Artificial Intelligence DT8012

Statistical learning methods

Chapter 20, AIMA 2nd ed.

Chapter 18, AIMA 3rd ed.

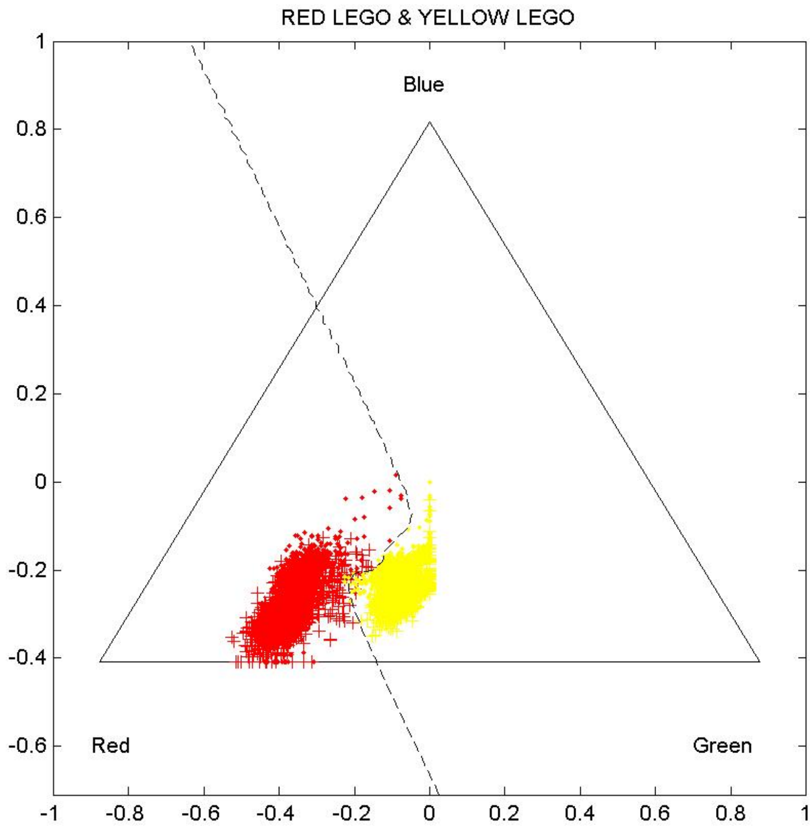
(only ANNs & SVMs)

Standard machine learning strategies

- – Supervised learning (labels for all examples)
- – Semi-supervised learning (labels for some examples)
- – Unsupervised learning (no labels)

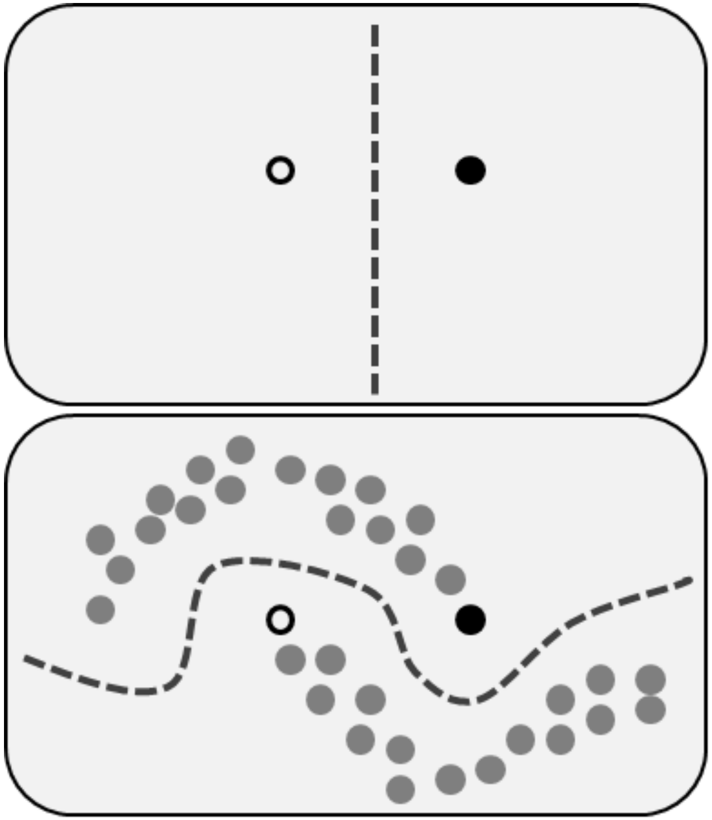
Standard machine learning strategies

- Supervised learning (labels for all examples)
- Semi-supervised learning (labels for some examples)
- Unsupervised learning (no labels)



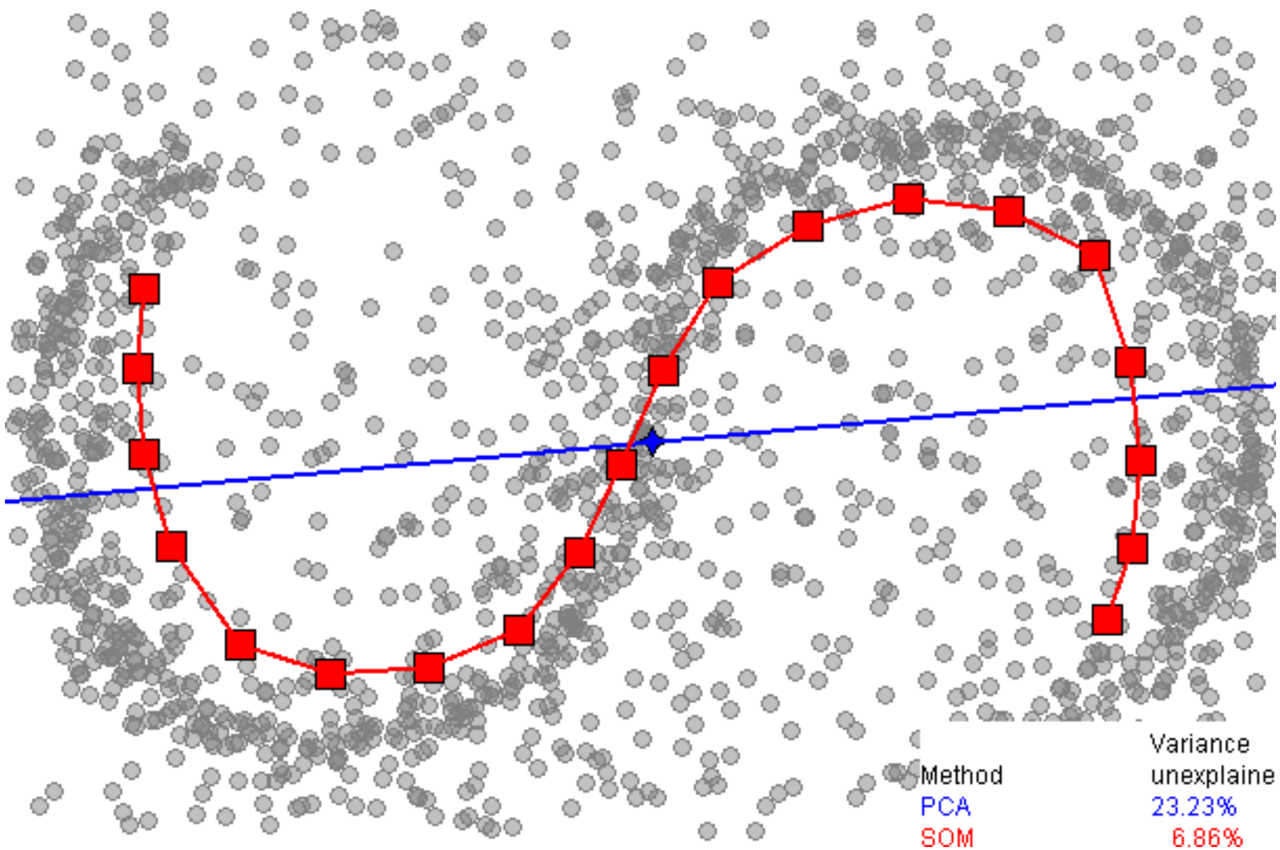
Standard machine learning strategies

- Supervised learning (labels for all examples)
- Semi-supervised learning (labels for some examples)
- Unsupervised learning (no labels)

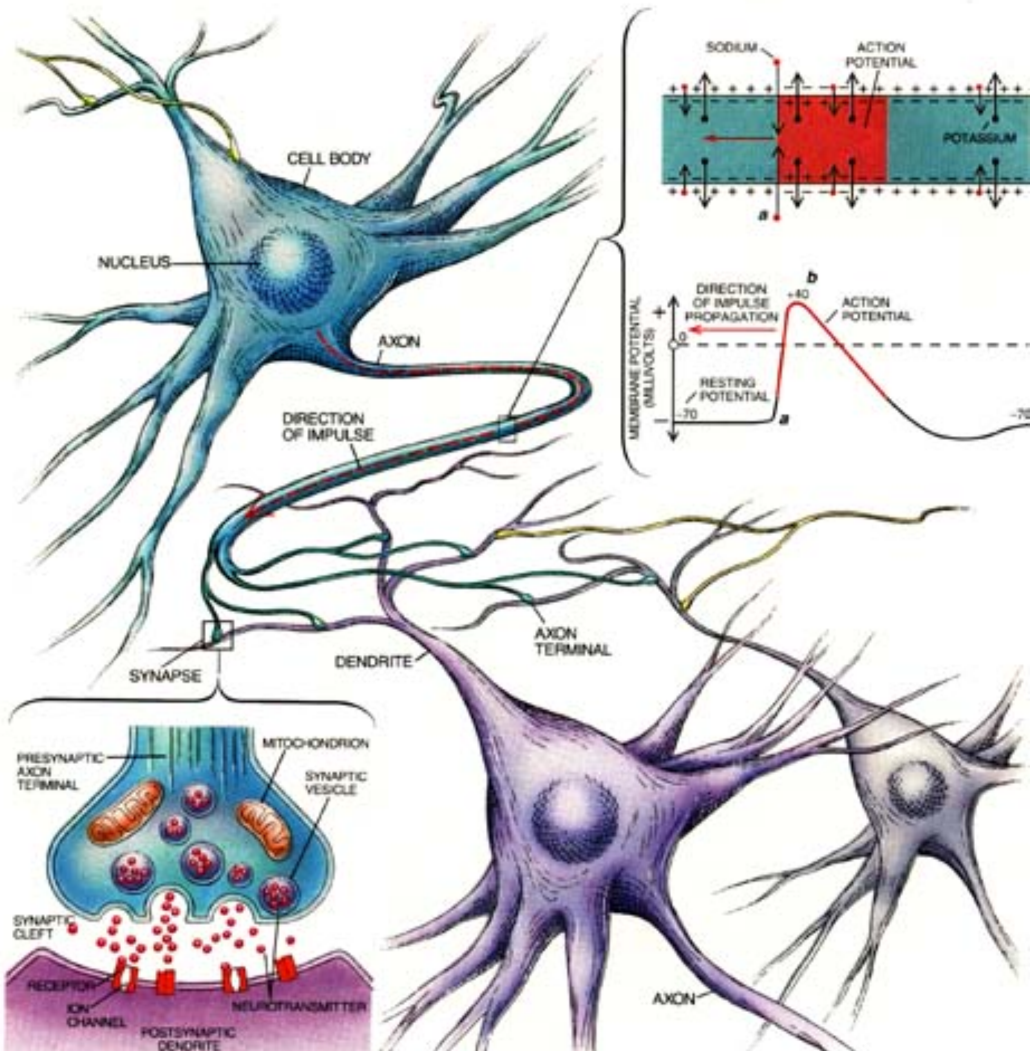


Standard machine learning strategies

- Supervised learning (labels for all examples)
- Semi-supervised learning (labels for some examples)
- Unsupervised learning (no labels)



Artificial neural networks



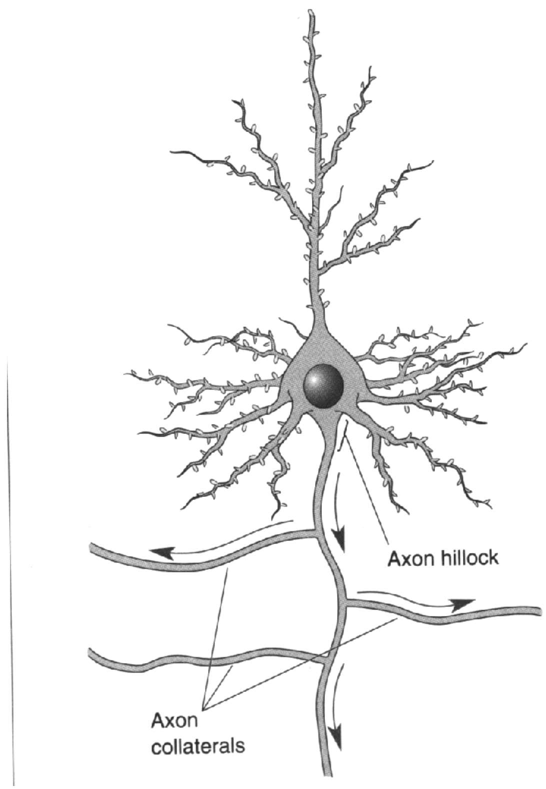
The brain is a pretty intelligent system.

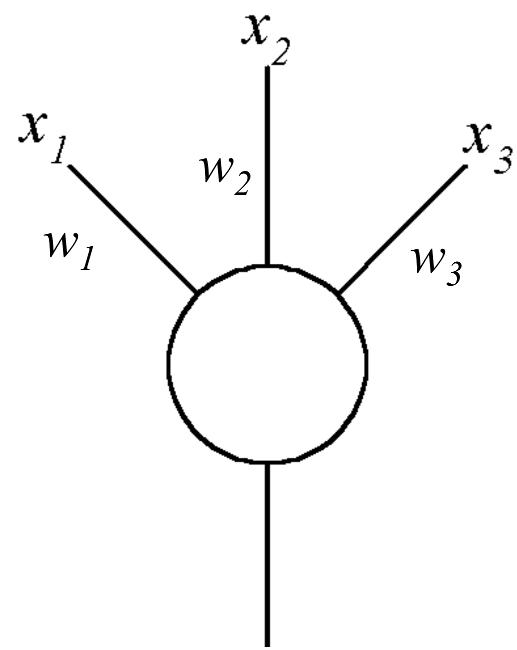
Can we ”copy” it?

There are approx. 10^{11} neurons in the human brain. Elephant brains have twice as many.

The simple model

- The McCulloch-Pitts model (1943)

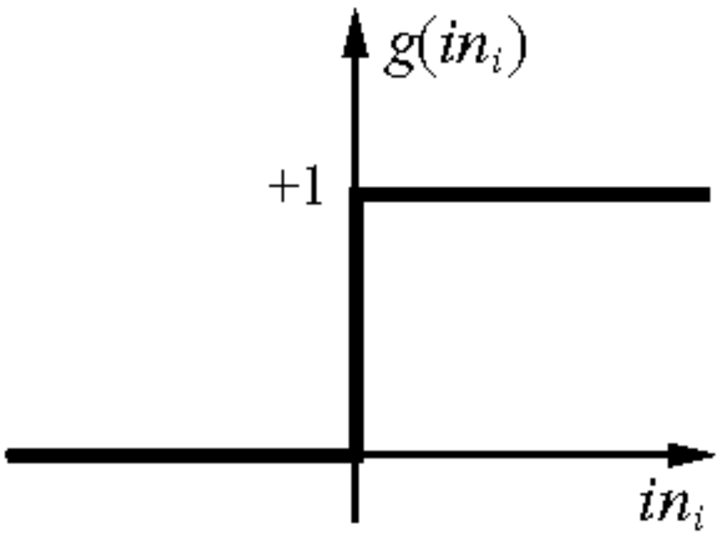




$$y = g(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3)$$

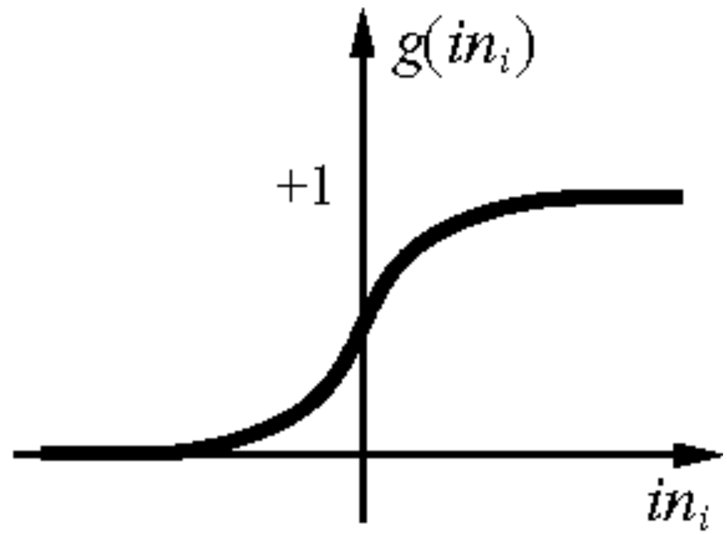
Image from
Neuroscience: Exploring the brain
by Bear, Connors, and Paradiso

Transfer functions *g(z)*



(a)

The Heaviside function



(b)

The logistic function

The simple perceptron

With {-1,+1} representation

$$y(\mathbf{x}) = \operatorname{sgn}[\mathbf{w}^T\mathbf{x}] = \begin{cases} +1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ -1 & \text{if } \mathbf{w}^T\mathbf{x} < 0 \end{cases}$$

Traditionally (early 60:s) trained with *Perceptron learning.*

$$\mathbf{w}^T\mathbf{x} = w_0 + w_1 x_1 + w_2 x_2 + \cdots$$

Perceptron learning

Desired output $f(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } A \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } B \end{cases}$

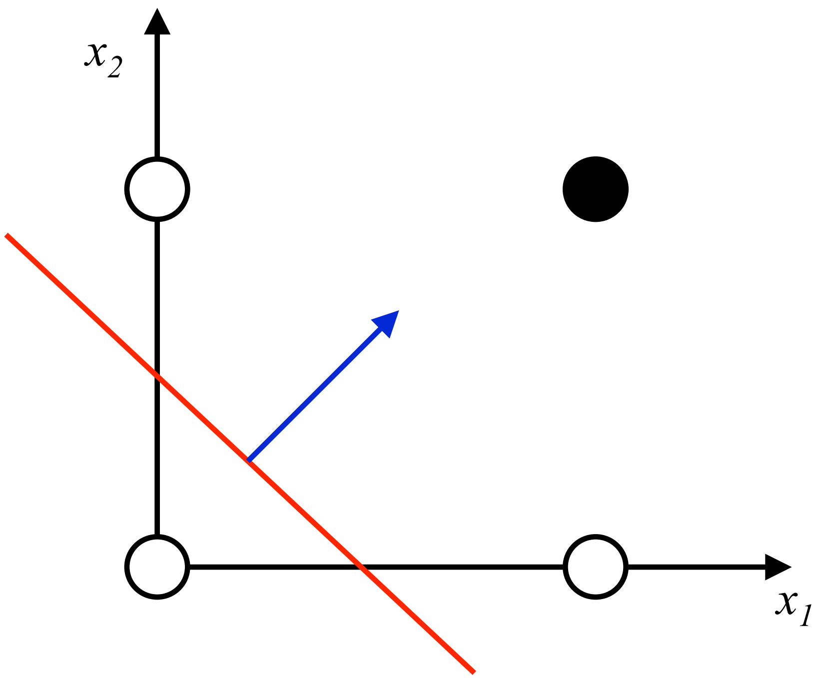
Repeat until no errors are made anymore

1. Pick a random example $[\mathbf{x}(n), f(n)]$
2. If the classification is correct,
   i.e. if $y(\mathbf{x}(n)) = f(n)$, then do nothing
3. If the classification is wrong, then do the following update to the parameters
   (η, the learning rate, is a small positive number)

$$w_i = w_i + \eta f(n) x_i(n)$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



The AND function

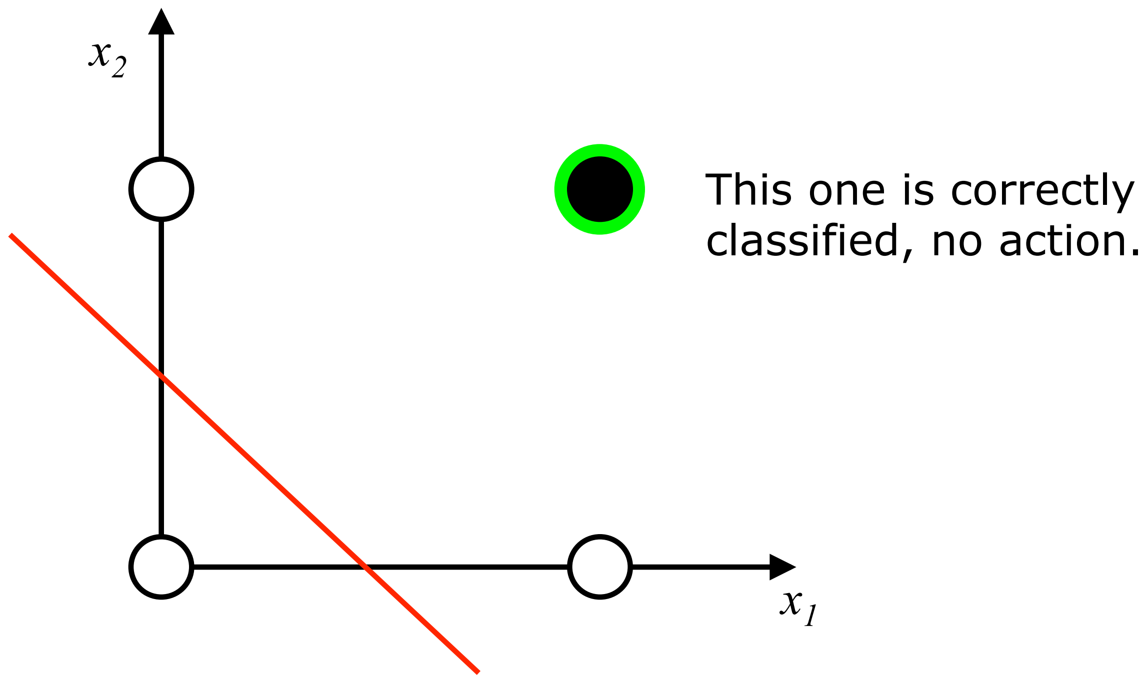
Initial values:

$\eta = 0.3$

$$\mathbf{w} = \begin{pmatrix} -0.5 \\ 1 \\ 1 \end{pmatrix}$$

Example: Perceptron learning

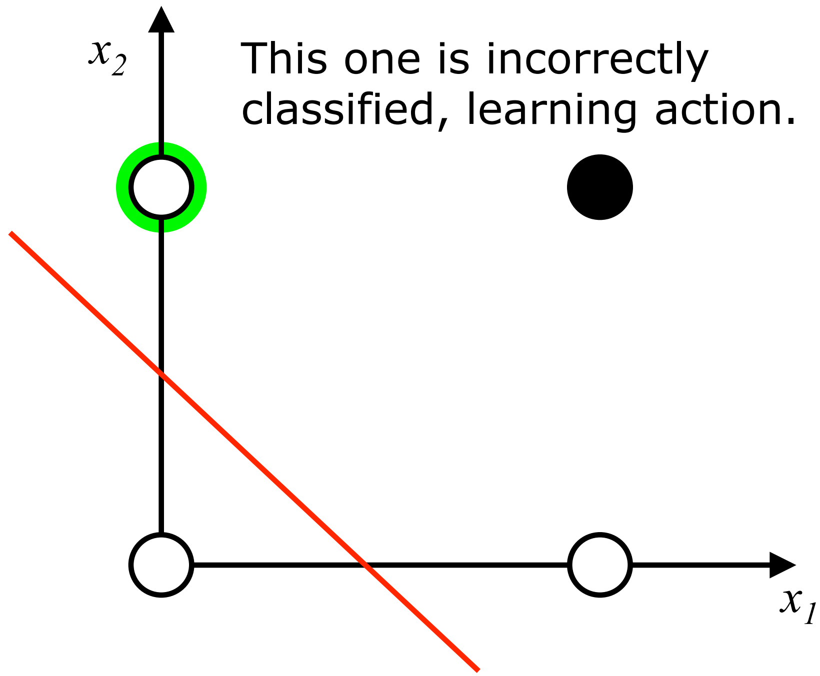
x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



$$\mathbf{w} = \begin{pmatrix} -0.5 \\ 1 \\ 1 \end{pmatrix}$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1

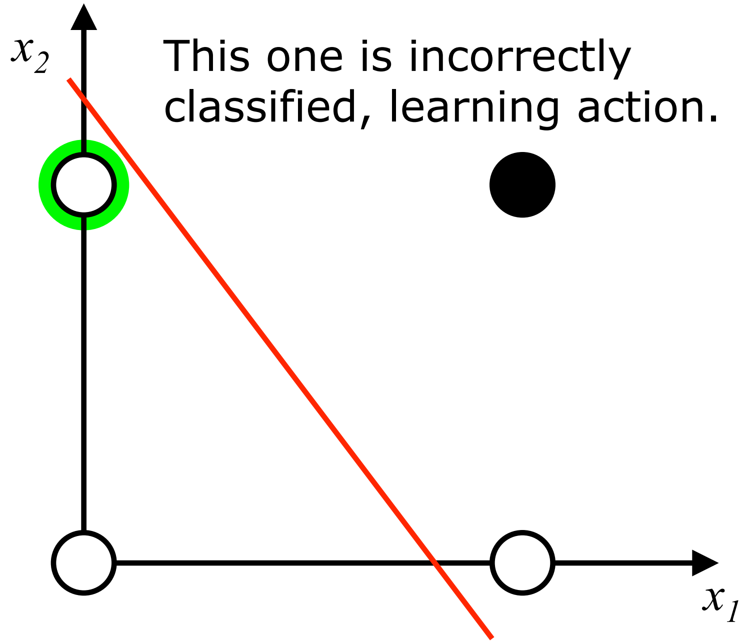


$$\mathbf{w} = \begin{pmatrix} -0.5 \\ 1 \\ 1 \end{pmatrix}$$

$$w_0 = w_0 - \eta \cdot 1 = -0.8$$
$$w_1 = w_1 - \eta \cdot 0 = +1$$
$$w_2 = w_2 - \eta \cdot 1 = 0.7$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



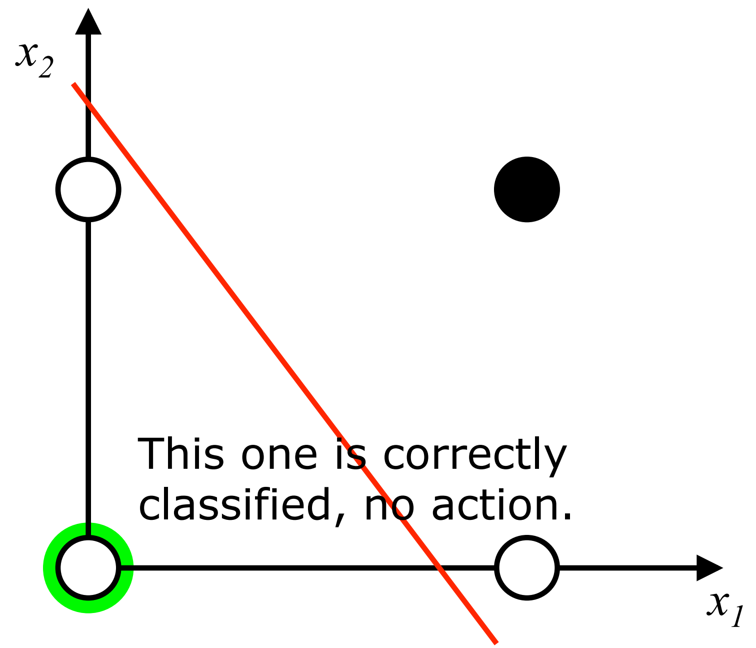
The AND function

$$\mathbf{w} = \begin{pmatrix} -0.8 \\ 1 \\ 0.7 \end{pmatrix}$$

$$w_0 = w_0 - \eta \cdot 1 = -0.8$$
$$w_1 = w_1 - \eta \cdot 0 = +1$$
$$w_2 = w_2 - \eta \cdot 1 = 0.7$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1

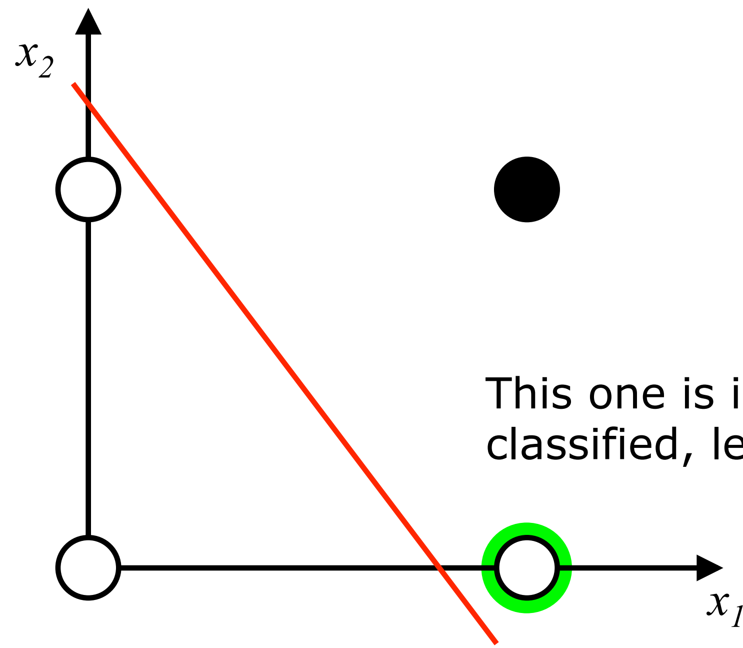


The AND function

$$\mathbf{w} = \begin{pmatrix} -0.8 \\ 1 \\ 0.7 \end{pmatrix}$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



This one is incorrectly classified, learning action.

The AND function

$$\mathbf{w} = \begin{pmatrix} -0.8 \\ 1 \\ 0.7 \end{pmatrix}$$

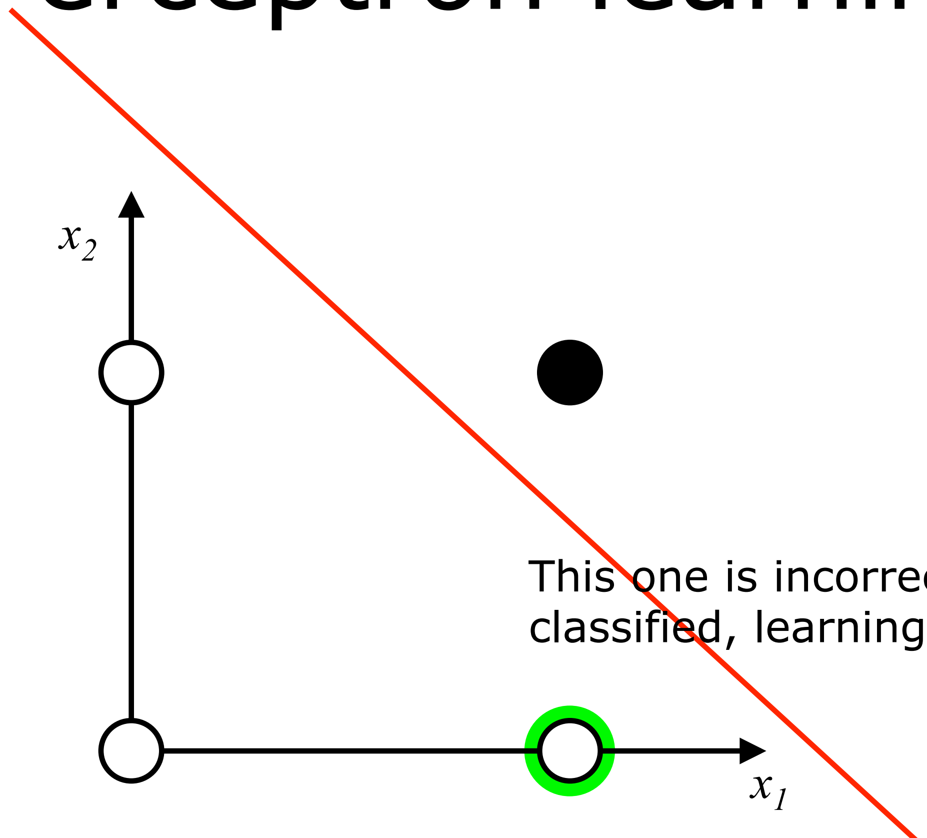
$$w_0 = w_0 - \eta \cdot 1 = -1.1$$

$$w_1 = w_1 - \eta \cdot 1 = 0.7$$

$$w_2 = w_2 - \eta \cdot 0 = 0.7$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



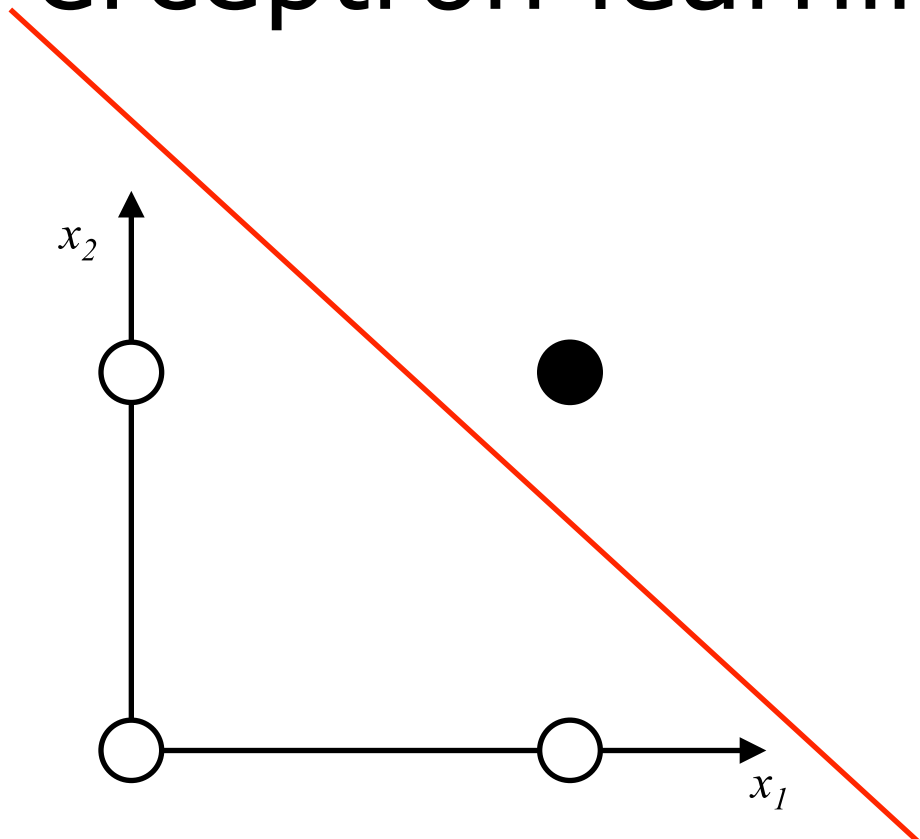
The AND function

$$\mathbf{w} = \begin{pmatrix} -1.1 \\ 0.7 \\ 0.7 \end{pmatrix}$$

$$w_0 = w_0 - \eta \cdot 1 = -1.1$$
$$w_1 = w_1 - \eta \cdot 1 = 0.7$$
$$w_2 = w_2 - \eta \cdot 0 = 0.7$$

Example: Perceptron learning

x_1	x_2	f
0	0	-1
0	1	-1
1	0	-1
1	1	+1



The AND function

$$\mathbf{w} = \begin{pmatrix} -1.1 \\ 0.7 \\ 0.7 \end{pmatrix}$$

Final solution

Perceptron learning

- Perceptron learning is guaranteed to find a solution in finite time, if a solution exists.
- Perceptron learning cannot be generalized to more complex networks.

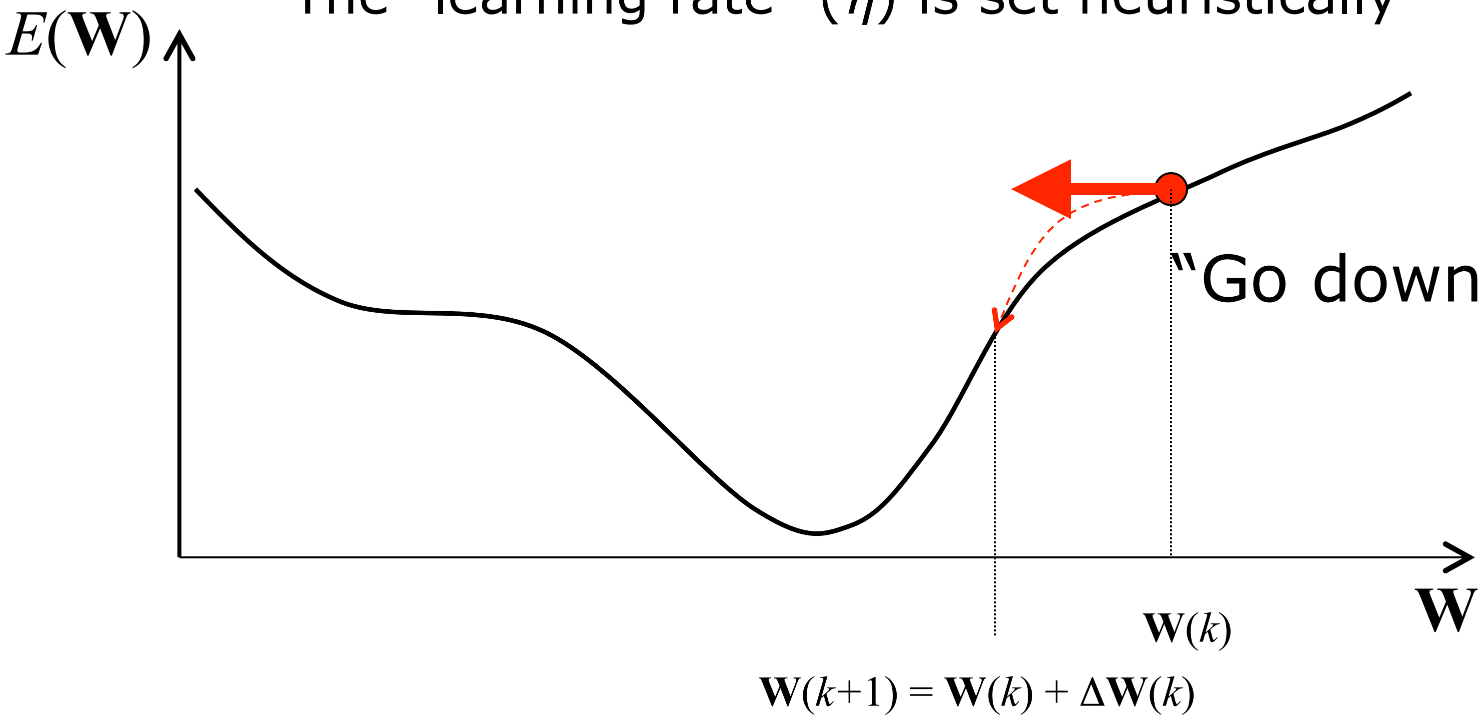
- Better to use gradient descent – based on formulating an error and differentiable functions

$$E(\mathbf{W}) = \sum_{n=1}^{N} \left[f(n) - y(\mathbf{W}, n) \right]^2$$

Gradient search

$$\Delta \mathbf{W} = -\eta \nabla_W E(\mathbf{W})$$

The "learning rate" (η) is set heuristically



The Multilayer Perceptron (MLP)

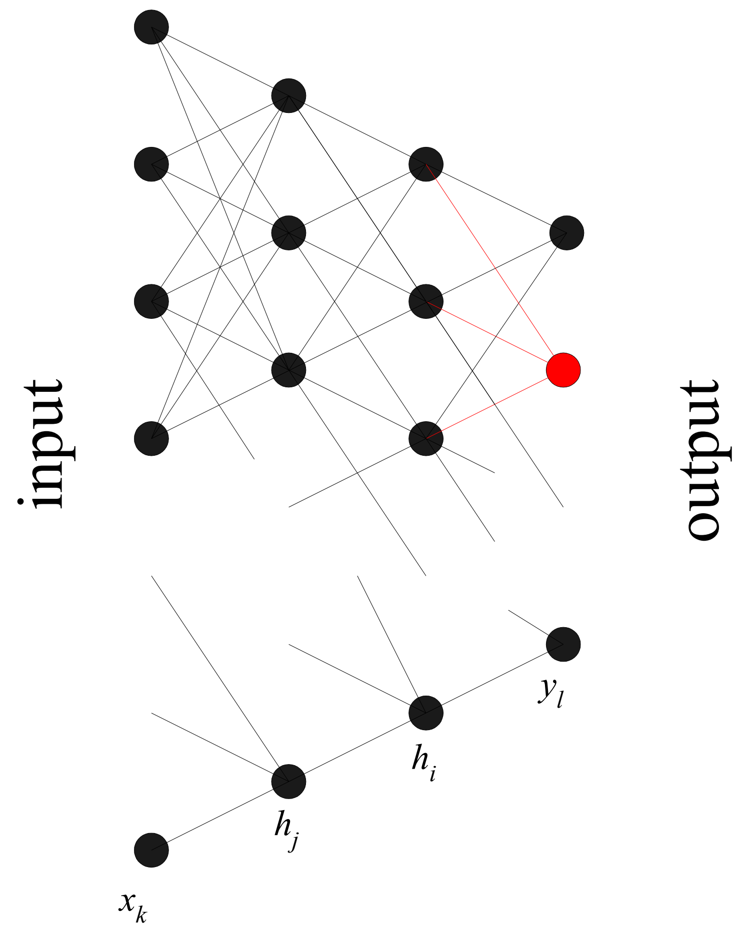
- Combine several single layer perceptrons.
- Each single layer perceptron uses a sigmoid function

E.g.

$$\phi(z) = \tanh(z)$$

$$\phi(z) = \left[1 + \exp(-z)\right]^{-1}$$

Can be trained using gradient descent



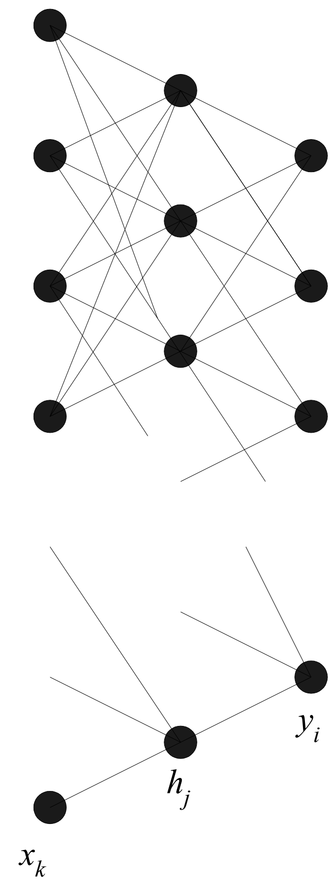
Example: One hidden layer

- Can approximate <u>any</u> continuous function

$$y_i(\mathbf{x}) = \theta\left[v_{i0} + \sum_{j=1}^{J} v_{ij} h_j(\mathbf{x})\right]$$

$$h_j(\mathbf{x}) = \phi\left[w_{j0} + \sum_{k=1}^{D} w_{jk} x_k\right]$$

$\theta(z)$ = sigmoid or linear,
$\phi(z)$ = sigmoid.



Example of computing the gradient

$$\Delta W = -\eta \nabla_W E(W)$$

$$E(W) = MSE = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}(W, x(n)) - y(n))^2 = \frac{1}{N}\sum_{n=1}^{N} e^2$$

$$\nabla_W E(W) = \nabla_W \left(\frac{1}{N}\sum_{n=1}^{N} e^2(n) \right) = \frac{2}{N}\sum_{n=1}^{N} e(n)(\nabla_W e(n)) = \frac{2}{N}\sum_{n=1}^{N} e(n)(\nabla_W \hat{y})$$

What we need to do is to compute $\nabla_W \hat{y}$

Equation for a single output, one hidden layer network:

$$\hat{y} = \theta(v_0 + \sum_{j=1}^{J} v_j h_j(w_{j0} + \sum_{k=1}^{K} x_k w_{jk}))$$

Gradient descent (Backpropagation)

$$\Delta W = -\eta \nabla_W E(W)$$

RPROP (Resilient PROPagation)

Parameter update rule:

$$\Delta W_i = -\eta_i(t) sign(\nabla_{W_i} E(W_i))$$

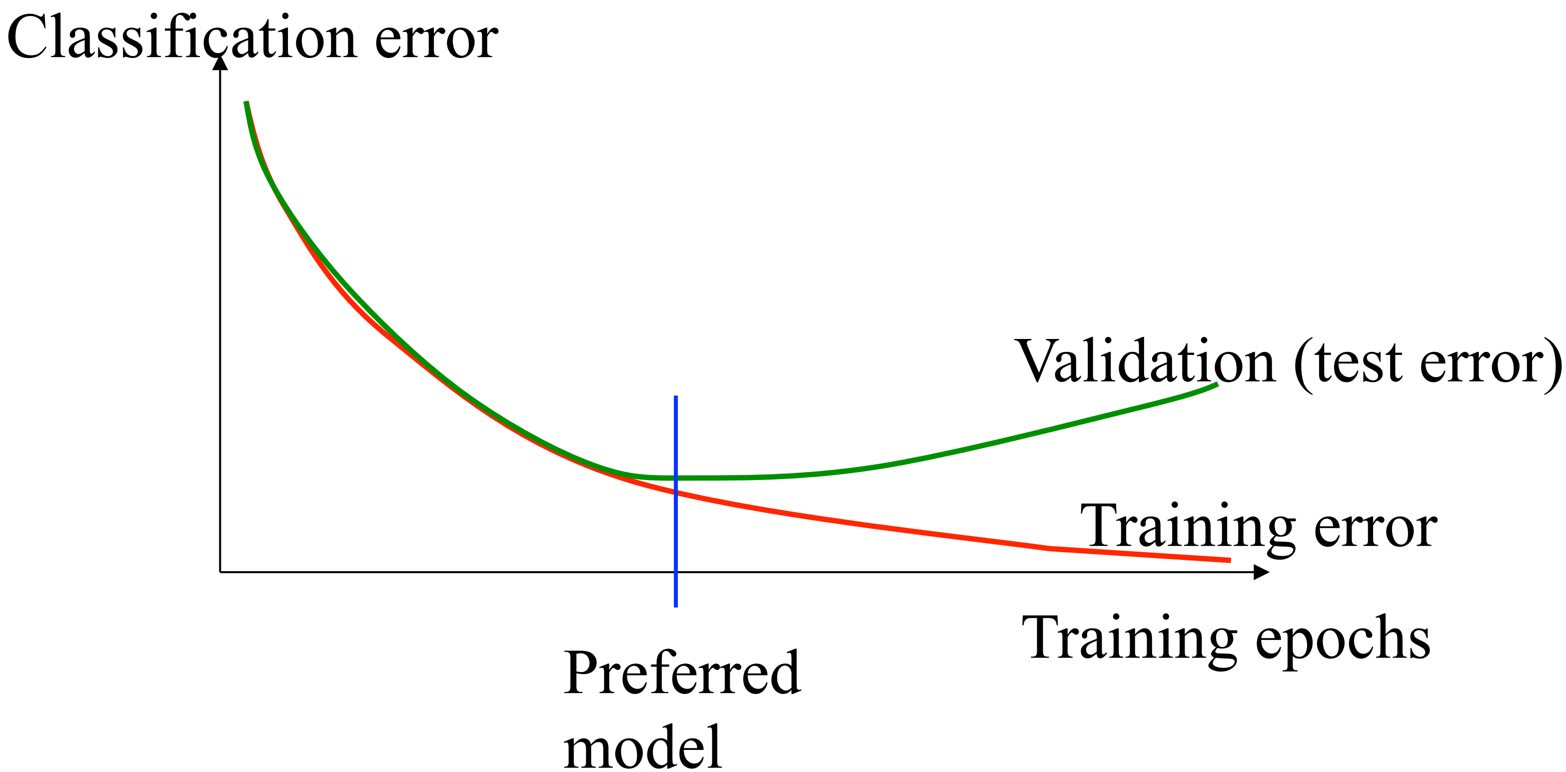
Learning rate update rule:

$$\eta_i(t) = \begin{cases} 1.2\eta_i(t-1) & if \quad \nabla_{W_i} E_t(W_i) \cdot \nabla_{W_i} E_{t-1}(W_i) > 0 \\ 0.5\eta_i(t-1) & if \quad \nabla_{W_i} E_t(W_i) \cdot \nabla_{W_i} E_{t-1}(W_i) < 0 \end{cases}$$

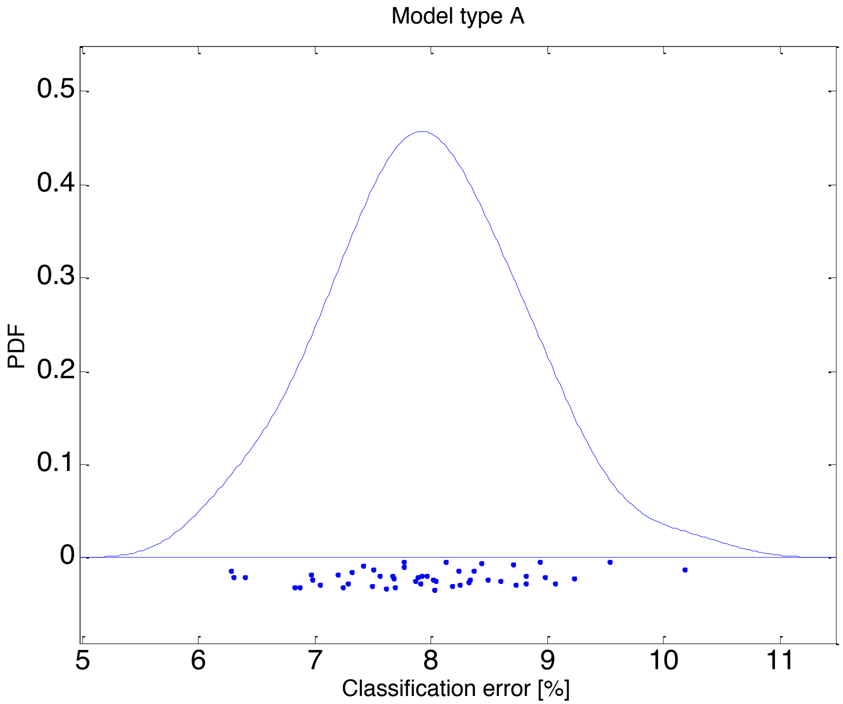
No parameter tuning unlike standard backpropagation!

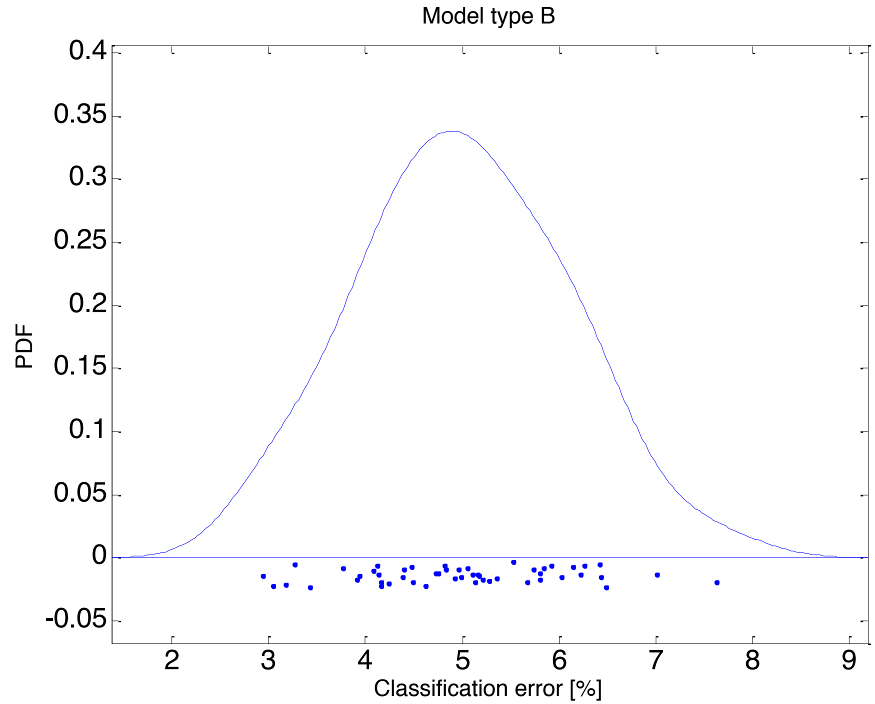
When should you stop learning?

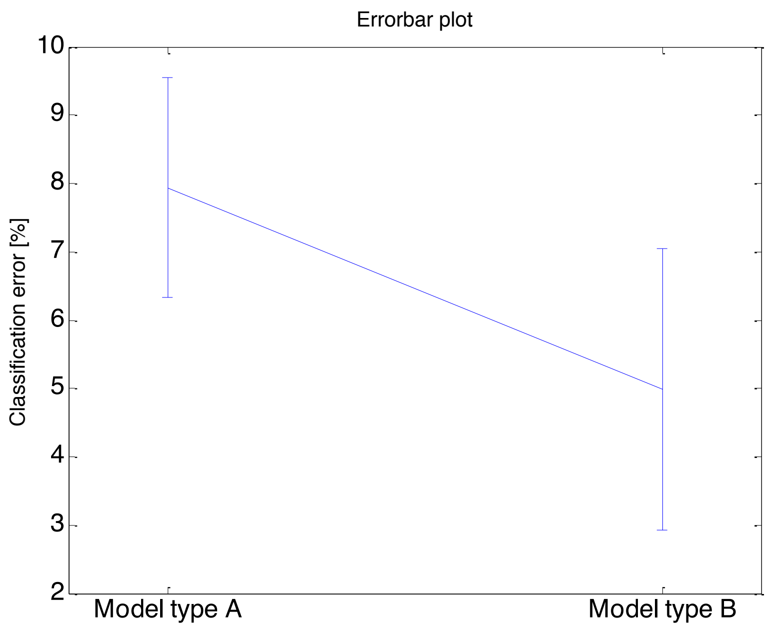
- After a set number of learning epochs
- When the change in the gradient becomes smaller than a certain number
- Validation data - “early stopping”



Model selection







<u>Can use to determine:</u>

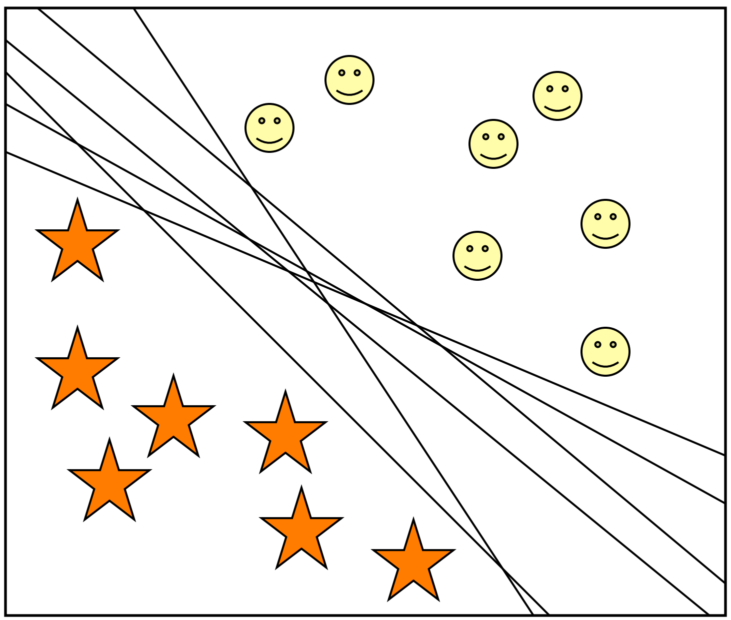
- Number of hidden nodes
- Which input signals to use
- If a pre-processing strategy is good or not
- Etc...

<u>Variability typically induced by:</u>

- Varying training and test data sets
- Random initial model parameters

Support vector machines

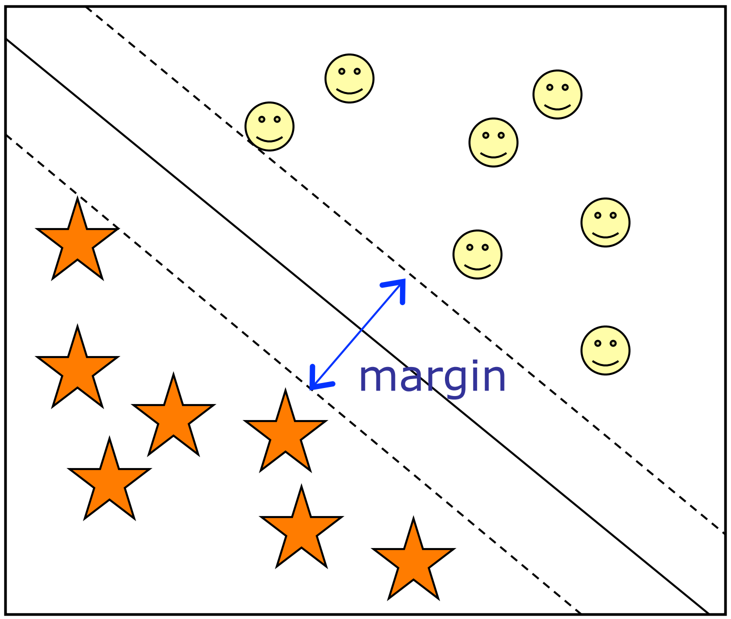
Linear classifier on a linearly separable problem



There are infinitely many lines that have zero training error.

Which line should we choose?

Linear classifier on a linearly separable problem



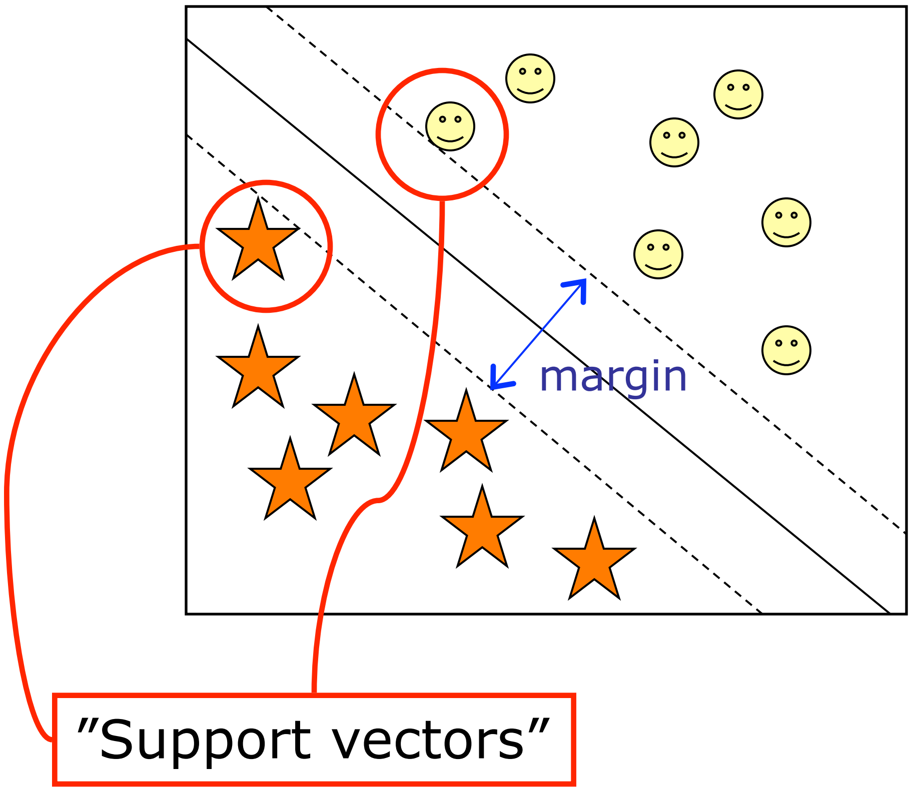
There are infinitely many lines that have zero training error.

Which line should we choose?

⇒ Choose the line with the largest margin.

The “large margin classifier”

Linear classifier on a linearly separable problem



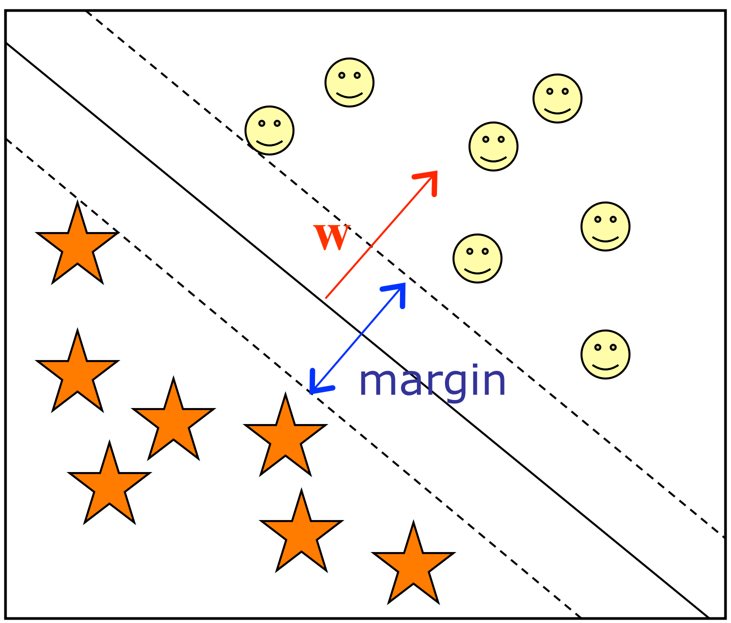
There are infinitely many lines that have zero training error.

Which line should we choose?

⇒ Choose the line with the <u>largest margin</u>.

The "large margin classifier"

Computing the margin



The plane separating  and  is defined by

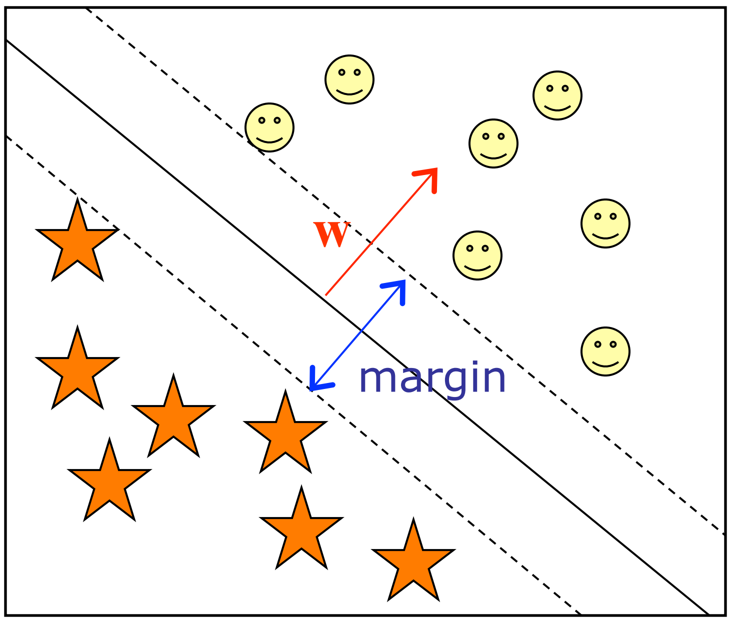
$$\mathbf{w}^T \mathbf{x} = a$$

The dashed planes are given by

$$\mathbf{w}^T \mathbf{x} = a + b$$

$$\mathbf{w}^T \mathbf{x} = a - b$$

Computing the margin



Divide by b

$$\mathbf{w}^T\mathbf{x}/b = a/b + 1$$

$$\mathbf{w}^T\mathbf{x}/b = a/b - 1$$

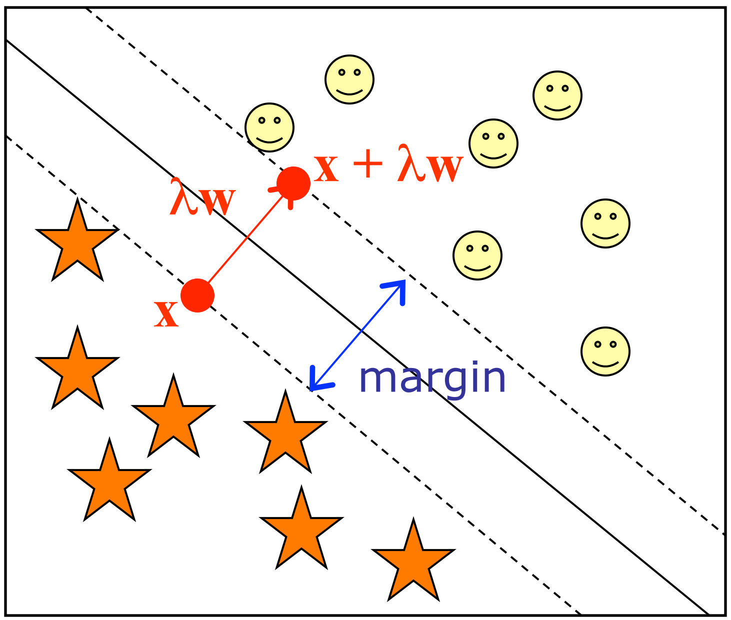
Define new $\mathbf{w} = \mathbf{w}/b$ and $\alpha = a/b$

$$\mathbf{w}^T\mathbf{x} = \alpha + 1$$

$$\mathbf{w}^T\mathbf{x} = \alpha - 1$$

We have defined a scale for $\mathbf{w}$ and b

Computing the margin



We have

$$\mathbf{w}^T\mathbf{x} = \alpha - 1$$

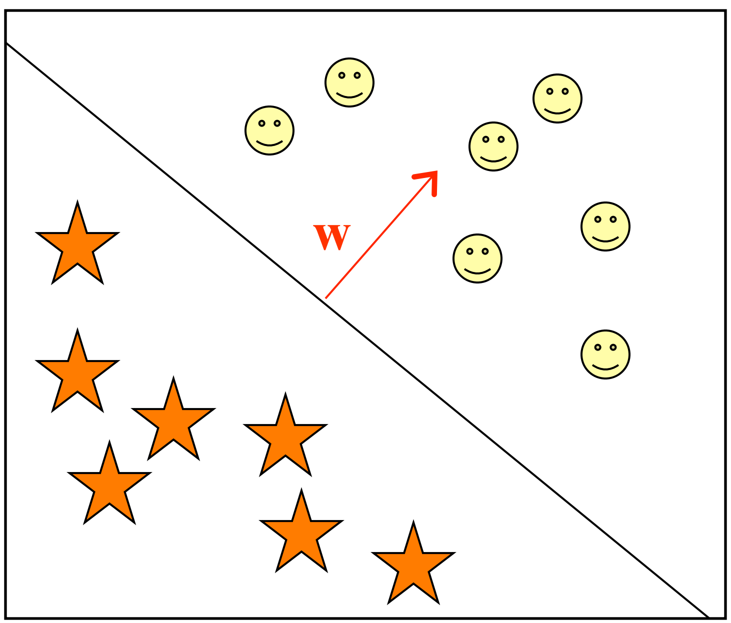
$$\mathbf{w}^T(\mathbf{x} + \lambda\mathbf{w}) = \alpha + 1$$

$$\|\lambda\mathbf{w}\| = \text{margin}$$

which gives

$$\text{margin} = \frac{2}{\|\mathbf{w}\|}$$

Linear classifier on a linearly separable problem

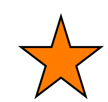


Maximizing the margin is equal to minimizing

$$||\mathbf{w}||$$

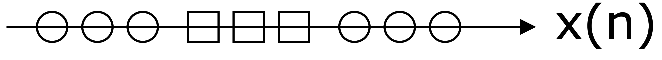
subject to the constraints

$\mathbf{w}^{\mathrm{T}}\mathbf{x}(n) - \alpha \geq +1$ for all 

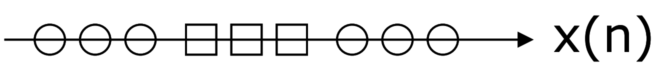
$\mathbf{w}^{\mathrm{T}}\mathbf{x}(n) - \alpha \leq -1$ for all 

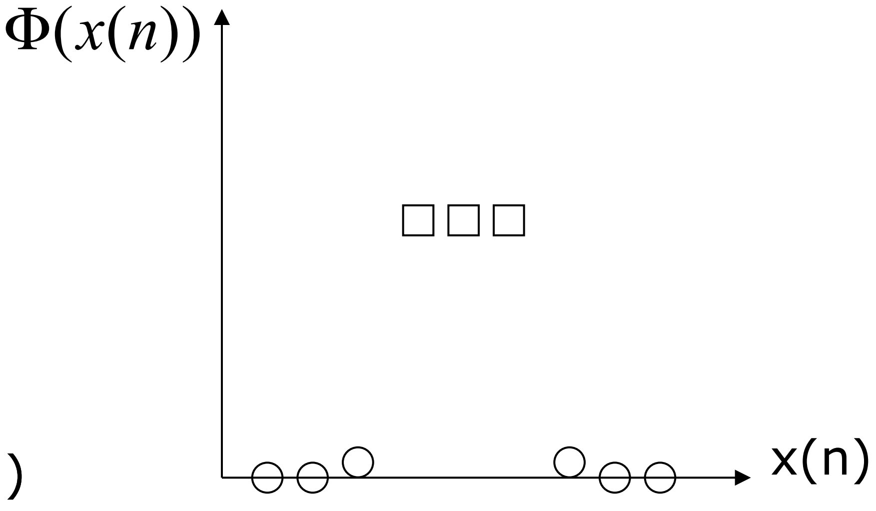
Quadratic programming problem,
constraints can be included with Lagrange multipliers.

How to deal with nonlinear case?

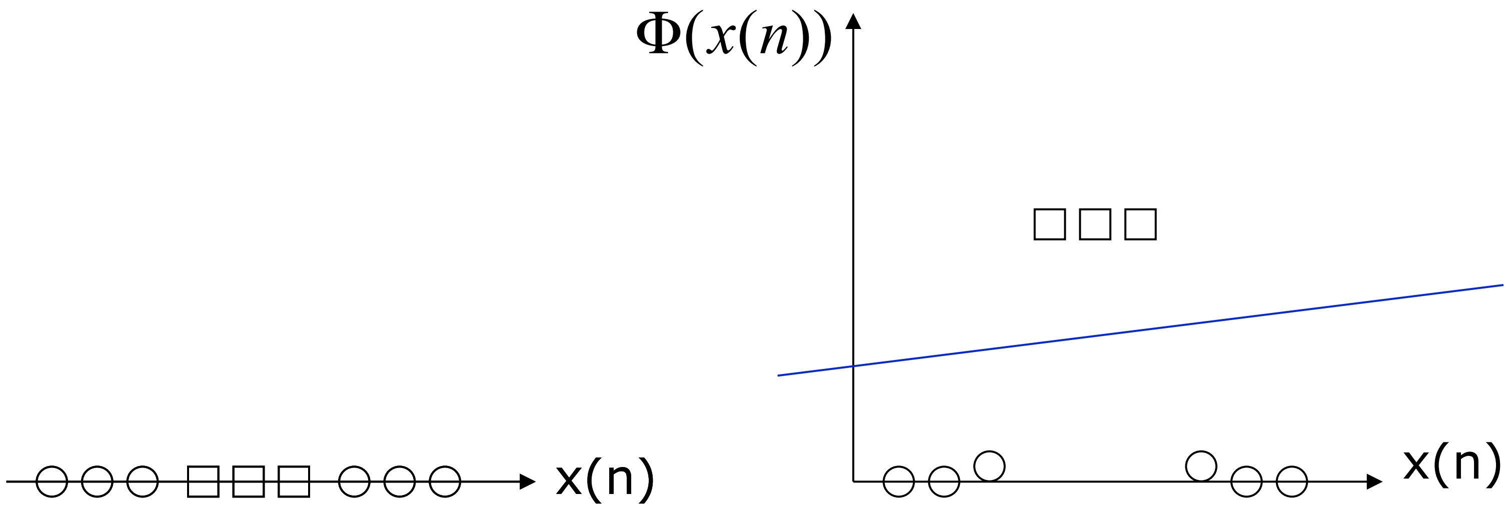


How to deal with nonlinear case?





How to deal with nonlinear case?



Scalar product kernel trick

If we can find kernel such that

$$K(\mathbf{x}(n), \mathbf{x}(m)) = \boldsymbol{\varphi}(\mathbf{x}(n))^T \boldsymbol{\varphi}(\mathbf{x}(m))$$

Then we don't even have to know the mapping
to solve the problem...

Kernel trick – computation example

$$K(x,z) = (x^T z)^2 = (\sum_{i=1}^{N} x_i z_i)(\sum_{j=1}^{N} x_j z_j) = \sum_{i=1}^{N}\sum_{j=1}^{N}(x_i x_j)(z_i z_j) = \varphi(x)^T \varphi(z)$$

Kernel trick – computation example

$$K(x,z) = (x^T z)^2 = (\sum_{i=1}^{N} x_i z_i)(\sum_{j=1}^{N} x_j z_j) = \sum_{i=1}^{N}\sum_{j=1}^{N}(x_i x_j)(z_i z_j) = \varphi(x)^T \varphi(z)$$

For N=3

$$\varphi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ \vdots \\ x_3 x_3 \end{bmatrix}$$

Need O(N^2) to compute $\varphi(x)$

Kernel trick – computation example

$$K(x,z) = (x^T z)^2 = (\sum_{i=1}^{N} x_i z_i)(\sum_{j=1}^{N} x_j z_j) = \sum_{i=1}^{N} \sum_{j=1}^{N} (x_i x_j)(z_i z_j) = \varphi(x)^T \varphi(z)$$

For N=3

$$\varphi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ \vdots \\ x_3 x_3 \end{bmatrix}$$

Need $O(N^2)$ to compute $\varphi(x)$

Need only O(N) to compute K(x,z)

Valid kernels (Mercer's theorem)

Define the matrix

$$\mathbf{K} = \begin{pmatrix} K[\mathbf{x}(1),\mathbf{x}(1)] & K[\mathbf{x}(1),\mathbf{x}(2)] & \cdots & K[\mathbf{x}(1),\mathbf{x}(N)] \\ K[\mathbf{x}(2),\mathbf{x}(1)] & K[\mathbf{x}(2),\mathbf{x}(2)] & \cdots & K[\mathbf{x}(2),\mathbf{x}(N)] \\ \vdots & \vdots & \ddots & \vdots \\ K[\mathbf{x}(N),\mathbf{x}(1)] & K[\mathbf{x}(N),\mathbf{x}(2)] & \cdots & K[\mathbf{x}(N),\mathbf{x}(N)] \end{pmatrix}$$

If $\mathbf{K}$ is symmetric, $\mathbf{K} = \mathbf{K}^T$, and positive semi-definite, then $K[\mathbf{x}(i),\mathbf{x}(j)]$ is a valid kernel.

Examples of kernels

$$K[\mathbf{x}(i), \mathbf{x}(j)] = \exp\left[- \left\| \mathbf{x}(i) - \mathbf{x}(j) \right\|^2 / 2\sigma \right]$$

$$K[\mathbf{x}(i), \mathbf{x}(j)] = \left[\mathbf{x}(i)^T \mathbf{x}(j) \right]^d$$

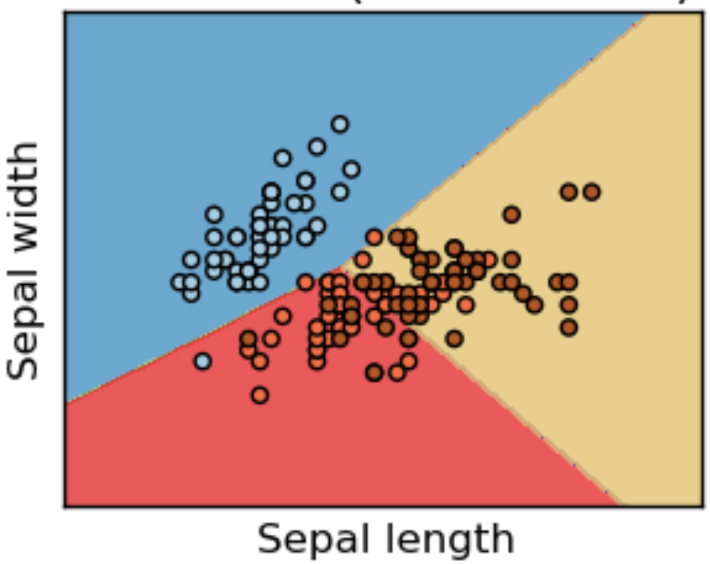
First, Gaussian kernel.

Second, polynomial kernel. With $d=1$ we have linear SVM.

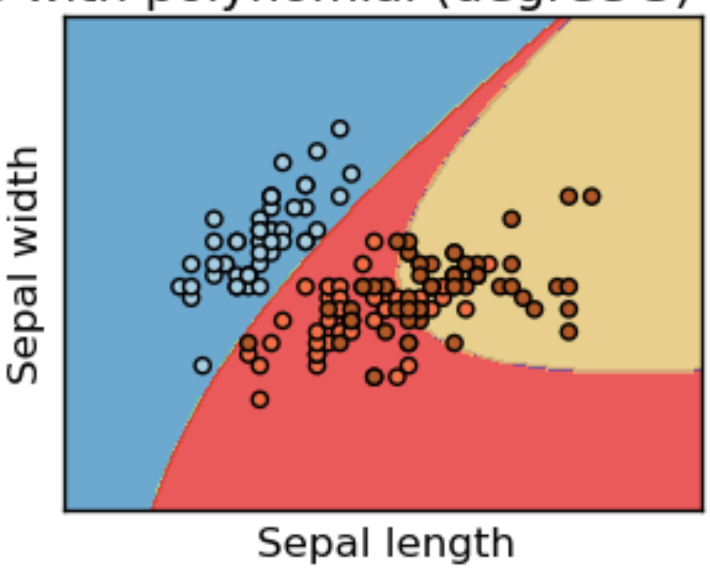
Linear SVM often used with good success on high dimensional data (e.g. text classification).

Practical examples

LinearSVC (linear kernel)

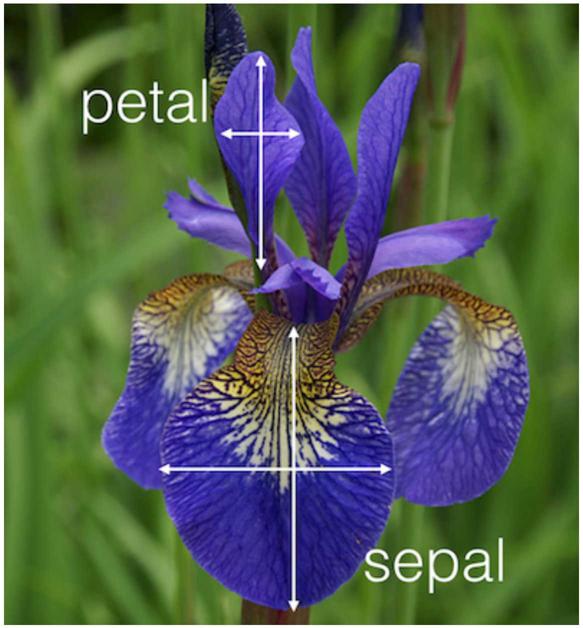


SVC with polynomial (degree 3) kernel



Iris data set

Three variations of a flower from the same ”family” of flowers



Python implementation

Available at

http://scikit-learn.org/stable/

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
y = iris.target

h = .02  # step size in the mesh

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
svc = svm.SVC(kernel='linear', C=C).fit(X, y)
poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# title for the plots
titles = ['LinearSVC (linear kernel)',
'SVC with polynomial (degree 3) kernel']

for i, clf in enumerate((svc, poly_svc)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, m_max]x[y_min, y_max].
    plt.subplot(2, 2, i + 1)
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title(titles[i])
```

Python implementation

iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[: :2]  # we only take the first two features. We could
                 # avoid this ugly slicing by using a two-dim dataset

ep size in the mesh

instance of SVM and fit out data. We do not scale our
ant to plot the support vectors
ularization parameter
l='linear', C=C).fit(X, y)
ernel='poly', degree=3, C=C).fit(X, y)

in
min() - 1, X[:, 0].max() + 1
min() - 1, X[:, 1].max() + 1
.arange(x_min, x_max, h),
(y_min, y_max, h))

inear kernel)',
 (degree 3) kernel']

erate((svc, poly_svc)):
decision boundary. For that, we will assign a color to each
 in the mesh [x_min, m_max]x[y_min, y_max].
subplot(2, 2, i + 1)
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title(titles[i])
```

Python implementation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
y = iris.target

h = .02  # step size in the mesh

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
svc = svm.SVC(kernel='linear', C=C).fit(X, y)
poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)
```

svc = svm.SVC(kernel='linear',C=C).fit(X, y)

poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)

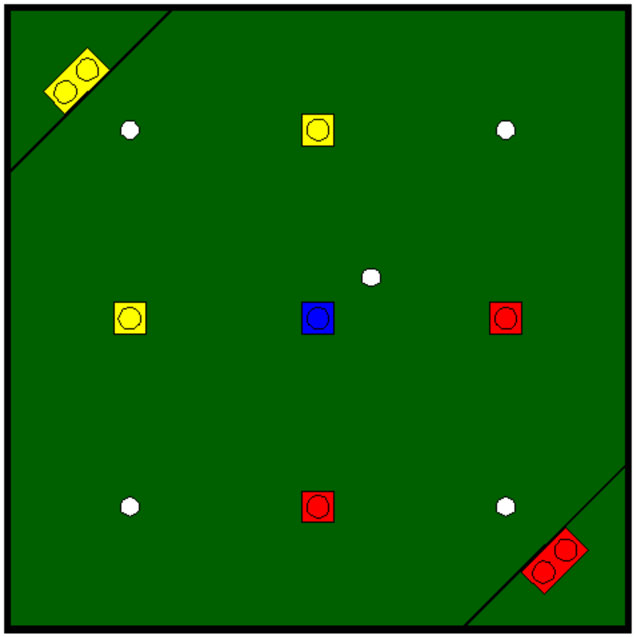
```
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title(titles[i])
```

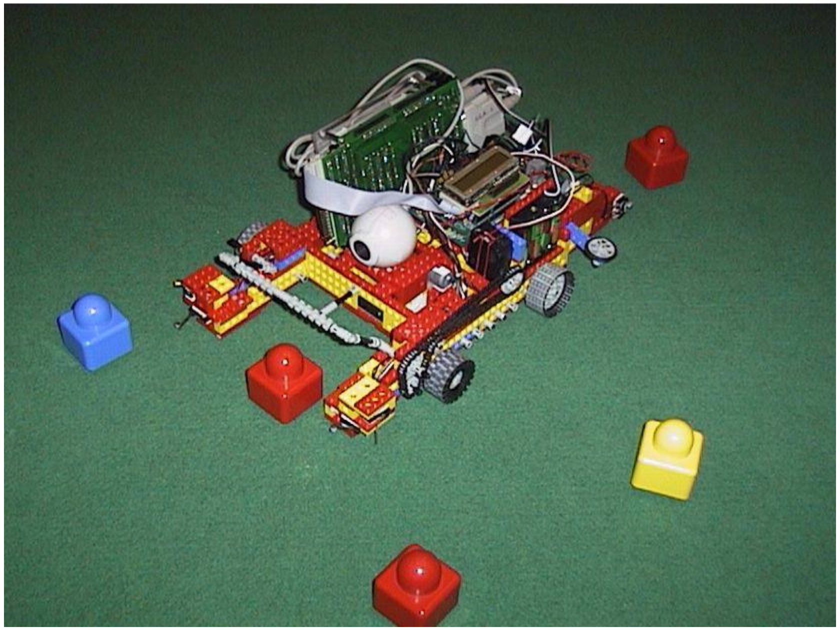
Python implementation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
y = iris.target

h = .02  # step size in the mesh

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
svc = svm.SVC(kernel='linear', C=C).fit(X, y)
poly_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# title for the plots
titles = ['LinearSVC (linear kernel)',
'SVC with polynomial (degree 3) kernel']

for i, clf in enumerate((svc, poly_svc)):
    # Plot the decision boundary. For that, we will assign a color to each
    ... the mesh [x_min, m_max]x[y_min, y_max].
    ... 2, i + 1)
    ... st(wspace=0.4, hspace=0.4)
    ... ()])
```

```
for i, clf in enumerate((svc, poly_svc)):
...
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
```

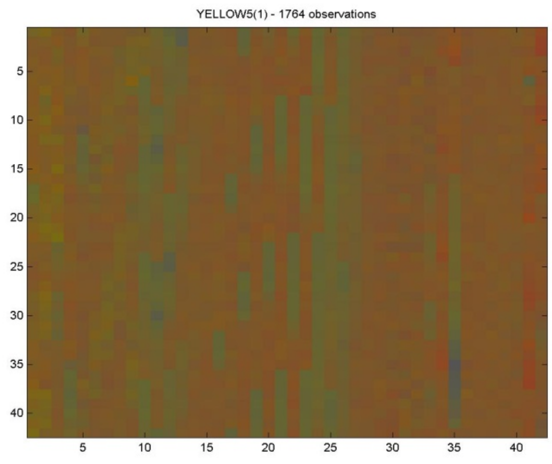
Example: Robot color vision

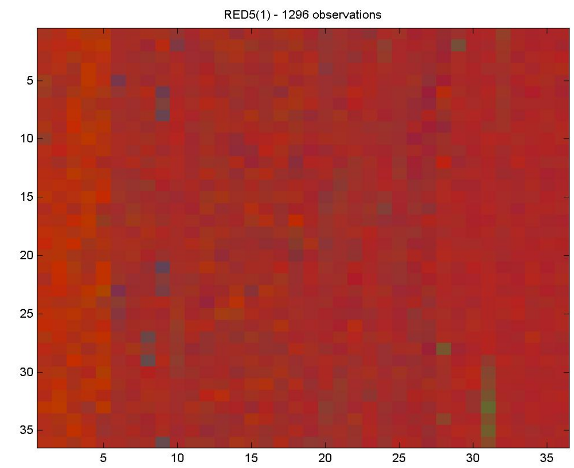




Classify the Lego pieces into *red*, *blue*, and *yellow.*
Classify *white* balls, *black* sideboard, and *green* carpet.

What the camera sees
(RGB space)

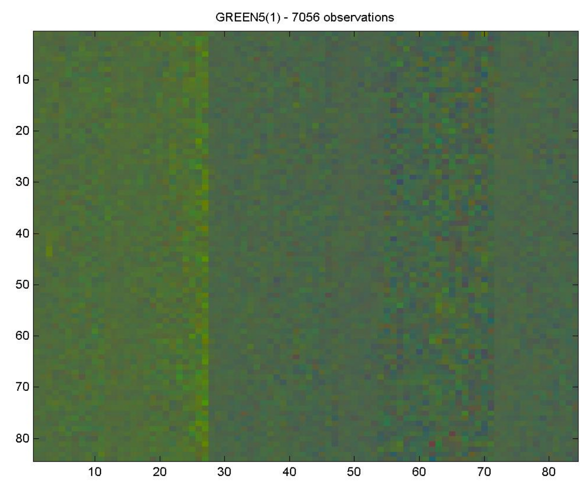




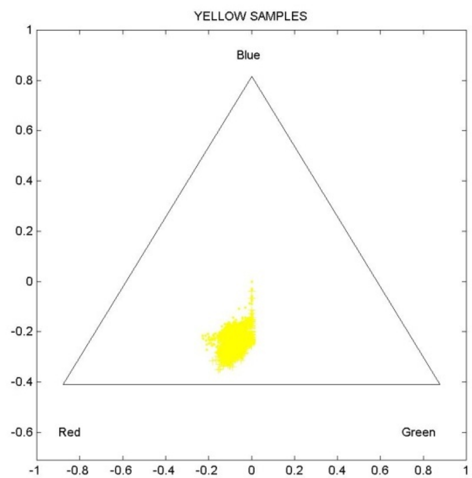
Yellow

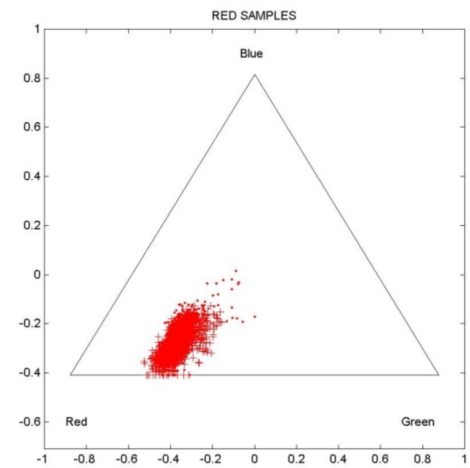
Red

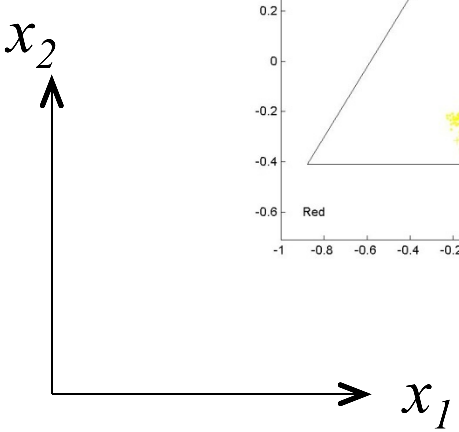
Green



Lego in normalized *rgb* space

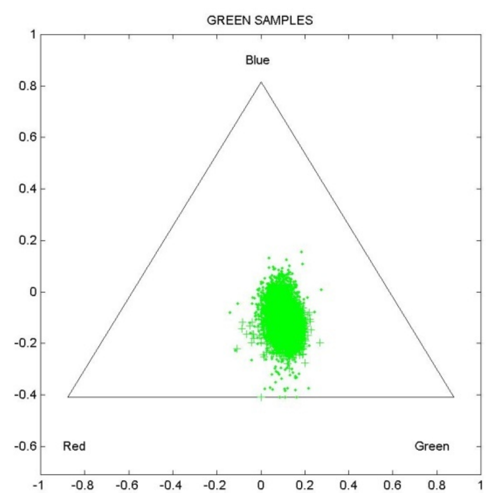






Input is 2D

$$\mathbf{x} \in X^2$$

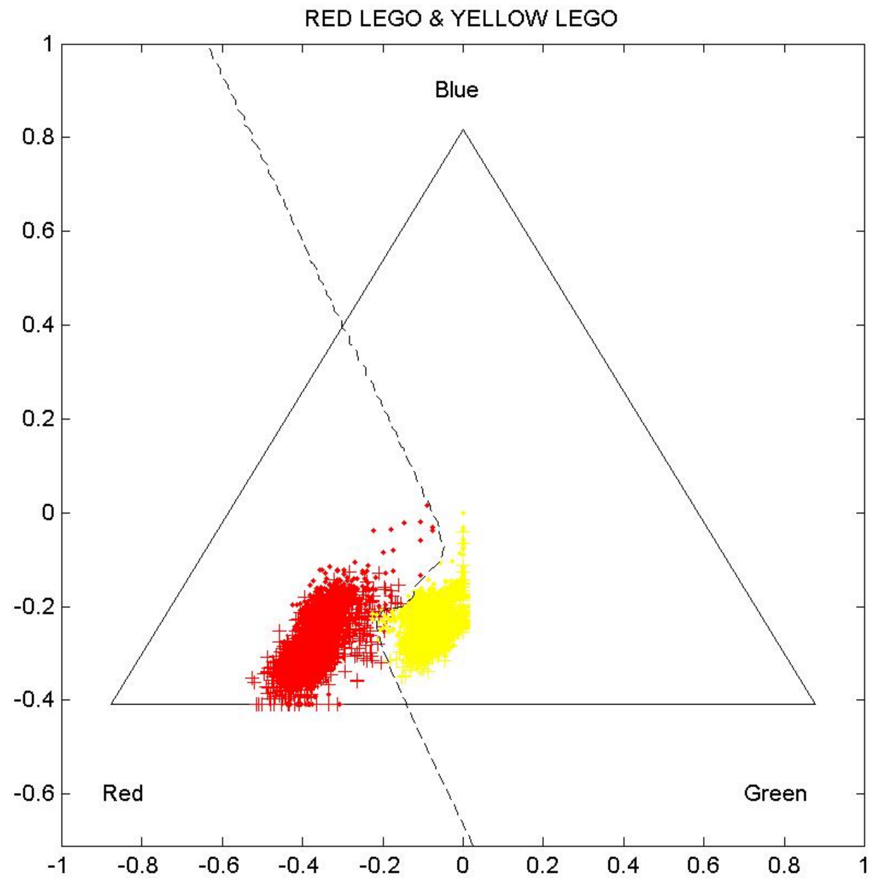


Output is 6D:
{red, blue, yellow, green, black, white}

$$\mathbf{c} \in C^6$$

MLP classifier

2-3-1 MLP
Levenberg-Marquardt

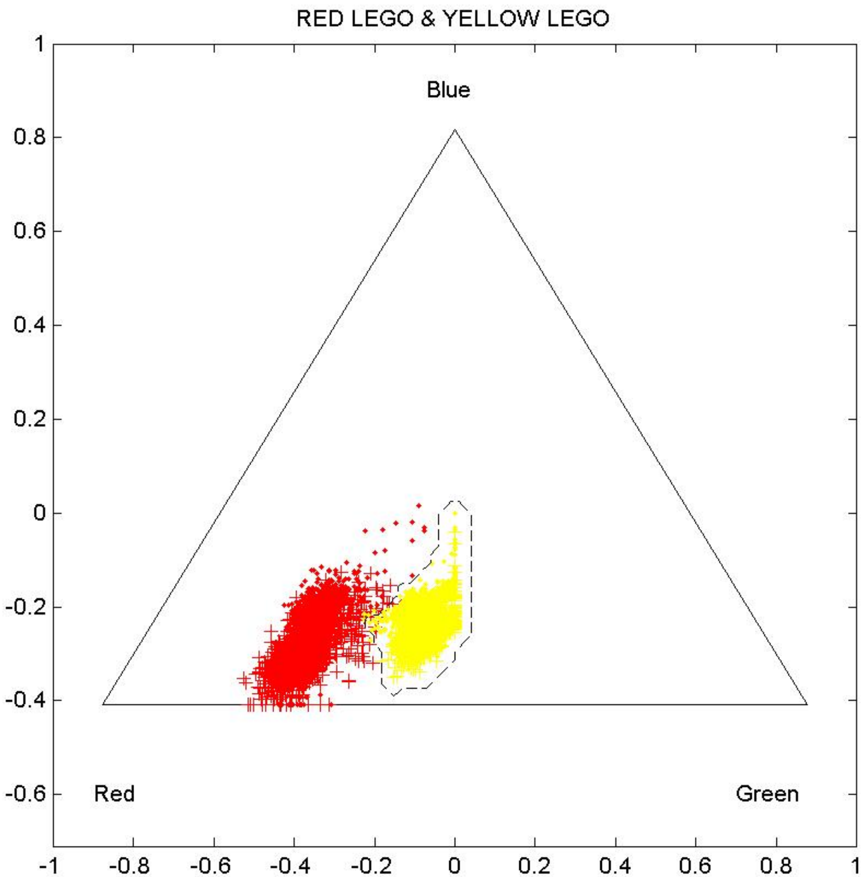


E_train = 0.21%
E_test = 0.24%

Training time (150 epochs): 51 seconds

SVM classifier

SVM with
γ = 1000



E_train = 0.19%
E_test = 0.20%

Training time:
22 seconds

$$K(\mathbf{x},\mathbf{y}) = \exp\left[-\gamma(\mathbf{x}-\mathbf{y})^2\right]$$

Machine Learning

- Machine learning (multilayer perceptrons, support vector machines, clustering) is covered in great detail in the course ”Learning Systems”.