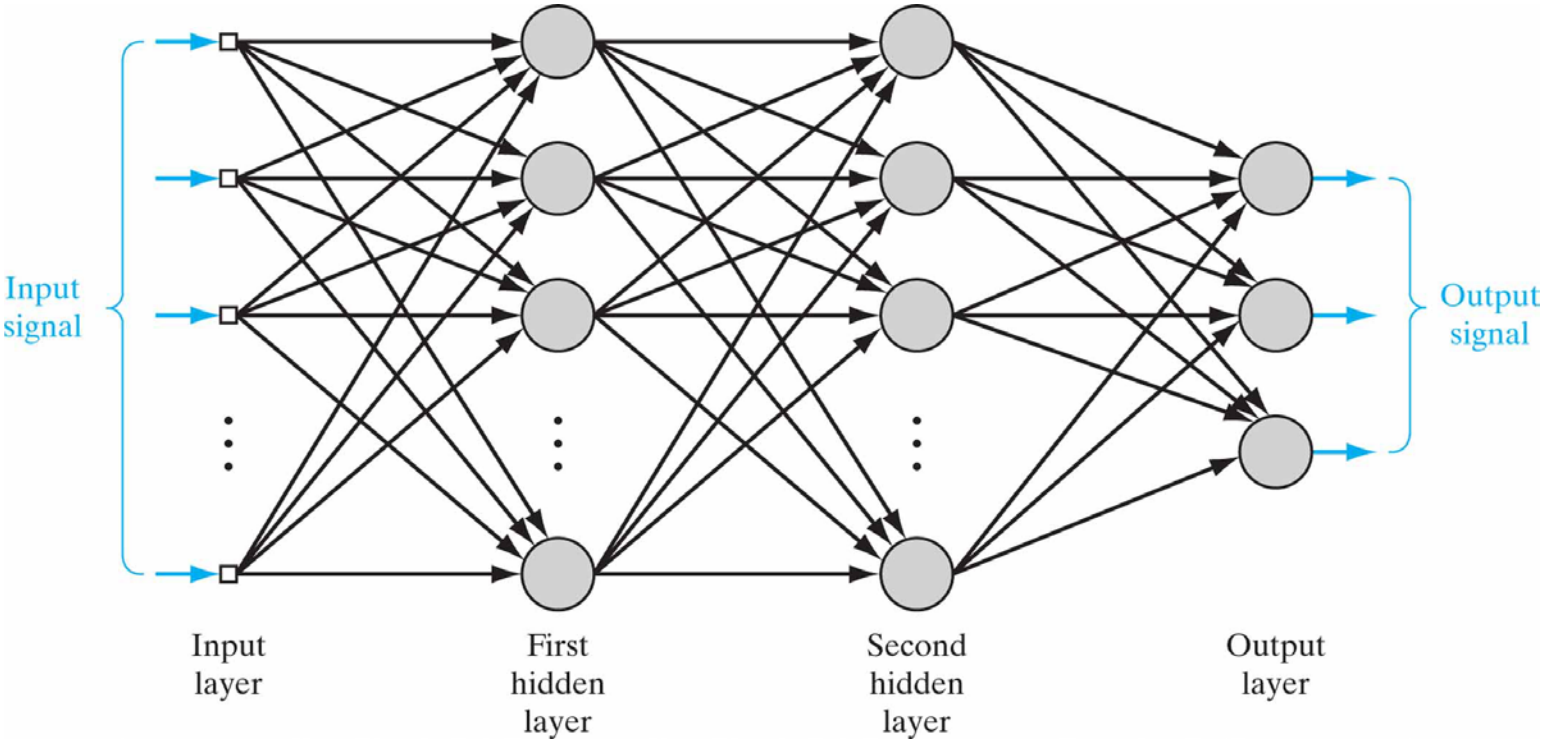


$\eta > 2(E''(w))^{-1}$



$\eta = (E''(w))^{-1}$

Multilayer perceptrons aren't really perceptrons



MLPs can be trained to minimize the MSE

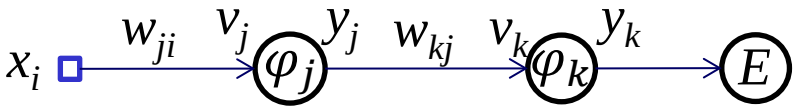
- Think of an MLP as a complicated, non-linear function of its input parametrized by $\boldsymbol{w}$:

$$\mathbf{y} = F(\boldsymbol{x}; \boldsymbol{w})$$

- Given a set of training data $\{\boldsymbol{x}_p, \boldsymbol{d}_p\}$, adjust $\boldsymbol{w}$ to minimize the mean square error of its predictions

$$\bar{E}(\boldsymbol{w}) = \sum_p E_p(\boldsymbol{w}) = \sum_p \frac{1}{2} \left\| \boldsymbol{d}_p - F(\boldsymbol{x}_p; \boldsymbol{w}) \right\|^2$$

Gradient descent in MLPs is called backprop



- Error assigned to each neuron

$$e_k = (d_k - y_k)$$

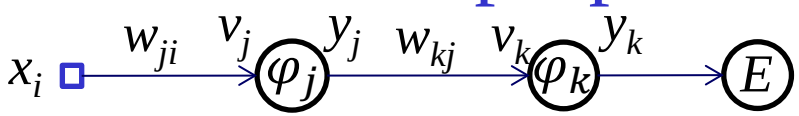
$$e_j = \sum_k e_k \varphi'(v_k) w_{kj}$$

- Gradients computed for each weight

$$\frac{\partial}{\partial w_{kj}} E(\boldsymbol{w}) = -e_k \varphi'(v_k) y_j$$

$$\frac{\partial}{\partial w_{ji}} E(\boldsymbol{w}) = -e_j \varphi'(v_j) x_i$$

Gradient descent in MLPs is called backprop



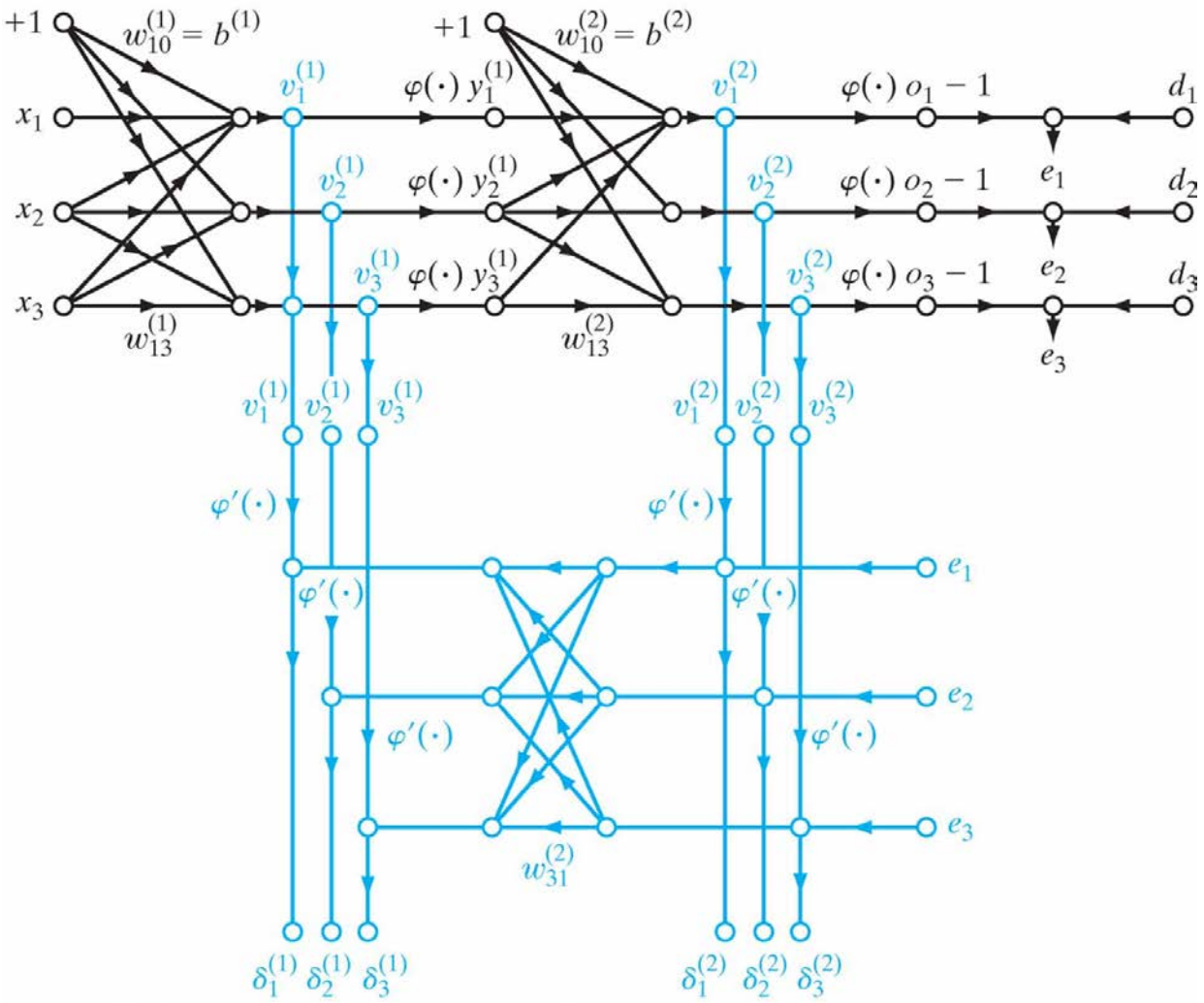
- So the weights are updated as

$$w_{kj}(n+1) = w_{kj}(n) + \eta e_k \varphi'(v_k) y_j$$

$$w_{ji}(n+1) = w_{ji}(n) + \eta e_j \varphi'\left(v_j\right) x_i$$

- Easy to extend to more layers
  - Although the gradient itself is less well behaved
  - So second-order methods are more necessary

Backprop can be visualized as a flow chart



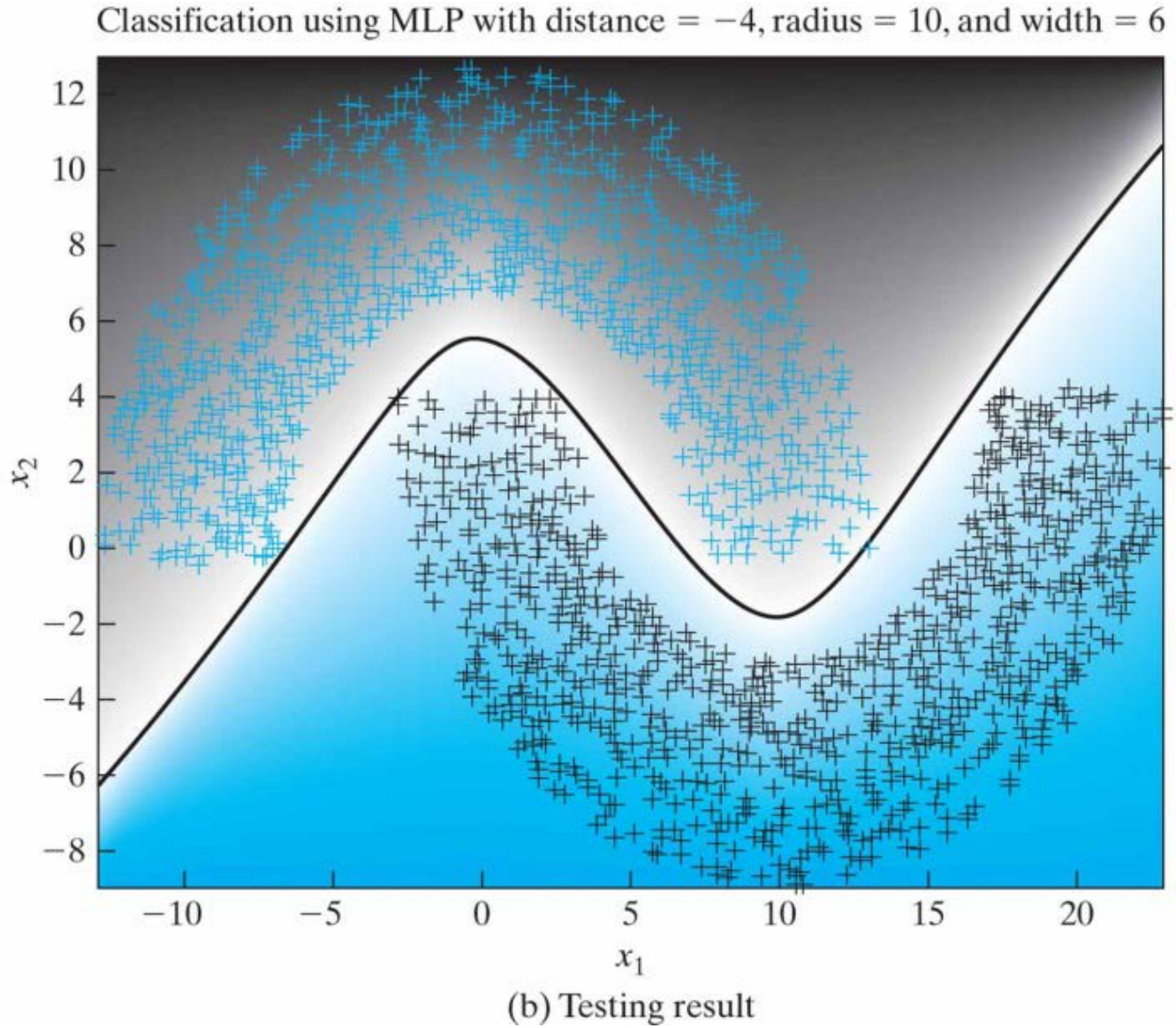
Must set several parameters to build an MLP

- Model parameters
  - Number of hidden layers
  - Number of units in each hidden layer
  - Activation function
  - Error function
- It is best to compare different settings empirically

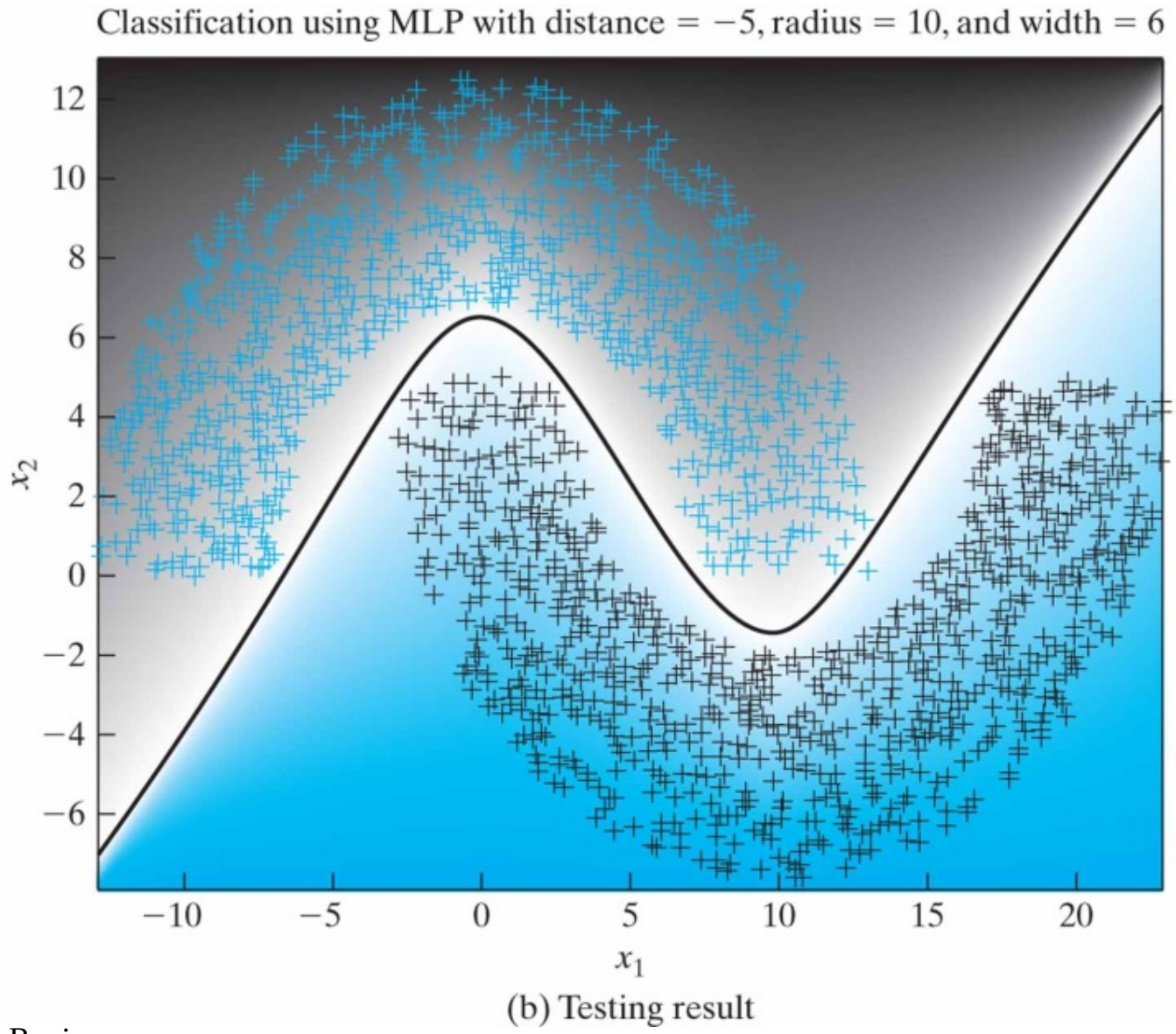
There are many optimization tricks for finding better local minima in backprop

- Momentum: mix in gradient from step $n - 1$
- Weight initialization: small random values
- Stopping criterion: early stopping
- Learning rate annealing: start large, slowly shrink
- Second order methods: use a separate η for each pair of parameters based on local curvature
- Randomize training example order
- Regularization: terms in $E(\boldsymbol{w})$ that only depend on $\boldsymbol{w}$

MLP learns double-moon, d = -4



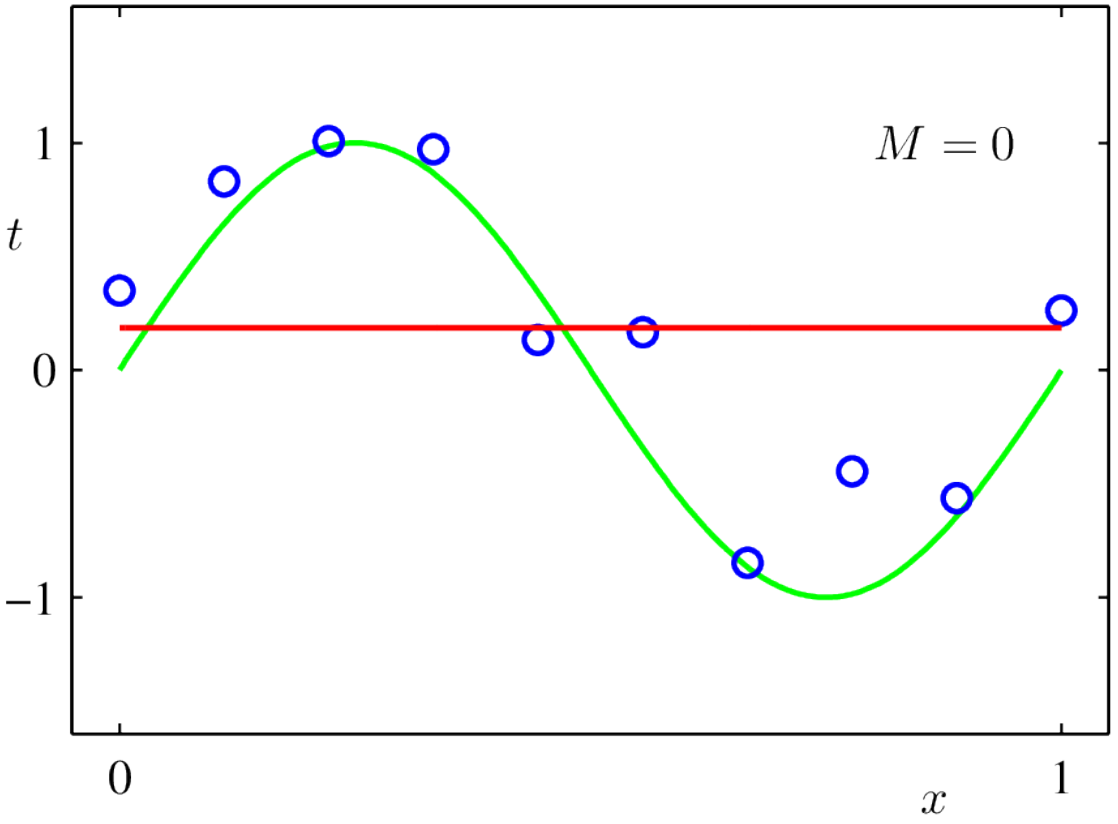
MLP learns double-moon, d = -5



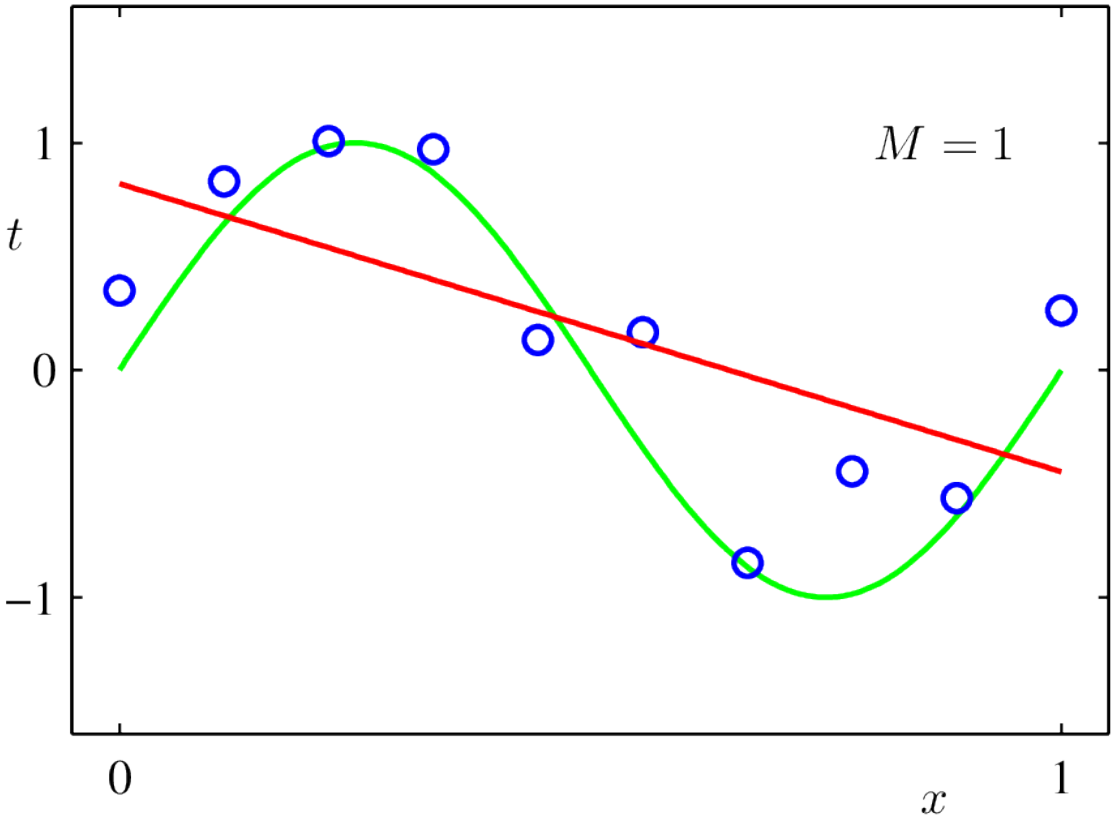
Our goal is to train models that generalize

- Models must be complex enough to capture important variations in the training data
- But not so complex that they capture the random variations in the training data
- We evaluate generalization by measuring performance on a held-out test or validation set

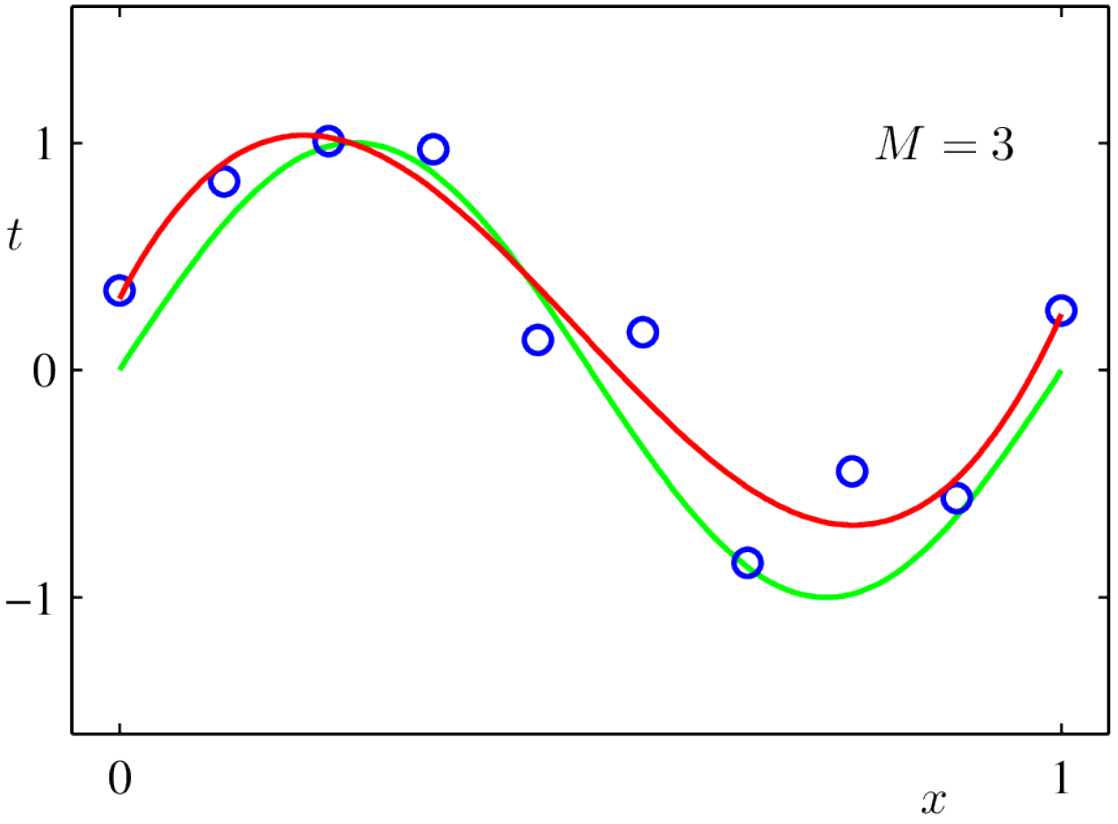
Polynomial of order 0
cannot capture important variations



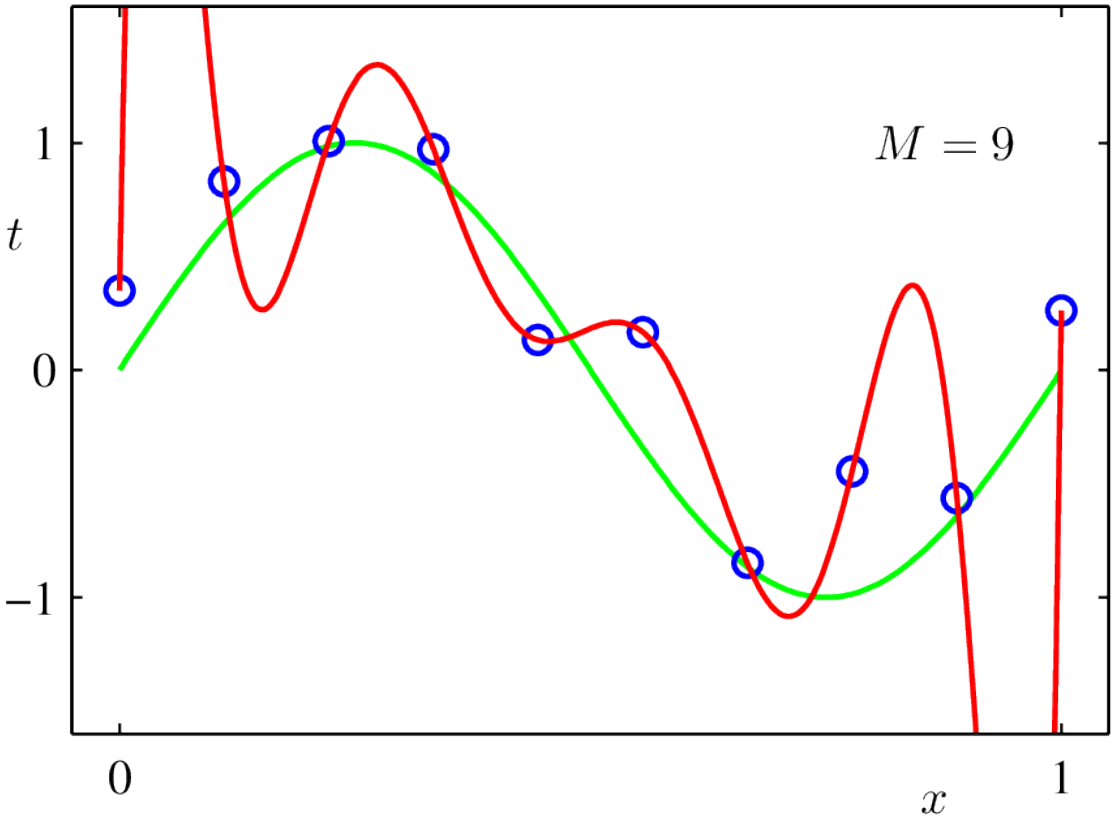
Polynomial of order 1
cannot capture important variations



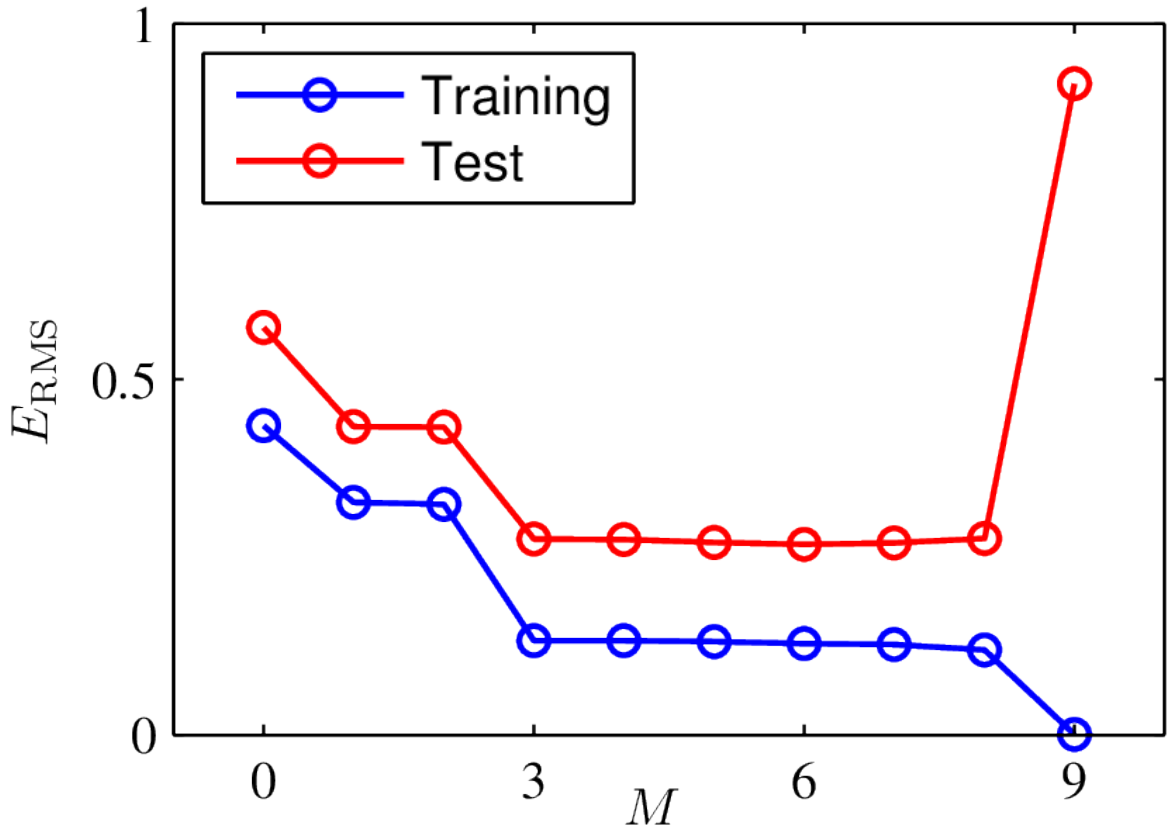
Polynomial of order 3 can capture important variations



Polynomial of order 9 captures unimportant variations

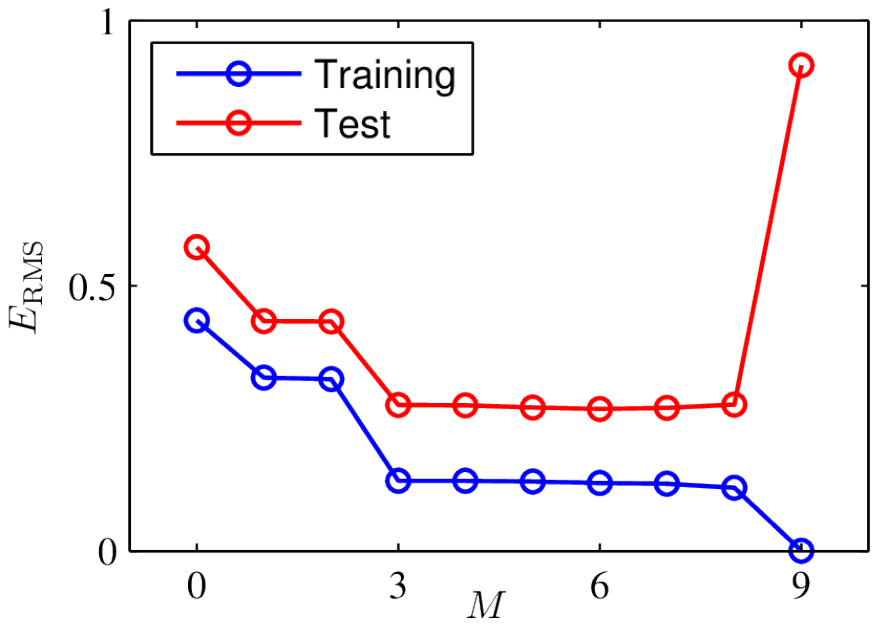


Learning curves can tell you whether a model is too complex or too simple



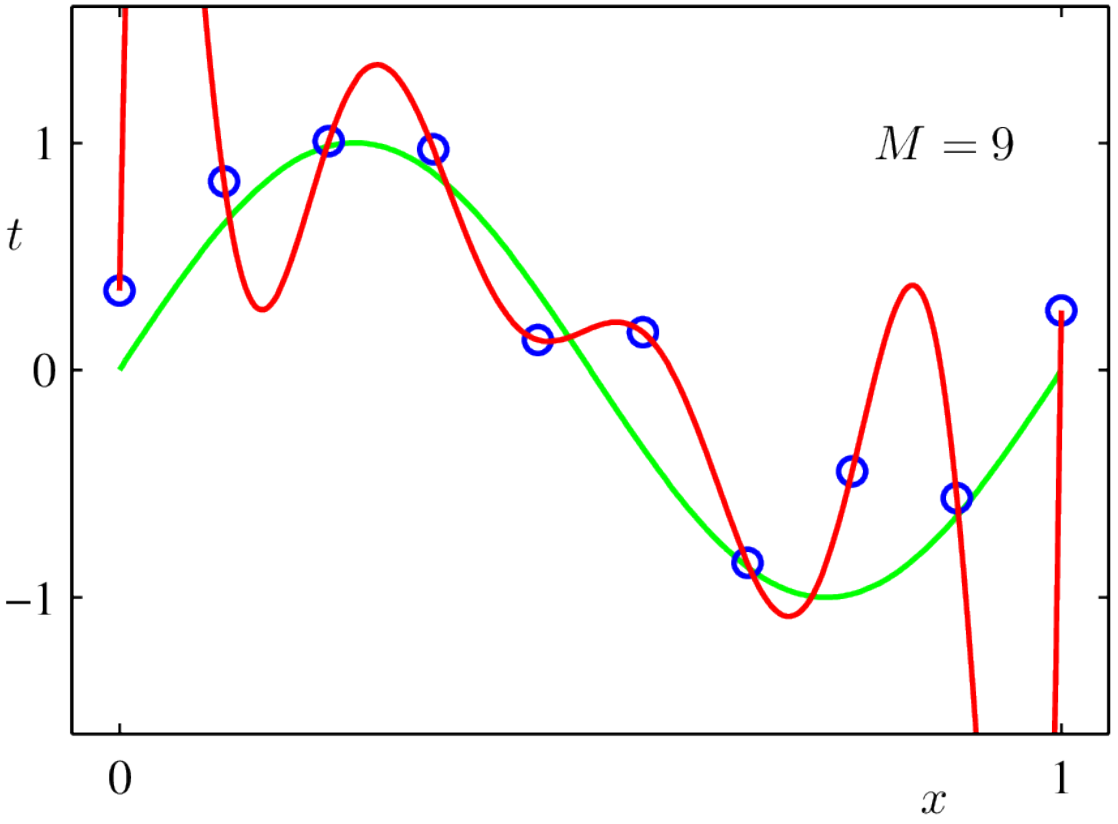
Simple models under-fit
complex models over-fit

	Under-fit	Good fit	Over-fit
Training error	High	Low	Low
Testing error	High	Low	High



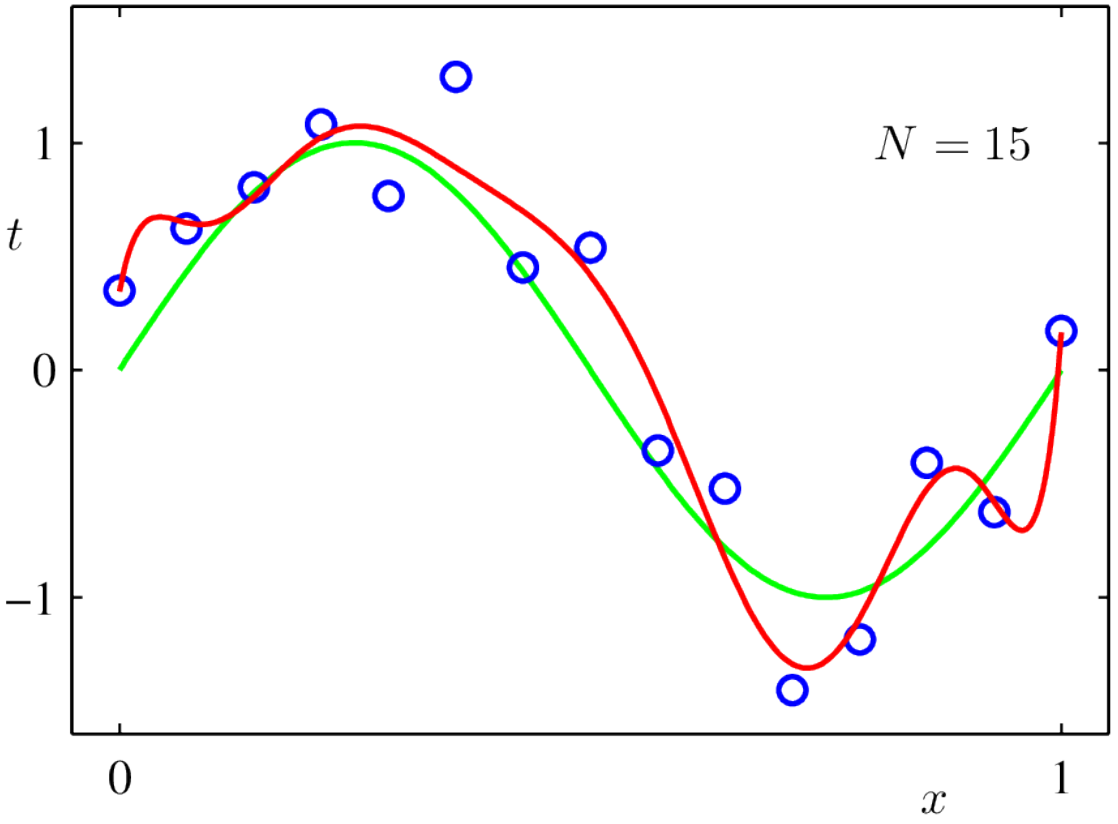
Fit is relative to the amount of training data

- Polynomial of order 9 fit to 10 points



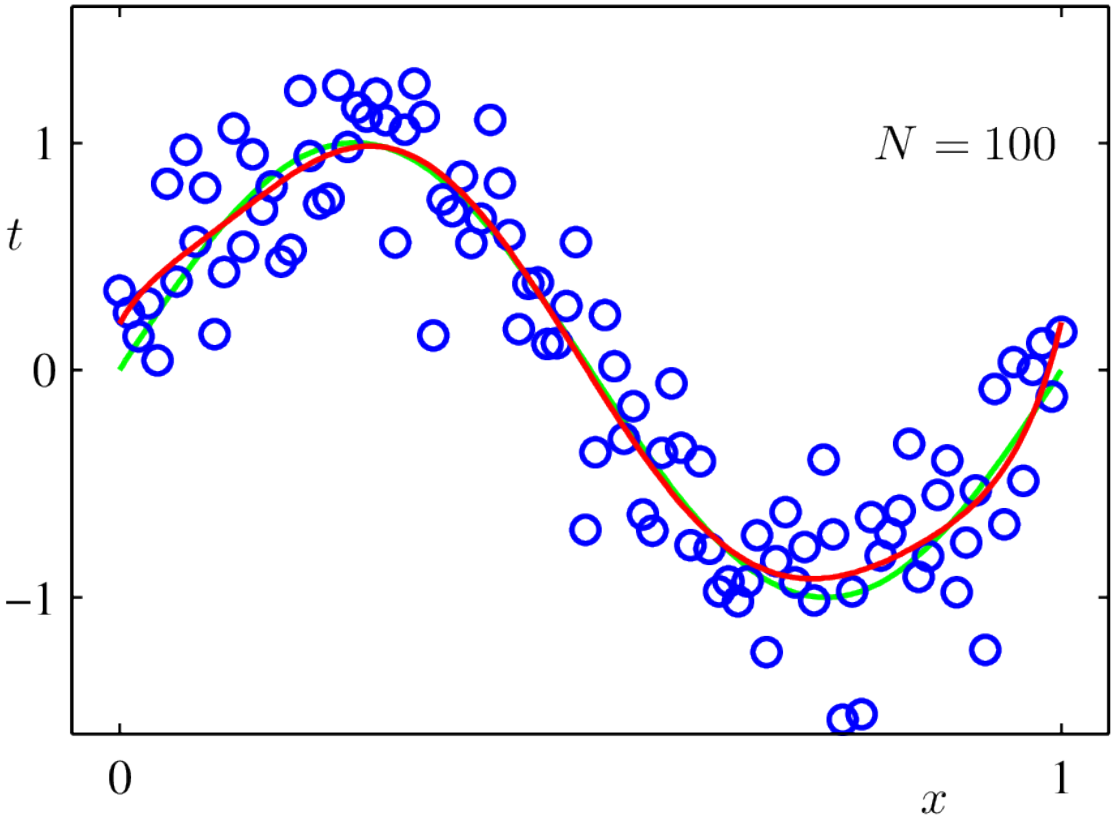
Fit is relative to the amount of training data

- Polynomial of order 9 fit to 15 points

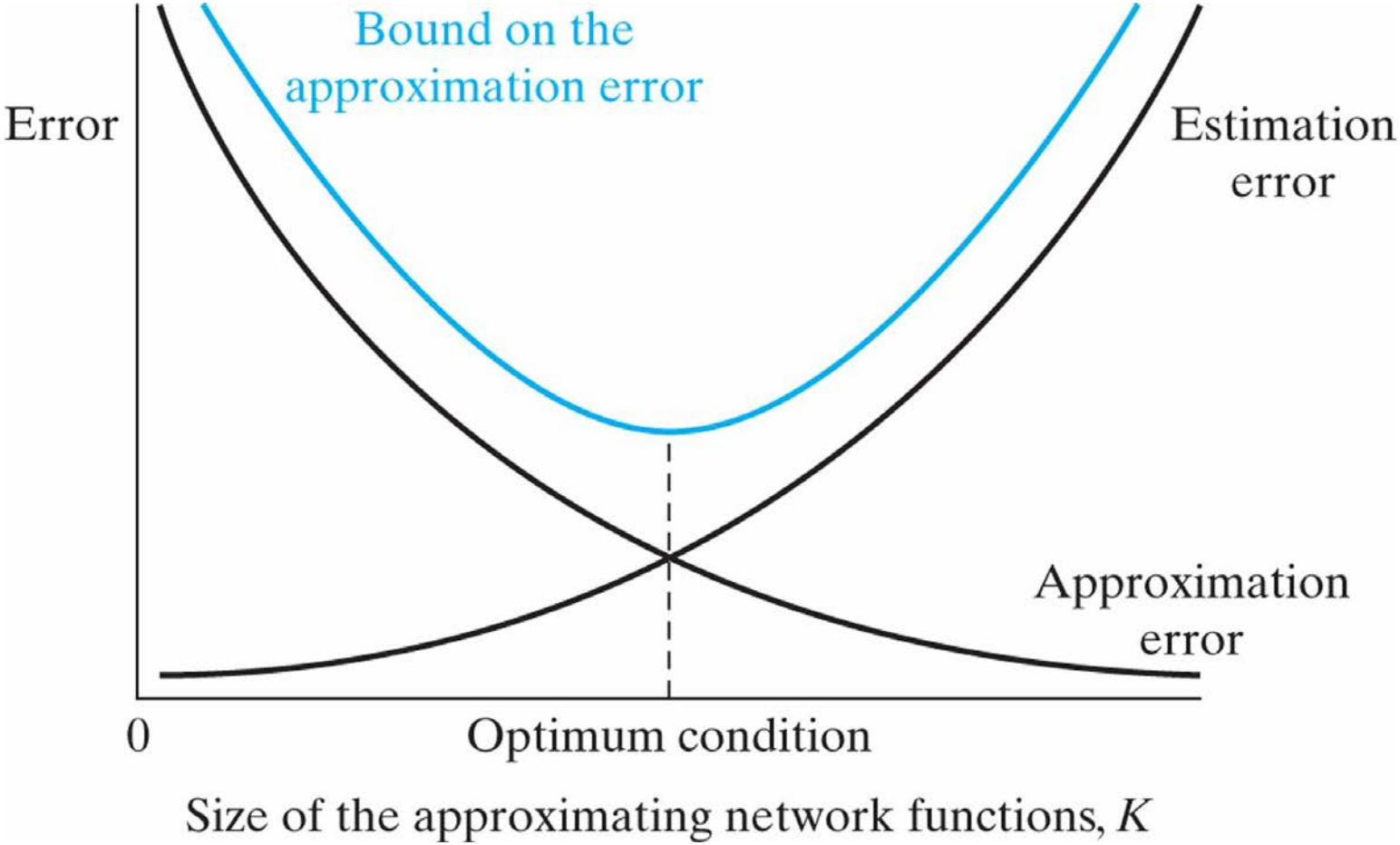


Fit is relative to the amount of training data

- Polynomial of order 9 fit to 100 points



Simple networks are dominated by bias complex networks are dominated by variance



Function approximation

- Think of the MSE as a measure of goodness of fit for function approximation
- We have discussed several function approximators

Model	$\boldsymbol{y_p = f(x_p)}$
M-P Neuron	$y = \text{signum}(\boldsymbol{w}^T \boldsymbol{x})$
Linear regression	$y = \boldsymbol{w}^T \boldsymbol{x}$
MLP	$y_k = \varphi\left(\sum_j w_{kj}\, \varphi\left(\sum_i w_{ji} x_i\right)_j\right)_k$
RBF network	$y = \sum_k w_k \varphi\left(\|\boldsymbol{x} - \boldsymbol{\mu}_k\|\right)$

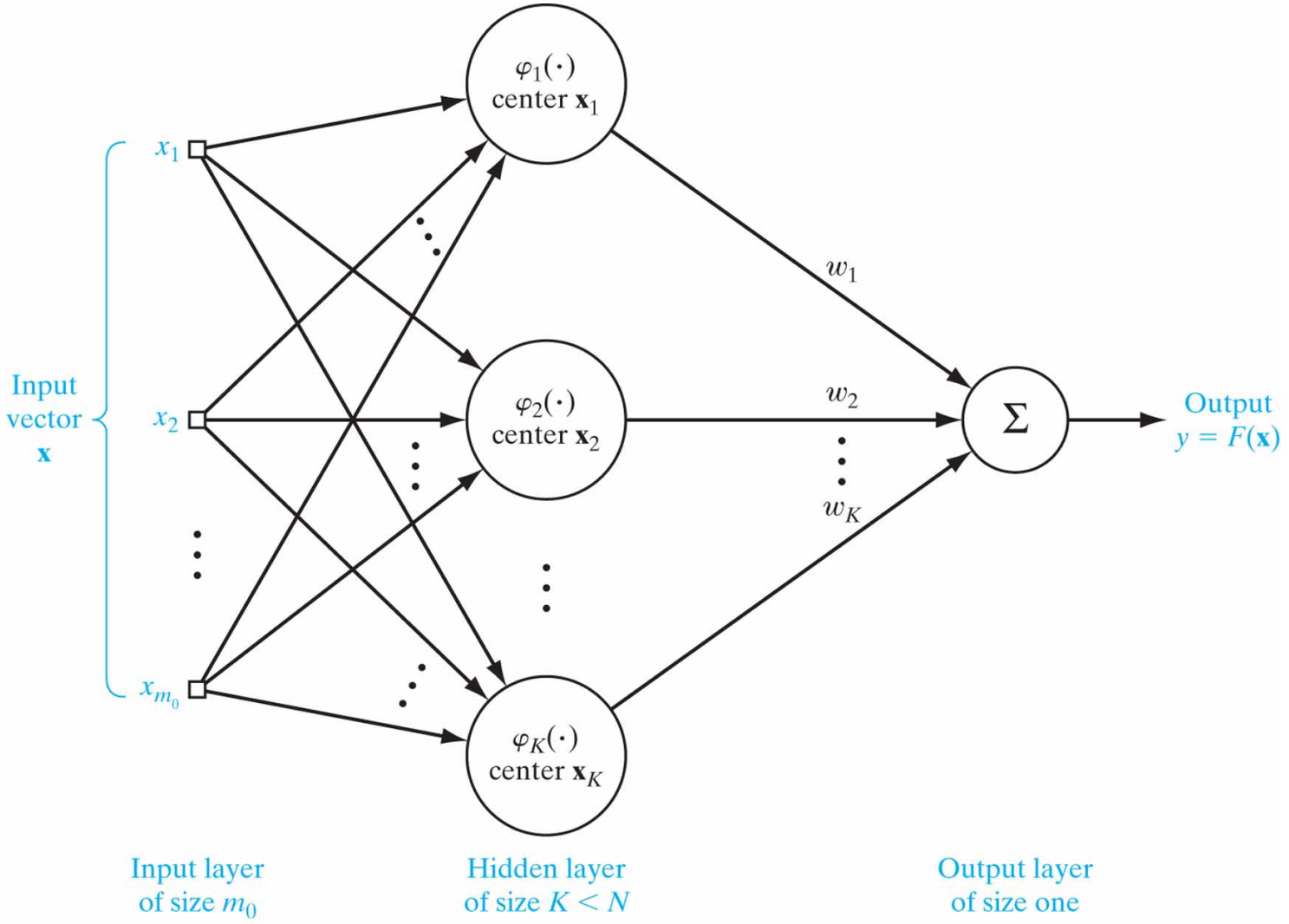
Linear projection computes weights for bases

- It is possible to approximate a function $f(\boldsymbol{x})$ by a linear combination of simpler functions

$$F(\boldsymbol{x}) = \sum_j w_j \varphi_j(\boldsymbol{x})$$

- If w_j's can be chosen so that approximation error is arbitrarily small for any function $f(\boldsymbol{x})$ over the domain of interest, then $\{\varphi_j\}$ has the property of universal approximation, or $\{\varphi_j\}$ is complete

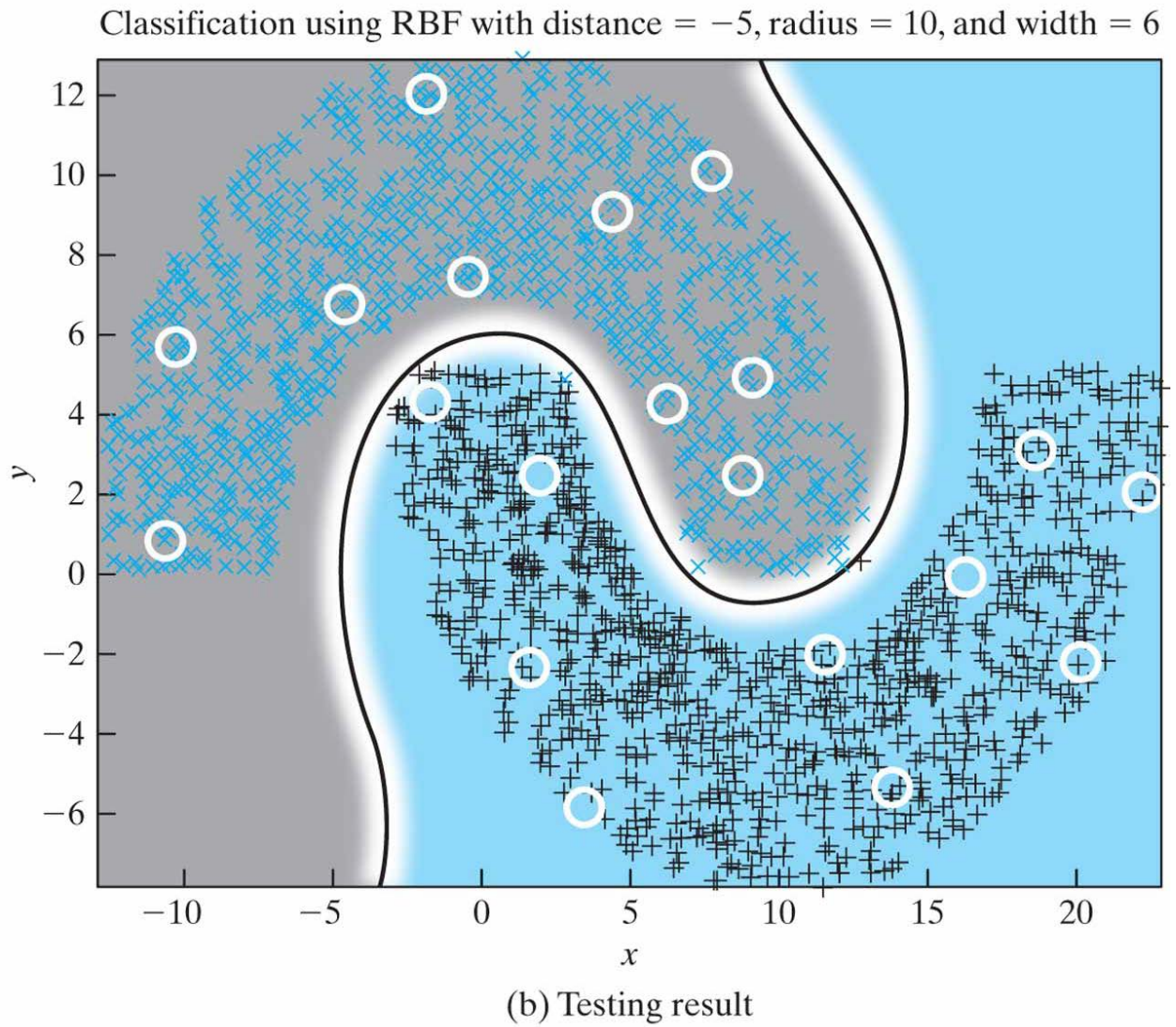
Radial basis function networks are similar to MLPs in structure



RBF nets are trained in three steps

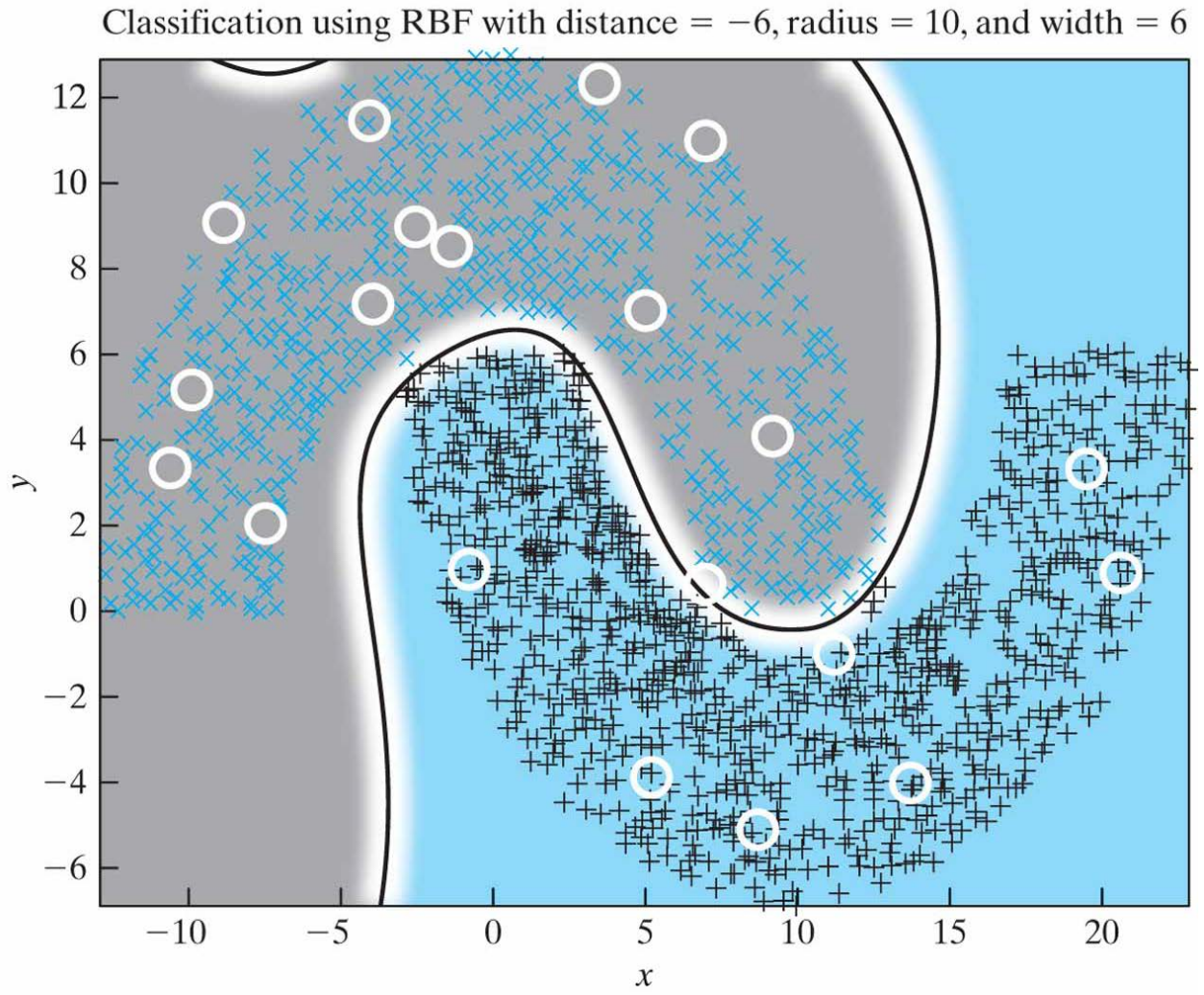
- To train
  1. Choose the Gaussian centers using *K*-means, etc.
  2. Determine the Gaussian widths as the variance of each cluster, or using $d_{\max}$
  3. Determine the weights w_j using linear regression
- Select the number of bases using (cross-)validation

RBF learns double-moon, d = -5



(b) Testing result

RBF net learns double-moon, d = -6



(b) Testing result