Probabilistic Classification

Based on the ML lecture by Raymond J. Mooney
University of Texas at Austin

Probabilistic Classification – Idea

Imagine that

- I look out of a window and see a bird,
- it is black, approx. 25 cm long, and has a rather yellow beak.

My daughter asks: What kind of bird is this?

My usual answer: This is *probably* a kind of blackbird (kos černý in Czech).

Here *probably* means that out of my extensive catalogue of four kinds of birds that I am able to recognize, "blackbird" gets the highest degree of belief based on *features* of this particular bird.

Frequentists might say that the largest proportion of birds with similar features I have ever seen were blackbirds.

The degree of belief (Bayesians), or the relative frequency (frequentists) is the *probability*.

Basic Discrete Probability Theory

- A finite or countably infinite set Ω of *possible outcomes*, Ω is called *sample space*.

  Experiment: Roll one dice once. Sample space: $\Omega = \{1, \ldots, 6\}$

- Each element ω of Ω is assigned a "probability" value $f(\omega)$, here f must satisfy
  - $f(\omega) \in [0, 1]$ for all $\omega \in \Omega$,
  - $\sum_{\omega \in \Omega} f(\omega) = 1$.

  If the dice is fair, then $f(\omega) = \frac{1}{6}$ for all $\omega \in \{1, \ldots, 6\}$.

- An *event* is any subset E of Ω.
- The *probability* of a given event $E \subseteq \Omega$ is defined as

$$P(E) = \sum_{\omega \in E} f(\omega)$$

  Let E be the event that an odd number is rolled, i.e., $E = \{1, 3, 5\}$. Then $P(E) = \frac{1}{2}$.

- Basic laws: $P(\Omega) = 1$, $P(\emptyset) = 0$, given disjoint sets A, B we have $P(A \cup B) = P(A) + P(B)$, $P(\Omega \setminus A) = 1 - P(A)$.

Conditional Probability and Independence

- $P(A \mid B)$ is the probability of A given B (assume $P(B) > 0$) defined by

  $$P(A \mid B) = P(A \cap B)/P(B)$$

  (We assume that B is all and only information known.)

  A fair dice: what is the probability that 3 is rolled assuming that an odd number is rolled? ... and assuming that an even number is rolled?

- **The law of total probability:** Let A be an event and $B_1, \ldots, B_n$ pairwise disjoint events such that $\Omega = \bigcup_{i=1}^{n} B_i$. Then

  $$P(A) = \sum_{i=1}^{n} P(A \cap B_i) = \sum_{i=1}^{n} P(A \mid B_i) \cdot P(B_i)$$

- A and B are independent if $P(A \cap B) = P(A) \cdot P(B)$.

  It is easy to show that if $P(B) > 0$, then

  A, B are independent iff $P(A \mid B) = P(A)$.

Random Variables

- A *random variable* X is a function $X : \Omega \to \mathbb{R}$.
  A dice: $X : \{1, \dots, 6\} \to \{0, 1\}$ such that $X(n) = n \mod 2$.
- A *probability mass function (pmf)* of X is a function p defined by

$$p(x) := P(X = x)$$

Often $P(X)$ is used to denote the pmf of X.

Random Vectors

- A *random vector* is a function $X : \Omega \to \mathbb{R}^d$.

  We use $X = (X_1, \ldots, X_d)$ where X_i is a random variable returning the i-th component of X.

- A *joint probability mass function* of X is $p_X(x_1, \ldots, x_d) := P(X_1 = x_1 \wedge \cdots \wedge X_d = x_d)$.

  I.e., p_X gives the probability of every combination of values.

  Often, $P(X_1, \cdots, X_d)$ denotes the joint pmf of $X_1, \ldots, X_d$. That is, $P(X_1, \cdots, X_d)$ stands for probabilities $P(X_1 = x_1 \wedge \cdots \wedge X_d = x_d)$ for all possible combinations of $x_1, \ldots, x_d$.

- The probability mass function p_{X_i} of each X_i is called *marginal probability mass function*. We have

$$p_{X_i}(x_i) = P(X_i = x_i) = \sum_{(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d)} p_X(x_1, \ldots, x_d)$$

Random Vectors – Example

Let Ω be a space of colored geometric shapes that are divided into two categories (positive and negative).

Assume a random vector $X = (X_{color}, X_{shape}, X_{cat})$ where

- $X_{color} : \Omega \to \{red, blue\}$,
- $X_{shape} : \Omega \to \{circle, square\}$,
- $X_{cat} : \Omega \to \{pos, neg\}$.

The joint pmf is given by the following tables:

positive:

	circle	square
red	0.2	0.02
blue	0.02	0.01

negative:

	circle	square
red	0.05	0.3
blue	0.2	0.2

Random Vectors – Example

The probability of all possible events can be calculated by summing the appropriate probabilities.

$$\begin{aligned} P(red \wedge circle) &= P(X_{color} = red \ \wedge \ X_{shape} = circle) \\ &= P(red \wedge circle \wedge positive) + \\ &\quad + P(red \wedge circle \wedge negative) \\ &= 0.2 + 0.05 = 0.25 \end{aligned}$$

$$P(red) = 0.2 + 0.02 + 0.05 + 0.3 = 0.57$$

Thus also all conditional probabilities can be computed:

$$P(positive \mid red \wedge cicle) = \frac{P(positive \wedge red \wedge circle)}{P(red \wedge circle)} = \frac{0.2}{0.25} = 0.8$$

Conditional Probability Mass Functions

We often have to deal with a pmf of a random vector X conditioned on values of a random vector Y.

I.e., we are interested in $P(X = x \mid Y = y)$ for all x and y.

We write $P(X \mid Y)$ to denote the pmf of X conditioned on Y.

Technically, $P(X \mid Y)$ is a function which takes a possible value x of X and a possible value y of Y and returns $P(X = x \mid Y = y)$.

This allows us to say, e.g., that two variables X_1 and X_2 are independent conditioned on Y by

$$P(X_1, X_2 \mid Y) = P(X_1 \mid Y) \cdot P(X_2 \mid Y)$$

Technically this means that for all possible values x_1 of X_1, all possible values x_2 of X_2, and all possible values y of Y we have

$$P(X_1 = x_1 \land X_2 = x_2 \mid Y = y) = \\ P(X_1 = x_1 \mid Y = y) \cdot P(X_2 = x_2 \mid Y = y)$$

Bayesian Classification

Let Ω be a sample space (a universum) of all objects that can be classified.

We assume a probability P on Ω.

A *training set* will be a subset of Ω randomly sampled according to P.

- Let Y be the random variable for the category which takes values in $\{y_1, \ldots, y_m\}$.
- Let X be the random vector describing n features of a given instance, i.e., $X = (X_1, \ldots, X_n)$
  - Denote by x_k possible values of X,
  - and by x_{ij} possible values of X_i.

Bayes classifier: Given a vector of feature values x_k,

$$C^{Bayes}(x_k) := y_\ell \text{ where } \ell = \arg\max_{i \in \{1,\ldots,m\}} P(Y = y_i \mid X = x_k)$$

Intuitively, C^{Bayes} assigns x_k to the most probable category it might be in.

Bayesian Classification – Example

Imagine a conveyor belt with apples and apricots.

A machine is supposed to correctly distinguish apples from apricots based on their weight and diameter.

That is,

- $Y = \{apple, apricot\}$,
- $X = (X_{weight}, X_{diam})$.

Assume that we are given a fruit that weighs $40g$ with $5cm$ diameter.

The Bayes classifier compares $P(Y = apple \mid X = (40g, 5cm))$ with $P(Y = apricot \mid X = (40g, 5cm))$ and selects the more probable category given the features.

Optimality of the Bayes Classifier

Let C be an arbitrary *classifier*, that is a function that to every x_k assigns a class out of $\{y_1, \ldots, y_m\}$.

Slightly abusing notation, we use C to also denote the random variable which assigns a category to every instance.

(Technically this is a composition $C \circ X$ of C and X which first determines the features using X and then classifies according to C).

Define the error of the classifier C by

$$E_C = P(Y \neq C)$$

Věta

The Bayes classifier C^{Bayes} minimizes E_C, that is

$$E_{C^{Bayes}} := \min_{C \text{ is a classifier}} E_C$$

Optimality of the Bayes Classifier

$$\begin{aligned}
E_C &= \sum_{i=1}^{m} P(Y = y_i \wedge C \neq y_i) \\
&= 1 - \sum_{i=1}^{m} P(Y = y_i \wedge C = y_i) \\
&= 1 - \sum_{i=1}^{m} \sum_{x_k} P(Y = y_i \wedge C = y_i \mid X = x_k) P(X = x_k) \\
&= 1 - \sum_{x_k} P(X = x_k) \sum_{i=1}^{m} P(Y = y_i \wedge C = y_i \mid X = x_k) \\
&= 1 - \sum_{x_k} P(X = x_k) P(Y = C(x_k) \mid X = x_k)
\end{aligned}$$

(Here the last equality follows from the fact that C is determined by x_k.)

Choosing

$$C(x_k) = C^{Bayes}(x_k) = y_\ell \text{ where } \ell = \operatorname*{arg\,max}_{i \in \{1,\ldots,m\}} P(Y = y_i \mid X = x_k)$$

maximizes $P(Y = C(x_k) \mid X = x_k)$ and thus minimizes E_C.

Practical Use of Bayes Classifier

The crucial problem: How to compute $P(Y = y_i \mid X = x_k)$?

Given no other assumptions, this requires a table giving the probability of each category for each possible vector of feature values, which is impossible to accurately estimate from a reasonably-sized training set.

Concretely, if all $Y, X_1, \ldots, X_n$ are binary, we need 2^n numbers to specify $P(Y = 0 \mid X = x_k)$ for each possible x_k.
(Note that we do not need to specify
$P(Y = 1 \mid X = x_k) = 1 - P(Y = 0 \mid X = x_k)$).

It is a bit better than $2^{n+1} - 1$ entries for specification of the complete joint pmf $P(Y, X_1, \ldots, X_n)$.

However, it is still too large for most classification problems.

Let’s Look at It the Other Way Round

Věta (Bayes,1764)

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

Důkaz.

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{\frac{P(A \cap B)}{P(A)} \cdot P(A)}{P(B)} = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

□

Bayesian Classification

Determine the category for x_k by finding y_i maximizing

$$P(Y = y_i \mid X = x_k) \quad = \quad \frac{P(Y = y_i) \cdot P(X = x_k \mid Y = y_i)}{P(X = x_k)}$$

So in order to make the classifier we need to compute:

- The prior $P(Y = y_i)$ for every y_i
- The conditionals $P(X = x_k \mid Y = y_i)$ for every x_k and y_i

Estimating the Prior and Conditionals

- $P(Y = y_i)$ can be easily estimated from data:
  - Given a set of p training examples where
  - n_i of the examples are in the category y_i,
  - we set

$$P(Y = y_i) = \frac{n_i}{p}$$

- If the dimension of features is small, $P(X = x_k \mid Y = y_i)$ can be estimated from data similarly as for $P(Y = y_i)$.

  Unfortunately, for higher dimensional data too many examples are needed to estimate all $P(X = x_k \mid Y = y_i)$ (there are too many x_k's).

  So where is the advantage of using the Bayes thm.?

We introduce *independence assumptions* about the features!

Naive Bayes

- We assume that features of an instance are (conditionally) independent *given the category*:

$$P(X \mid Y) = P(X_1, \cdots, X_n \mid Y) = \prod_{i=1}^{n} P(X_i \mid Y)$$

- Therefore, we only need to specify $P(X_i \mid Y)$, that is $P(X_i = x_{ij} \mid Y = y_k)$ for each possible pair of a feature-value x_{ij} and a class y_k.

  Note that if Y and all X_i are binary (values in $\{0, 1\}$), this requires specifying only $2n$ parameters:

$$P(X_i = 1 \mid Y = 1) \text{ and } P(X_i = 1 \mid Y = 0) \text{ for each } X_i$$

  since $P(X_i = 0 \mid Y) = 1 - P(X_i = 1 \mid Y)$.

Compared to specifying 2^n parameters without any independence assumptions.

Naive Bayes – Example

	positive	negative
P(Y)	0.5	0.5
$P(small \mid Y)$	0.4	0.4
$P(medium \mid Y)$	0.1	0.2
$P(large \mid Y)$	0.5	0.4
$P(red \mid Y)$	0.9	0.3
$P(blue \mid Y)$	0.05	0.3
$P(green \mid Y)$	0.05	0.4
$P(square \mid Y)$	0.05	0.4
$P(triangle \mid Y)$	0.05	0.3
$P(circle \mid Y)$	0.9	0.3

Is $(medium, red, circle)$ positive?

	positive	negative
P(Y)	0.5	0.5
$P(medium \mid Y)$	0.1	0.2
$P(red \mid Y)$	0.9	0.3
$P(circle \mid Y)$	0.9	0.3

Denote $x_k = (medium, red, circle)$.

$$\begin{aligned} P(pos \mid X = x_k) = & \\ & = P(pos) \cdot P(medium \mid pos) \cdot P(red \mid pos) \cdot P(circle \mid pos) \,/\, P(X = x_k) \\ & = 0.5 \cdot 0.1 \cdot 0.9 \cdot 0.9 \,/\, P(X = x_k) = 0.0405/P(X = x_k) \end{aligned}$$

$$\begin{aligned} P(neg \mid X = x_k) = & \\ & = P(neg) \cdot P(medium \mid neg) \cdot P(red \mid neg) \cdot P(circle \mid neg) \,/\, P(X = x_k) \\ & = 0.5 \cdot 0.2 \cdot 0.3 \cdot 0.3 \,/\, P(X = x_k) = 0.009/P(X = x_k) \end{aligned}$$

Apparently,

$$P(pos \mid X = x_k) = 0.0405/P(X = x_k) > 0.009/P(X = x_k) = P(neg \mid X = x_k)$$

So we classify x_k as positive.

Estimating Probabilities (In General)

- Normally, probabilities are estimated on observed frequencies in the training data (see the previous example).
- Let us have
  - n_k training examples in class y_k,
  - n_{ijk} of these n_k examples have the value for X_i equal to x_{ij}.

  Then we put $\bar{P}(X_i = x_{ij} \mid Y = y_k) = \frac{n_{ijk}}{n_k}$.
- **A problem:** If, by chance, a rare value x_{ij} of a feature X_i never occurs in the training data, we get

$$\bar{P}(X_i = x_{ij} \mid Y = y_k) = 0 \quad \text{for all } k \in \{1, \ldots, m\}$$

  But then $\bar{P}(X = x_k) = 0$ for x_k containing the value x_{ij} for X_i, and thus $\bar{P}(Y = y_k \mid X = x_k)$ is not well defined. Moreover, $\bar{P}(Y = y_k) \cdot \bar{P}(X = x_k \mid Y = y_k) = 0$ (for all y_k) so even this cannot be used for classification.

Probability Estimation Example

Training data:

Size	Color	Shape	Class
small	red	circle	pos
large	red	circle	pos
small	red	triangle	neg
large	blue	circle	neg

Learned probabilities:

	positive	negative
$\bar{P}(Y)$	0.5	0.5
$\bar{P}(small \mid Y)$	0.5	0.5
$\bar{P}(medium \mid Y)$	0	0
$\bar{P}(large \mid Y)$	0.5	0.5
$\bar{P}(red \mid Y)$	1	0.5
$\bar{P}(blue \mid Y)$	0	0.5
$\bar{P}(green \mid Y)$	0	0
$\bar{P}(square \mid Y)$	0	0
$\bar{P}(triangle \mid Y)$	0	0.5
$\bar{P}(circle \mid Y)$	1	0.5

Note that $\bar{P}(medium \wedge red \wedge circle) = 0$.

So what is $\bar{P}(pos \mid medium \wedge red \wedge circle)$?

Smoothing

- To account for estimation from small samples, probability estimates are adjusted or *smoothed*.
- *Laplace smoothing* using an m-estimate works *as if*
  - each feature is given a prior probability p,
  - such feature have been observed with this probability p in a sample of size m (recall that m is the number of classes).

  We get

$$\bar{P}(X_i = x_{ij} \mid Y = y_k) = \frac{n_{ijk} + mp}{n_k + m}$$

  (Recall that n_k is the number of training examples of class y_k, and n_{ijk} is the number of training examples of class y_k for which the i-th feature X_i has the value x_{ij}.)

Laplace Smothing Example

- Assume training set contains 10 positive examples:
  - 4 small
  - 0 medium
  - 6 large
- Estimate parameters as follows ($m = 2$ and $p = 1/3$)
  - $\bar{P}(small \mid positive) = (4 + 2/3)/(10 + 2) = 0.389$
  - $\bar{P}(medium \mid positive) = (0 + 2/3)/(10 + 2) = 0.056$
  - $\bar{P}(large \mid positive) = (6 + 2/3)/(10 + 2) = 0.556$

Continuous Features

Ω may be (potentially) continuous, X_i may assign a continuum of values in $\mathbb{R}$.

- The probabilities are computed using *probability density* $p : \mathbb{R} \to \mathbb{R}^+$ instead of pmf.

  A random variable $X : \Omega \to \mathbb{R}^+$ has a density $p : \mathbb{R} \to \mathbb{R}^+$ if for every interval $[a, b]$ we have

  $$P(a \leq X \leq b) = \int_a^b p(x)dx$$

  Usually, $P(X_i \mid Y = y_k)$ is used to denote the *density* of X_i conditioned on $Y = y_k$.
- The densities $P(X_i \mid Y = y_k)$ are usually estimated using Gaussian densities as follows:
  - Estimate the mean μ_{ik} and the standard deviation σ_{ik} based on training data.
  - Then put

  $$\bar{P}(X_i \mid Y = y_k) = \frac{1}{\sigma_{ik}\sqrt{2\pi}} \exp\left(\frac{-(X_i - \mu_{ik})^2}{2\sigma_{ik}^2}\right)$$

Comments on Naive Bayes

- Tends to work well despite rather strong assumption of conditional independence of features.
- Experiments show it to be quite competitive with other classification methods.

  Even if the probabilities are not accurately estimeted, it often picks the correct maximum probability category.
- Directly constructs a hypothesis from parameter estimates that are calculated from the training data.
- Typically handles noise well.
- Missing values are easy to deal with (simply average over all missing values in feature vectors).

Bayes Classifier vs MAP vs MLE

Recall that the Bayes classifier chooses the category as follows:

$$\begin{aligned} C^{Bayes}(x_k) &= \operatorname*{arg\,max}_{i \in \{1,\dots,m\}} P(Y = y_i \mid X = x_k) \\ &= \operatorname*{arg\,max}_{i \in \{1,\dots,m\}} \frac{P(Y = y_i) \cdot P(X = x_k \mid Y = y_i)}{P(X = x_k)} \end{aligned}$$

As the denominator $P(X = x_k)$ is not influenced by i, the Bayes is equivalent to the Maximum Aposteriori Probability rule:

$$C^{MAP}(x_k) = \operatorname*{arg\,max}_{i \in \{1,\dots,m\}} P(Y = y_i) \cdot P(X = x_k \mid Y = y_i)$$

If we do not care about the prior (or assume uniform) we may use the Maximum Likelihood Estimate rule:

$$C^{MLE}(x_k) = \operatorname*{arg\,max}_{i \in \{1,\dots,m\}} P(X = x_k \mid Y = y_i)$$

(Intuitively, we maximize the probability that the data x_k have been generated into the category y_i.)

Bayesian Networks (Basic Information)

In the Naive Bayes we have assumed that *all* features $X_1, \ldots, X_n$ are independent.

This is usually not realistic.

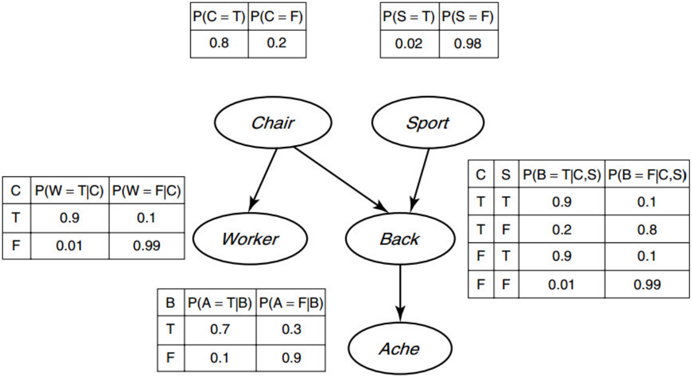
E.g. Variables "rain" and "grass wet" are (usually) strongly dependent.

What if we return some dependencies back?

(But now in a well-defined sense.)

Bayesian networks are a graphical model that uses a directed acyclic graph to specify dependencies among variables.

Bayesian Networks – Example



P(C = T)	P(C = F)
0.8	0.2

P(S = T)	P(S = F)
0.02	0.98

C	P(W = T|C)	P(W = F|C)
T	0.9	0.1
F	0.01	0.99

C	S	P(B = T|C,S)	P(B = F|C,S)
T	T	0.9	0.1
T	F	0.2	0.8
F	T	0.9	0.1
F	F	0.01	0.99

B	P(A = T|B)	P(A = F|B)
T	0.7	0.3
F	0.1	0.9

Now, e.g.,

$$P(C, S, W, B, A) = P(C) \cdot P(S) \cdot P(W \mid C) \cdot P(B \mid C, S) \cdot P(A \mid B)$$

Now we may e.g. infer what is the probability $P(C = T \mid A = T)$ that we sit in a bad chair assuming that our back aches.

We have to store only 10 numbers as opposed to $2^5 - 1$ if the whole joint pmf is stored.

Bayesian Networks – Learning & Naive Bayes

Many algorithms have been developed for learning:

- the structure of the graph of the network,
- the *conditional probability tables*.

The methods are based on maximum-likelihood estimation, gradient descent, etc.

Automatic procedures are usually combined with expert knowledge.

Can you express the naive Bayes for $Y, X_1, \dots, X_n$ using a Bayesian network?

Numerical features

- Throughout this lecture we assume that all features are numerical, i.e. feature vectors belong to $\mathbb{R}^n$.
- Most non-numerical features can be conveniently transformed to numerical ones.

  For example:

  - Colors $\{blue, red, yellow\}$ can be represented by

    $$\{(1,0,0),(0,1,0),(0,0,1)\}$$

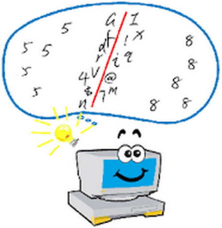
    (one-hot encoding)
  - A black-and-white picture of $x \times y$ pixels can be encoded as a vector of xy numbers that capture the shades of gray of the pixels.

Basic Problems

We consider two basic problems:

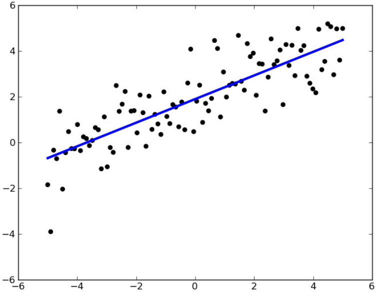
- (Binary) classification

  Our goal: Classify inputs into two categories.



- Function approximation (regression)

  Our goal: Find a (hypothesized) functional dependency in data.



Binary classification in $\mathbb{R}^n$

- Assume
  - a set of instances $X \subseteq \mathbb{R}^n$,
  - an *unknown* categorization function $c : X \rightarrow \{0, 1\}$.
- **Our goal:**
  - Given a set D of training examples of the form $(\vec{x}, c(\vec{x}))$ where $\vec{x} \in X$,
  - construct a hypothesized categorization function $h \in \mathcal{H}$ that is consistent with c on the training examples, i.e.,

$$h(\vec{x}) = c(\vec{x}) \text{ for all training examples } (\vec{x}, c(\vec{x})) \in D$$

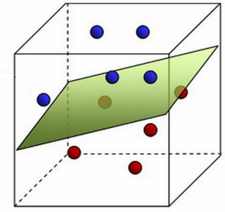
Comments:

- In practice, we often do not strictly demand $h(\vec{x}) = c(\vec{x})$ for all training examples $(\vec{x}, c(\vec{x})) \in D$ (often it is impossible)
- We are more interested in good **generalization**, that is how well h classifies new instances that do not belong to D.
  - Recall that we usually evaluate accuracy of the resulting hypothesized function h on a test set.

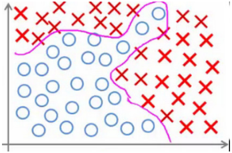
Hypothesis Spaces

We consider two kinds of hypothesis spaces:

- Linear (affine) classifiers (this lecture)



- Classifiers based on combinations of linear and sigmoidal functions (classical neural networks) (next lecture)



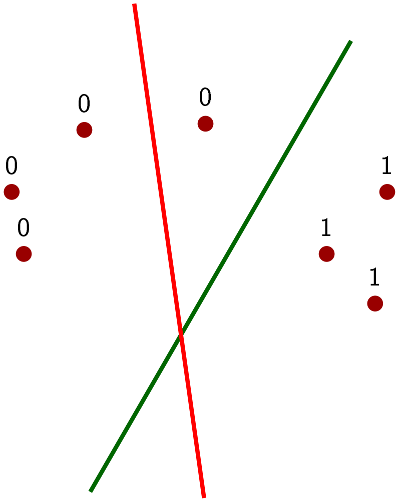
Length and Scalar Product of Vectors

- We consider vectors $\vec{x} = (x_1, \ldots, x_m) \in \mathbb{R}^m$.
- Typically, we use Euclidean metric on vectors: $|\vec{x}| = \sqrt{\sum_{i=1}^{m} x_i^2}$
  The distance between two vectors (points) $\vec{x}, \vec{y}$ is $|\vec{x} - \vec{y}|$.
- We use the *scalar product* $\vec{x} \cdot \vec{y}$ of vectors $\vec{x} = (x_1, \ldots, x_m)$ and $\vec{y} = (y_1, \ldots, y_m)$ defined by

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{m} x_i y_i$$

  - Recall that $\vec{x} \cdot \vec{y} = |\vec{x}||\vec{y}| \cos\theta$ where θ is the angle between $\vec{x}$ and $\vec{y}$. That is $\vec{x} \cdot \vec{y}$ is the length of the projection of $\vec{y}$ on $\vec{x}$ multiplied by $|\vec{x}|$.
  - Note that $\vec{x} \cdot \vec{x} = |\vec{x}|^2$

Linear classifier - example



- classification in plane using a linear classifier
- if a point is incorrectly classified, the learning algorithm turns the line (hyperplane) to improve the classification.

Linear Classifier

A *linear classifier* $h[\vec{w}]$ is determined by a vector of *weights* $\vec{w} = (w_0, w_1, \ldots, w_n) \in \mathbb{R}^{n+1}$ as follows:

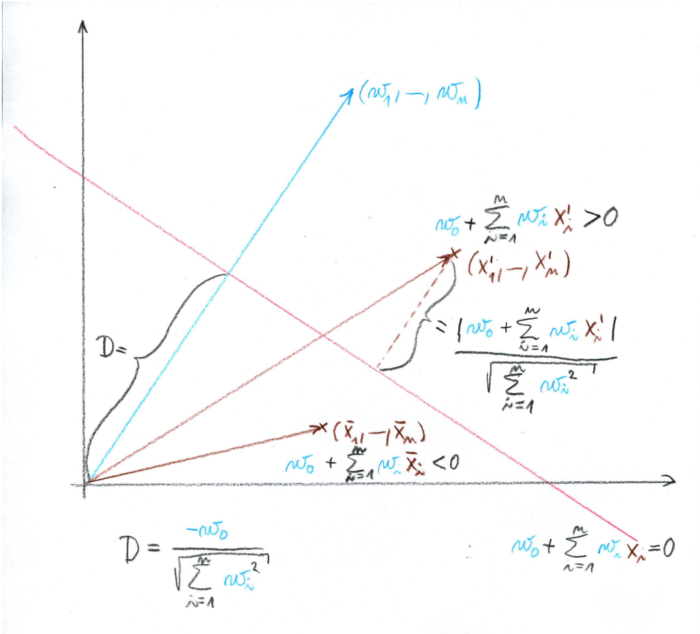
Given $\vec{x} = (x_1, \ldots, x_n) \in X \subseteq \mathbb{R}^n$,

$$h[\vec{w}](\vec{x}) := \begin{cases} 1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i \geq 0 \\ 0 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i < 0 \end{cases}$$

More succinctly:

$$h(\vec{x}) = sgn\left(w_0 + \sum_{i=1}^{n} w_i \cdot x_i\right) \qquad \text{where} \qquad sgn(y) = \begin{cases} 1 & y \geq 0 \\ 0 & y < 0 \end{cases}$$

Linear Classifier – Geometry


$(w_1, —, w_m)$
$w_0 + \sum_{i=1}^{m} w_i x'_i > 0$
$(x'_1, —, x'_m)$
$= \frac{|w_0 + \sum_{i=1}^{m} w_i x'_i|}{\sqrt{\sum_{i=1}^{m} w_i^2}}$
$D =$
$(\bar{x}_1 — \bar{x}_m)$
$w_0 + \sum_{i=1}^{m} w_i \bar{x}_i < 0$
$D = \frac{-w_0}{\sqrt{\sum_{i=1}^{m} w_i^2}}$
$w_0 + \sum_{i=1}^{m} w_i x_i = 0$

Linear Classifier – Notation

Given $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ we define an *augmented feature vector*

$$\tilde{\mathbf{x}} = (x_0, x_1, \ldots, x_n) \quad \text{where } x_0 = 1$$

This makes the notation for the linear classifier more succinct:

$$h[\vec{w}](\vec{x}) = sgn(\vec{w} \cdot \tilde{\mathbf{x}})$$

Perceptron Learning

- Given a training set

  $$D = \{(\vec{x}_1, c(\vec{x}_1)), (\vec{x}_2, c(\vec{x}_2)), \ldots, (\vec{x}_p, c(\vec{x}_p))\}$$

  Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in X \subseteq \mathbb{R}^n$ and $c(\vec{x}_k) \in \{0, 1\}$.

  We write c_k instead of $c(\vec{x}_k)$.
  Note that $\tilde{\mathbf{x}}_k = (x_{k0}, x_{k1} \ldots, x_{kn})$ where $x_{k0} = 1$.

- A weight vector $\vec{w} \in \mathbb{R}^{n+1}$ is **consistent with** D if

  $$h[\vec{w}](\vec{x}_k) = sgn(\vec{w} \cdot \tilde{\mathbf{x}}_k) = c_k \quad \text{for all } k = 1, \ldots, p$$

  D is **linearly separable** if there is a vector $\vec{w} \in \mathbb{R}^{n+1}$ which is consistent with D.

- Our goal is to find a consistent $\vec{w}$ assuming that D is linearly separable.

Perceptron – Learning Algorithm

Online learning algorithm:

Idea: Cyclically go through the training examples in D and adapt weights. Whenever an example is incorrectly classified, turn the hyperplane so that the example becomes closer to it's correct half-space.

Compute a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- $\vec{w}^{(0)}$ is randomly initialized close to $\vec{0} = (0, \ldots, 0)$
- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:

$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \left(h[\vec{w}^{(t)}](\vec{x}_k) - c_k\right) \cdot \tilde{\mathbf{x}}_k \\ &= \vec{w}^{(t)} - \varepsilon \cdot \left(sgn\left(\vec{w}^{(t)} \cdot \tilde{\mathbf{x}}_k\right) - c_k\right) \cdot \tilde{\mathbf{x}}_k\end{aligned}$$

Here $k = (t \mod p) + 1$, i.e. the examples are considered cyclically, and $0 < \varepsilon \leq 1$ is a **learning speed**.

Věta (Rosenblatt)

If D is linearly separable, then there is t^* such that $\vec{w}^{(t^*)}$ is consistent with D.

Example

Training set:

$$D = \{((2,-1),1),((2,1),1),((1,3),0)\}$$

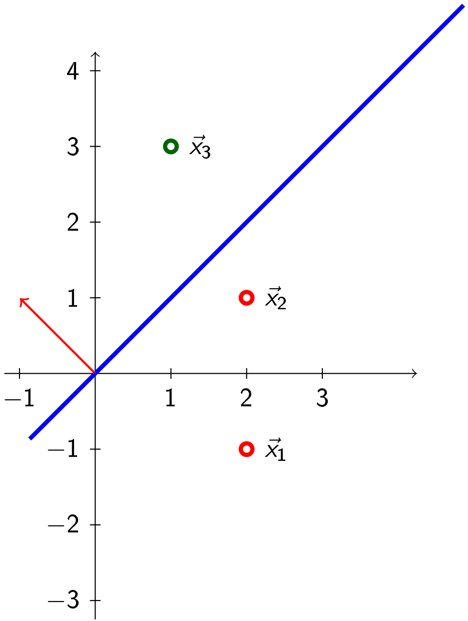
That is

$$\begin{aligned} \vec{x}_1 &= (2,-1) & \qquad \tilde{\mathbf{x}}_1 &= (1,2,-1) \\ \vec{x}_2 &= (2,1) & \qquad \tilde{\mathbf{x}}_2 &= (1,2,1) \\ \vec{x}_3 &= (1,3) & \qquad \tilde{\mathbf{x}}_3 &= (1,1,3) \end{aligned}$$

$$\begin{aligned} c_1 &= 1 \\ c_2 &= 1 \\ c_3 &= 0 \end{aligned}$$

Assume that the initial vector $\vec{w}^{(0)}$ is $\vec{w}^{(0)} = (0,-1,1)$.
Consider $\varepsilon = 1$.

Example: Separating by $\vec{w}^{(0)}$



Denoting $\vec{w}^{(0)} = (w_0, w_1, w_2) = (0, -1, 1)$ the blue separating line is given by $w_0 + w_1 x_1 + w_2 x_2 = 0$.

The red vector normal to the blue line is (w_1, w_2).

The points on the side of (w_1, w_2) are assigned 1 by the classifier, the others zero. (In this case $\vec{x}_3$ is assigned one and $\vec{x}_1, \vec{x}_2$ are assigned zero, all of this is inconsistent with $c_1 = 1, c_2 = 1, c_3 = 0$.)

Example: $\vec{w}^{(1)}$

We have

$$\vec{w}^{(0)} \cdot \tilde{\mathbf{x}}_1 = (0, -1, 1) \cdot (1, 2, -1) = 0 - 2 - 1 = -3$$

thus

$$sgn\left(\vec{w}^{(0)} \cdot \tilde{\mathbf{x}}_1\right) = 0$$

and thus

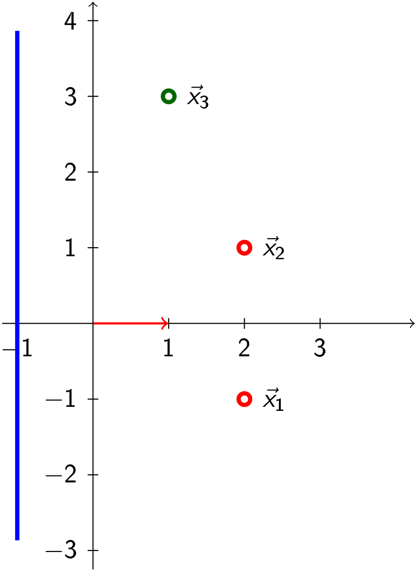
$$sgn\left(\vec{w}^{(0)} \cdot \tilde{\mathbf{x}}_1\right) - c_1 = 0 - 1 = -1$$

(This means that $\vec{x}_1$ is not well classified, and $\vec{w}^{(0)}$ is not consistent with D.)

Hence,

$$\begin{aligned} \vec{w}^{(1)} &= \vec{w}^{(0)} - \left(sgn\left(\vec{w}^{(0)} \cdot \tilde{\mathbf{x}}_1\right) - c_1\right) \cdot \tilde{\mathbf{x}}_1 \\ &= \vec{w}^{(0)} + \tilde{\mathbf{x}}_1 \\ &= (0, -1, 1) + (1, 2, -1) \\ &= (1, 1, 0) \end{aligned}$$

Example



Example: Separating by $\vec{w}^{(1)}$

We have

$$\vec{w}^{(1)} \cdot \tilde{\mathbf{x}}_2 = (1,1,0) \cdot (1,2,1) = 1 + 2 = 3$$

thus

$$sgn\left(\vec{w}^{(1)} \cdot \tilde{\mathbf{x}}_2\right) = 1$$

and thus

$$sgn\left(\vec{w}^{(1)} \cdot \tilde{\mathbf{x}}_2\right) - c_2 = 1 - 1 = 0$$

(This means that $\vec{x}_2$ *is* currently well classified by $\vec{w}^{(1)}$. However, as we will see, $\vec{x}_3$ is not well classified.)

Hence,

$$\vec{w}^{(2)} = \vec{w}^{(1)} = (1,1,0)$$

Example: $\vec{w}^{(3)}$

We have

$$\vec{w}^{(2)} \cdot \tilde{\mathbf{x}}_3 = (1,1,0) \cdot (1,1,3) = 1 + 1 = 2$$

thus

$$sgn\left(\vec{w}^{(2)} \cdot \tilde{\mathbf{x}}_3\right) = 1$$

and thus

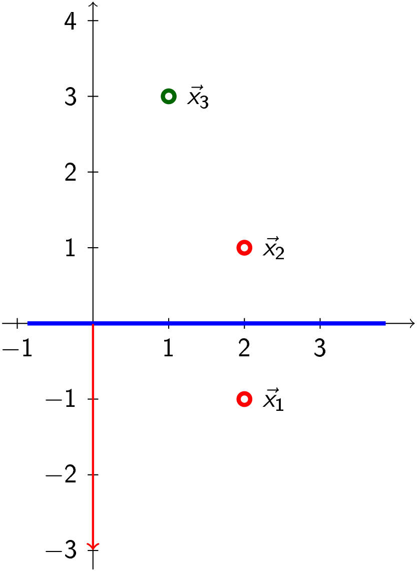
$$sgn\left(\vec{w}^{(2)} \cdot \tilde{\mathbf{x}}_3\right) - c_3 = 1 - 0 = 1$$

(This means that $\vec{x}_3$ is not well classified, and $\vec{w}^{(2)}$ is not consistent with D.)

Hence,

$$\begin{aligned}\vec{w}^{(3)} &= \vec{w}^{(2)} - \left(sgn\left(\vec{w}^{(2)} \cdot \tilde{\mathbf{x}}_3\right) - c_3\right) \cdot \tilde{\mathbf{x}}_3 \\ &= \vec{w}^{(2)} - \tilde{\mathbf{x}}_3 \\ &= (1,1,0) - (1,1,3) \\ &= (0,0,-3)\end{aligned}$$

Example: Separating by $\vec{w}^{(3)}$



Example: $\vec{w}^{(4)}$

We have

$$\vec{w}^{(3)} \cdot \tilde{\mathbf{x}}_1 = (0, 0, -3) \cdot (1, 2, -1) = 3$$

thus

$$sgn\left(\vec{w}^{(3)} \cdot \tilde{\mathbf{x}}_1\right) = 1$$

and thus

$$sgn\left(\vec{w}^{(3)} \cdot \tilde{\mathbf{x}}_1\right) - c_1 = 1 - 1 = 0$$

(This means that $\vec{x}_1$ *is* currently well classified by $\vec{w}^{(3)}$. However, as we will see, $\vec{x}_2$ is not.)

Hence,

$$\vec{w}^{(4)} = \vec{w}^{(3)} = (0, 0, -3)$$

Example: $\vec{w}^{(5)}$

We have

$$\vec{w}^{(4)} \cdot \tilde{\mathbf{x}}_2 = (0, 0, -3) \cdot (1, 2, 1) = -3$$

thus

$$sgn\left(\vec{w}^{(4)} \cdot \tilde{\mathbf{x}}_2\right) = 0$$

and thus

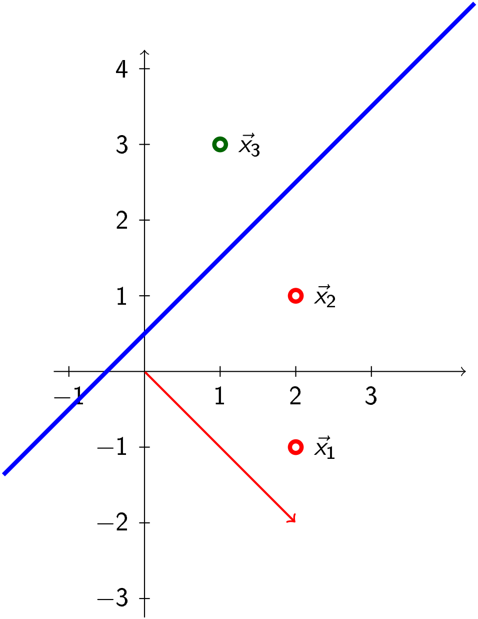
$$sgn\left(\vec{w}^{(4)} \cdot \tilde{\mathbf{x}}_2\right) - c_2 = 0 - 1 = -1$$

(This means that $\vec{x}_2$ is not well classified, and $\vec{w}^{(4)}$ is not consistent with D.)

Hence,

$$\begin{aligned}
\vec{w}^{(5)} &= \vec{w}^{(4)} - \left(sgn\left(\vec{w}^{(4)} \cdot \tilde{\mathbf{x}}_2\right) - c_2\right) \cdot \tilde{\mathbf{x}}_2 \\
&= \vec{w}^{(4)} + \tilde{\mathbf{x}}_2 \\
&= (0, 0, -3) + (1, 2, 1) \\
&= (1, 2, -2)
\end{aligned}$$

Example: Separating by $\vec{w}^{(5)}$



Example: The result

The vector $\vec{w}^{(5)}$ is consistent with D:

$$sgn\left(\vec{w}^{(5)} \cdot \tilde{\mathbf{x}}_1\right) = sgn\left((1,2,-2)\cdot(1,2,-1)\right) = sgn(7) = 1 = c_1$$

$$sgn\left(\vec{w}^{(5)} \cdot \tilde{\mathbf{x}}_2\right) = sgn\left((1,2,-2)\cdot(1,2,1)\right) = sgn(3) = 1 = c_2$$

$$sgn\left(\vec{w}^{(5)} \cdot \tilde{\mathbf{x}}_3\right) = sgn\left((1,2,-2)\cdot(1,1,3)\right) = sgn(-3) = 0 = c_3$$

Perceptron – Learning Algorithm

Batch learning algorithm:

Compute a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

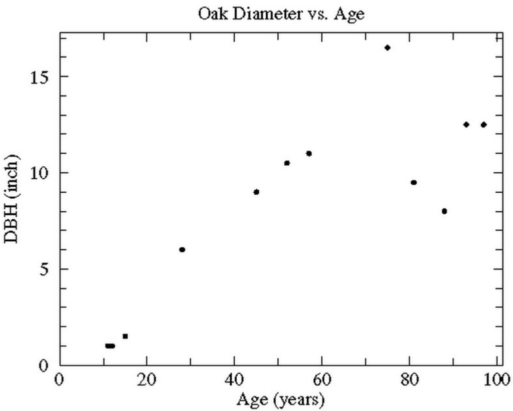
- $\vec{w}^{(0)}$ is randomly initialized close to $\vec{0} = (0, \ldots, 0)$
- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:

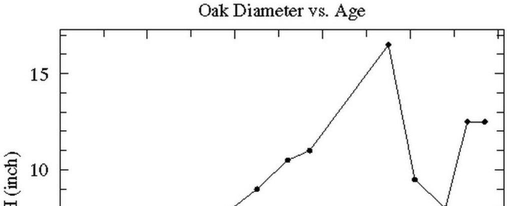
$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(h[\vec{w}^{(t)}](\vec{x}_k) - c_k \right) \cdot \tilde{\mathbf{x}}_k \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(sgn \left(\vec{w}^{(t)} \cdot \tilde{\mathbf{x}}_k \right) - c_k \right) \cdot \tilde{\mathbf{x}}_k\end{aligned}$$

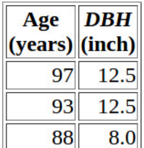
Here $k = (t \mod p) + 1$, i.e. the examples are considered cyclically, and $0 < \varepsilon \leq 1$ is a **learning speed**.

Function Approximation – Oaks in Wisconsin

This example is from *How to Lie with Statistics* by Darrell Huff (1954)







Age (years)	*DBH* (inch)
97	12.5
93	12.5
88	8.0

Function Approximation

- Assume
    - a set $X \subseteq \mathbb{R}^n$ of instances,
    - an *unknown* function $f : X \to \mathbb{R}$.
- **Our goal:**
    - Given a set D of training examples of the form $(\vec{x}, f(\vec{x}))$ where $\vec{x} \in X$,
    - construct a hypothesized function $h \in \mathcal{H}$ such that

      $$h(\vec{x}) \approx f(\vec{x}) \text{ for all training examples } (\vec{x}, f(\vec{x})) \in D$$

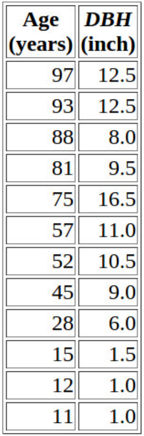
      Here $\approx$ means that the values are somewhat close to each other w.r.t. an appropriate *error function* E.
- In what follows we use the *least squares* defined by

  $$E = \frac{1}{2} \sum_{(\vec{x}, f(\vec{x})) \in D} (f(\vec{x}) - h(\vec{x}))^2$$

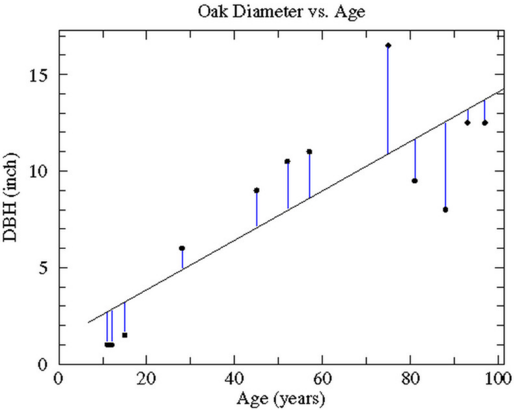
  Our goal is to minimize E.

  The main reason is that this function has nice mathematical properties (as opposed e.g. to $\sum_{(\vec{x}, f(\vec{x})) \in D} |f(\vec{x}) - h(\vec{x})|$).

Least Squares – Oaks in Wisconsin



Age (years)	*DBH* (inch)
97	12.5
93	12.5
88	8.0
81	9.5
75	16.5
57	11.0
52	10.5
45	9.0
28	6.0
15	1.5
12	1.0
11	1.0



Linear Function Approximation

- Given a set D of training examples:

  $$D = \{(\vec{x}_1, f(\vec{x}_1)), (\vec{x}_2, f(\vec{x}_2)), \ldots, (\vec{x}_p, f(\vec{x}_p))\}$$

  Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in \mathbb{R}^n$ and $f_k(\vec{x}) \in \mathbb{R}$.
  Recall that $\tilde{\mathbf{x}}_k = (x_{k0}, x_{k1} \ldots, x_{kn})$.

  Our goal: Find $\vec{w}$ so that $h[\vec{w}](\vec{x}) = \vec{w} \cdot \tilde{\mathbf{x}}$ approximates the function f some of whose values are given by the training set.

- **Least Squares Error Function:**

  $$E(\vec{w}) = \frac{1}{2} \sum_{k=1}^{p} (\vec{w} \cdot \tilde{\mathbf{x}}_k - f_k)^2 = \frac{1}{2} \sum_{k=1}^{p} \left(\sum_{i=0}^{n} w_i x_{ki} - f_k \right)^2$$

Gradient of the Error Function

Consider the **gradient** of the error function:

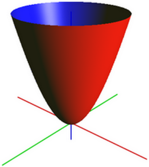
$$\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_0}(\vec{w}), \ldots, \frac{\partial E}{\partial w_n}(\vec{w}) \right) = \sum_{k=1}^{p} (\vec{w} \cdot \tilde{\mathbf{x}}_k - f_k) \cdot \tilde{\mathbf{x}}_k$$

What is the gradient $\nabla E(\vec{w})$? It is a vector in $\mathbb{R}^{n+1}$ which points in the direction of the steepest *ascent* of E (it's length corresponds to the steepness). Note that here the vectors $\tilde{\mathbf{x}}_k$ are *fixed* parameters of E!

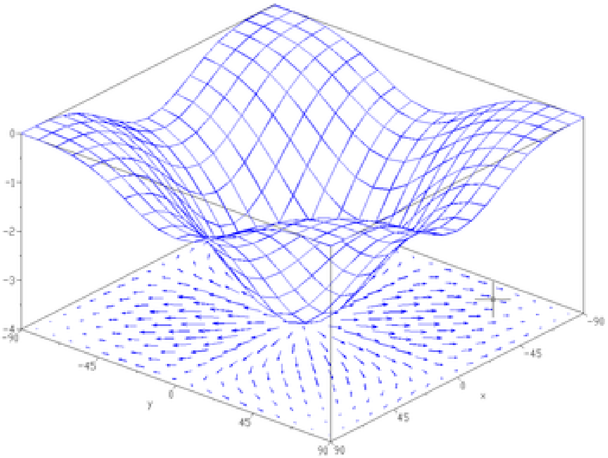
Fakt

If $\nabla E(\vec{w}) = \vec{0} = (0, \ldots, 0)$, *then* $\vec{w}$ *is a global minimum of* E.

This follows from the fact that E is a convex paraboloid that has a unique extreme which is a minimum.



Gradient – illustration



Function Approximation – Learning

Gradient Descent:

- Weights $\vec{w}^{(0)}$ are initialized randomly close to $\vec{0}$.
- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:

$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \tilde{\mathbf{x}}_k - f_k \right) \cdot \tilde{\mathbf{x}}_k \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(h[\vec{w}^{(t)}](\vec{x}_k) - f_k \right) \cdot \tilde{\mathbf{x}}_k\end{aligned}$$

Here $k = (t \mod p) + 1$ and $0 < \varepsilon \leq 1$ is the learning speed.

Note that the algorithm is almost similar to the batch perceptron algorithm!

Tvrzení

For sufficiently small $\varepsilon > 0$ the sequence $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$ converges (component-wisely) to the global minimum of E.

Finding the Minimum in Dimension One

Assume $n = 1$. Then the error function E is

$$E(w_0, w_1) = \frac{1}{2}\sum_{k=1}^{p}(w_0 + w_1 x_k - f_k)^2$$

Minimize E w.r.t. w_0 a w_1:

$$\frac{\delta E}{\delta w_0} = 0 \quad \Leftrightarrow \quad w_0 = \bar{f} - w_1\bar{x} \quad \Leftrightarrow \quad \bar{f} = w_0 + w_1\bar{x}$$

where $\bar{x} = \frac{1}{p}\sum_{k=1}^{p} x_k$ a $\bar{f} = \frac{1}{p}\sum_{k=1}^{p} f_k$

$$\frac{\delta E}{\delta w_1} = 0 \quad \Leftrightarrow \quad w_1 = \frac{\frac{1}{p}\sum_{k=1}^{p}(f_k - \bar{f})(x_k - \bar{x})}{\frac{1}{p}\sum_{k=1}^{p}(x_k - \bar{x})^2}$$

i.e. $w_1 = cov(f, x)/var(x)$

Finding the Minimum in Arbitrary Dimension

Let A be a matrix $p \times (n+1)$ (p rows, $n+1$ columns) whose k-th row is the vector $\tilde{\mathbf{x}}_k$.

Let $\vec{f} = (f_1, \dots, f_p)^\top$ be the *column* vector formed by values of f in the training set.

Then

$$\nabla E(\vec{w}) = 0 \quad \Leftrightarrow \quad \vec{w} = (A^\top A)^{-1} A^\top \vec{f}$$

if $(A^\top A)^{-1}$ exists

(Then $(A^\top A)^{-1} A^\top$ is the so called Moore-Penrose pseudoinverse of A.)

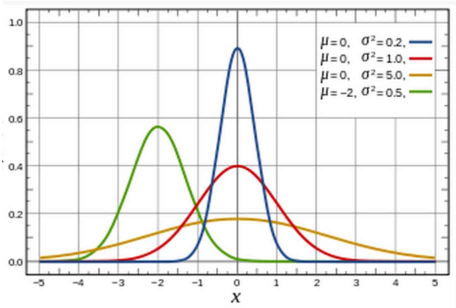
Normal Distribution – Reminder

Distribution of continuous random variables.

Density (one dimensional, that is over $\mathbb{R}$):

$$p(x) \quad = \quad \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ -\frac{(x-\mu)^2}{2\sigma^2} \right\} \quad =: \quad N[\mu, \sigma^2](x)$$

μ is the expected value (the mean), σ^2 is the variance.



Maximum Likelihood vs Least Squares (Dim 1)

Fix a training set $D = \{(x_1, f_1), (x_2, f_2), \ldots, (x_p, f_p)\}$
Assume that each f_k has been generated randomly by

$$f_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Here

- w_0, w_1 are **unknown numbers**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Denote by $p(f_1, \ldots, f_p \mid w_0, w_1, \sigma^2)$ the probability density according to which the values $f_1, \ldots, f_n$ were generated assuming fixed $w_0, w_1, \sigma^2, x_1, \ldots, x_p$.

(For interested: The independence and normality imply

$$p(f_1, \ldots, f_p \mid w_0, w_1, \sigma^2) = \prod_{k=1}^{p} N[w_0 + w_1 x_k, \sigma^2](f_k)$$

where $N[w_0 + w_1 x_k, \sigma^2](f_k)$ is a normal distribution with the mean $w_0 + w_1 x_k$ and the variance σ^2.)

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $f_1, \ldots, f_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(f_1, \ldots, f_p \mid w_0, w_1, \sigma^2)$$

Věta

(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes $E(w_0, w_1)$, i.e. the least squares error function.*

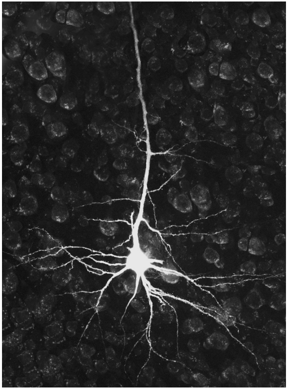
Note that the maximizing/minimizing (w_0, w_1) does not depend on σ^2.

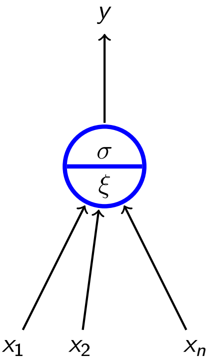
Maximizing σ^2 satisfies $\sigma^2 = \frac{1}{p} \sum_{k=1}^{p} (f_k - w_0 - w_1 \cdot x_k)^2$.

Comments on Linear Models

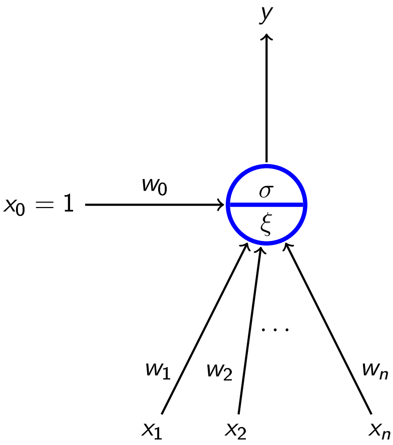
- Linear models are parametric, i.e. they have a fixed form with a small number of parameters that need to be learned from data (as opposed e.g. to decision trees where the structure is not fixed in advance).
- Linear models are stable, i.e. small variations in the training data have only limited impact on the learned model. (tree models typically vary more with the training data).
- Linear models are less likely to overfit (low variance) the training data but sometimes tend to underfit (high bias).

(Primitive) Mathematical Model of Neuron





Formal neuron



- $x_1, \dots, x_n$ real *inputs*
- x_0 special input, always 1
- $w_0, w_1, \dots, w_n$ real *weights*
- $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$ *inner potential*;
  In general, other potentials are considered (e.g. Gaussian), more on this in PV021.
- y *output* defined by $y = \sigma(\xi)$ where σ is an *activation function*.
  We consider several activation functions.

  e.g. *linear threshold function*

$$\sigma(\xi) = sgn(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Formal Neuron vs Linear Models

Both linear classifier and linear (affine) function are special cases of the formal neuron.

- If σ is a linear threshold function

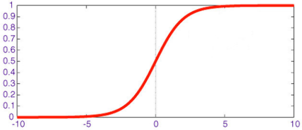
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

  we obtain a linear classifier.
- If σ is identity, i.e. $\sigma(\xi) = \xi$, we obtain a linear (affine) function.

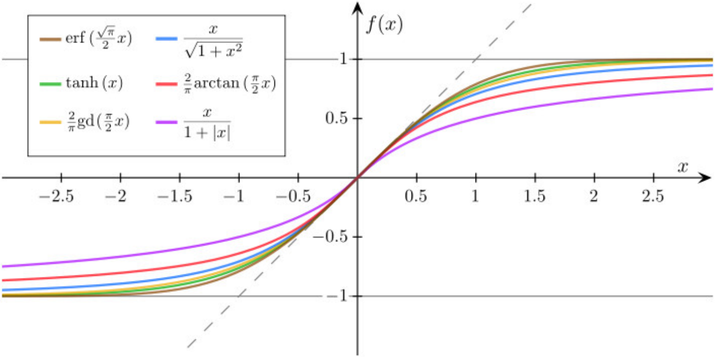
Many more activation functions are used in neural networks!

Sigmoid Functions

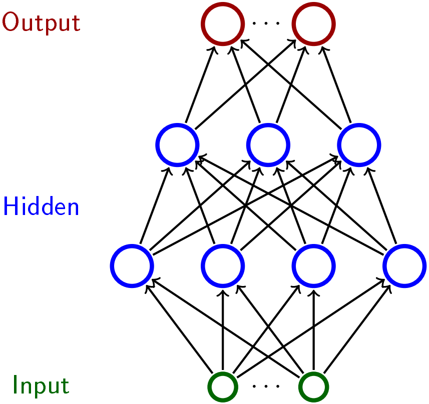
Logistic sigmoid $\sigma(\xi) = \dfrac{1}{1 + e^{-\xi}}$



Others ...



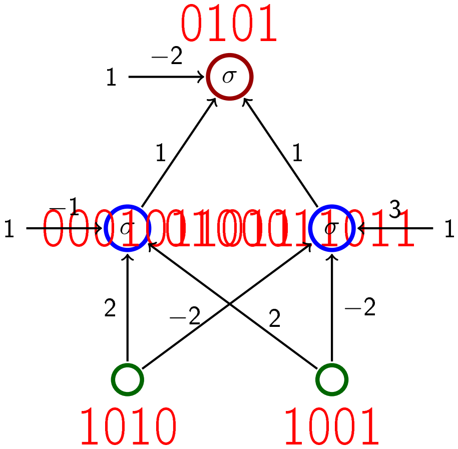
Multilayer Perceptron (MLP)



- Neurons are organized in *layers*
  (input layer, output layer, possibly several hidden layers)
- Layers are numbered from 0;
  the input is 0-th
- Neurons in the ℓ-th layer are connected with all neurons in the $\ell + 1$-th layer

Intuition: The network computes a function as follows: Assign input values to the input neurons and 0 to the rest. Proceed upwards through the layers, one layer per step. In the ℓ-th step consider output values of neurons in $\ell - 1$-th layer as inputs to neurons of the ℓ-th layer. Compute output values of neurons in the ℓ-th layer.

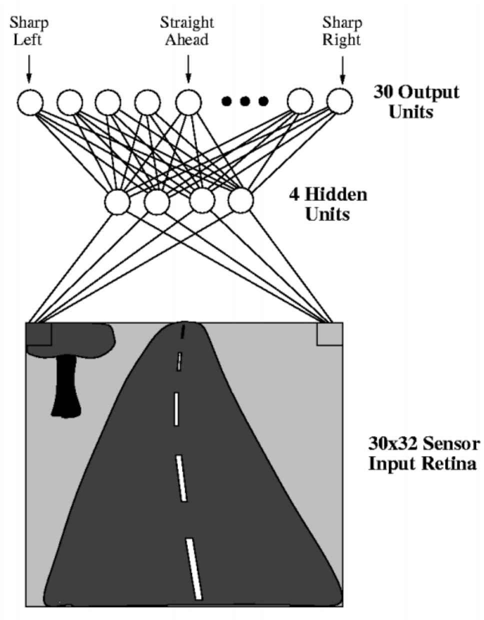
Example



- Activation function: linear threshold

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Classical Example – ALVINN

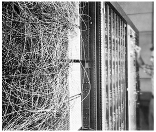


- One of the first autonomous car driving systems (in 90s)
- ALVINN drives a car
- The net has $30 \times 32 = 960$ input neurons (the input space is $\mathbb{R}^{960}$).
- The value of each input captures the shade of gray of the corresponding pixel.
- Output neurons indicate where to turn (to the center of gravity).

Source: `http://jmvidal.cse.sc.edu/talks/ann/alvin.html`

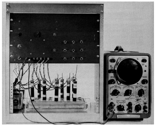
A Bit of History

- Perceptron (Rosenblatt et al, 1957)



  - Single layer (i.e. no hidden layers), the activation function *linear threshold*

    (i.e., a bit more general linear classifier)
  - Perceptron learning algorithm
  - Used to recognize numbers
- Adaline (Widrow & Hof, 1960)



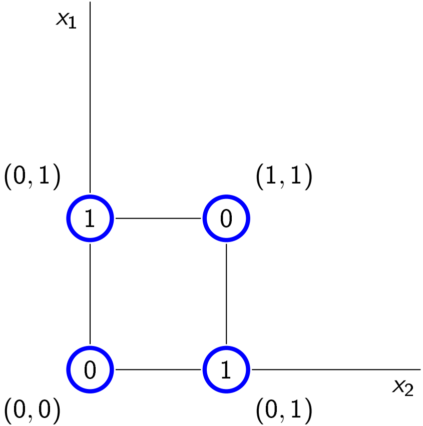
  - Single layer, the activation function *identity*

    (i.e., a bit more linear function)
  - Online version of the gradient descent
  - Used a new circuitry element called *memristor* which was able to "remember"history of current in form of resistance

In both cases, the expressive power is rather limited – can express only linear decision boundaries and linear (affine) functions.

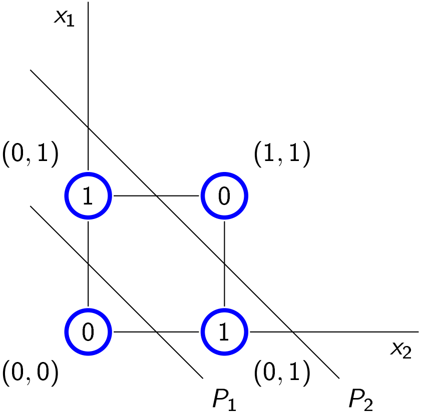
A Bit of History

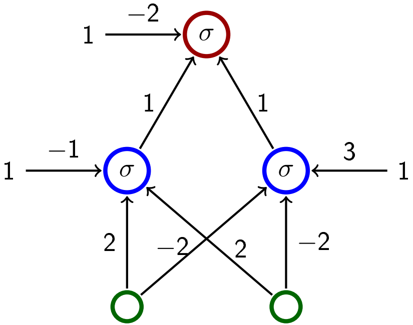
One of the famous (counter)-examples: XOR



No perceptron can distinguish between ones and zeros.

XOR vs Multilayer Perceptron





(Here σ is a linear threshold function.)

$P_1 : -1 + 2x_1 + 2x_2 = 0$ $P_2 : 3 - 2x_1 - 2x_2 = 0$

The output neuron performs an intersection of half-spaces.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function with range $(0, 1)$.

  Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

So multi-layer perceptrons are suffuciently powerful for any application.

But for a long time, at least throughout 60s and 70s, nobody well-known knew any efficient method for training multilayer networks!

... then the **backpropagation** appeared in 1986!

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g. i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)
- w_{ji} is the weight of the arc **from** the neuron i **to** the neuron j.
- $j_{\leftarrow}$ is the set of all neurons from which there are edges to j

  (i.e. $j_{\leftarrow}$ is the layer directly below j)
- $j^{\rightarrow}$ is the set of all neurons to which there are edges from j.

  (i.e. $j^{\rightarrow}$ is the layer directly above j)

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- for simplicity, the activation function of every neuron will be the logistic sigmoid $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$.

  (We may of course consider logistic sigmoids with different steepness paramaters, or other sigmoidal functions, more in PV021.)

- A value of a non-input neuron $j \in Z \setminus X$ when the computation is finished is $y_j = \sigma(\xi_j)$

  (y_j is determined by weights $\vec{w}$ and a given input $\vec{x}$, so it's sometimes written as $y_j[\vec{w}](\vec{x})$)

- Fixing weights of all neurons, the network computes a function $F[\vec{w}] : \mathbb{R}^{|X|} \to \mathbb{R}^{|Y|}$ as follows: Assign values of a given vector $\vec{x} \in \mathbb{R}^{|X|}$ to the input neurons, evaluate the network, then $F[\vec{w}](\vec{x})$ is the vector of values of the output neurons.

  Here we implicitly assume a fixed orderings on input and output vectors.

MLP – Learning

- Given a set D of training examples:

$$D = \left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $\vec{d}_k \in \mathbb{R}^{|Y|}$. We write d_{kj} to denote the value in $\vec{d}_k$ corresponding to the output neuron j.

- **Least Squares Error Function:** Let $\vec{w}$ be a vector of all weights in the network.

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} (y_j[\vec{w}](\vec{x}_k) - d_{kj})^2$$

MLP – Learning Algorithm

Batch Learning – Gradient Descent:

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \dots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0
- in the step $t+1$ (here $t = 0, 1, 2\dots$) is $\vec{w}^{(t+1)}$ computed as follows:

  $$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

  where

  $$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

  is the weight change w_{ji} and $0 < \varepsilon(t) \leq 1$ is the learning speed in the step $t+1$.

Note that $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$ is a component of ∇E, i.e. the weight change in the step $t+1$ can be written as follows: $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

MLP – Gradient Computation

For every weight w_{ji} we have (obviously)

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

So now it suffices to compute $\frac{\partial E_k}{\partial w_{ji}}$, that is the error for a fixed training example $(\vec{x}_k, d_k)$.

It holds that

$$\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$$

where

$$\delta_j = y_j - d_{kj} \qquad \text{pro } j \in Y$$

$$\delta_j = \sum_{r \in j^{\rightarrow}} \delta_r \cdot y_r(1 - y_r) \cdot w_{rj} \qquad \text{pro } j \in Z \smallsetminus (Y \cup X)$$

(Here $y_r = y[\vec{w}](\vec{x}_k)$ where $\vec{w}$ are the current weights and $\vec{x}_k$ is the input of the k-th training example.)

Multilayer Perceptron – Backpropagation

So to compute all $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$:

Compute all $\frac{\partial E_k}{\partial w_{ji}} = \delta_j \cdot y_j(1 - y_j) \cdot y_i$ for every training example $(\vec{x}_k, \vec{d}_k)$:

- Evaluate all values y_i of neurons using the standard bottom-up procedure with the input $\vec{x}_k$.
- Compute δ_j using *backpropagation* through layers top-down :
  - Assign $\delta_j = y_j - d_{kj}$ for all $j \in Y$
  - In the layer ℓ, assuming that δ_r has been computed for all neurons r in the layer $\ell + 1$, compute

    $$\delta_j = \sum_{r \in j^{\rightarrow}} \delta_r \cdot y_r(1 - y_r) \cdot w_{rj}$$

    for all j from the ℓ-th layer.

Example

Assume $w_{30}^{(0)} = w_{50}^{(0)} = w_{41}^{(0)} = w_{42}^{(0)} = w_{54}^{(0)} = 1$ and $w_{40}^{(0)} = w_{31}^{(0)} = w_{32}^{(0)} = w_{53}^{(0)} = -1$. Consider a training set $\{((1,0),1)\}$.

Then

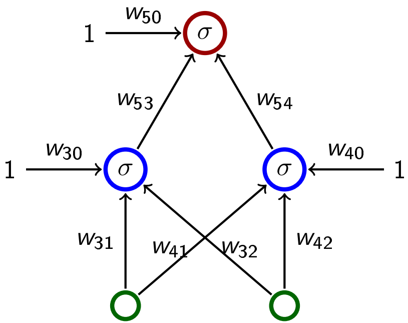
$y_1 = 1,$

$y_2 = 0,$

$y_3 = \sigma(w_{30} + w_{31}^{(0)} y_1 + w_{32}^{(0)} y_2) = 0.5,$

$y_4 = 0.5,$

$y_5 = 0.731058.$



$\delta_5 = y_5 - 1 = -0.268942,$

$\delta_4 = \delta_5 \cdot y_5 \cdot (1 - y_5) * w_{54}^{(0)} = -0.052877,$

$\delta_3 = 0.052877.$

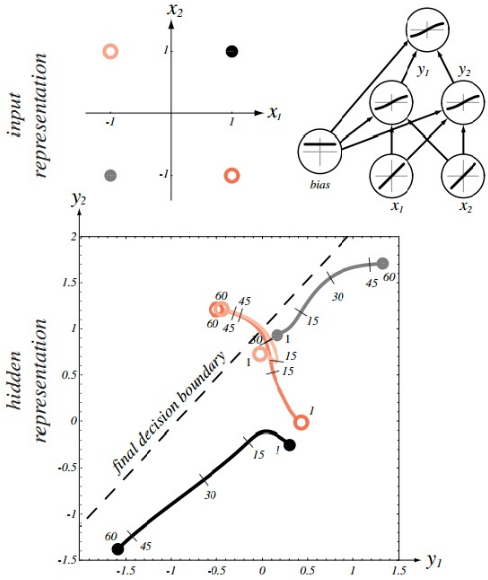
$$\frac{\partial E_1}{\partial w_{53}} = \delta_5 \cdot y_5 \cdot (1 - y_5) \cdot y_3 = -0.026438,$$

$$\frac{\partial E_1}{\partial w_{54}} = \delta_5 \cdot y_5 \cdot (1 - y_5) \cdot y_4 = -0.026438,$$

$$\frac{\partial E_1}{\partial w_{30}} = \delta_3 \cdot y_3 \cdot (1 - y_3) \cdot 1 = 0.01321925,$$

....

Illustration of Gradient Descent – XOR



Source: Pattern Classification (2nd Edition); Richard O. Duda, Peter E. Hart, David G. Stork

Comments on Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- In practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (random restarts).
    - Take results of trial with lowest training set error.
    - Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).

There are many more issues concerning learning efficiency (data normalization, selection of activation functions, weight initialization, training speed, efficiency of the gradient descent itself etc.) – see PV021.

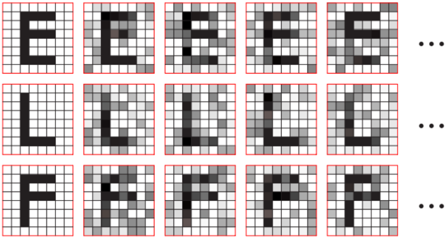
Hidden Neurons Representations

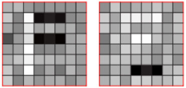
Trained hidden neurons can be seen as newly constructed features.

E.g., in a two layer network used for classification, the hidden layer transforms the input so that important features become explicit (and hence the result may become linearly separable).

Consider a two-layer MLP, 64-2-3 for classification of letters (three output neurons, each corresponds to one of the letters).

sample training patterns

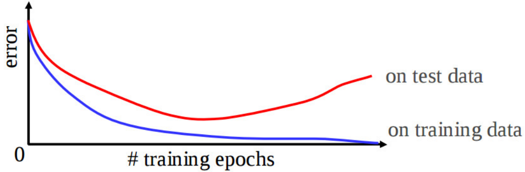




learned input-to-hidden weights

Overfitting

- Due to their expressive power, neural networks are quite sensitive to overfitting.

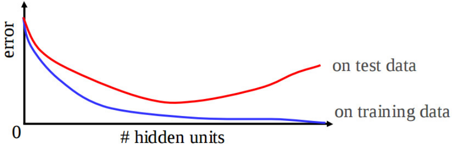


- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase the validation error.

Overfitting – The Number of Hidden Neurons

- Too few hidden neurons prevent the network from adequately fitting the data.
- Too many hidden units can result in overfitting.

  (There are advanced methods that prevent overfitting even for rich models, such as regularization, where the error function penalizes overfitting – see PV021.)



- Use cross-validation to empirically determine an optimal number of hidden units.

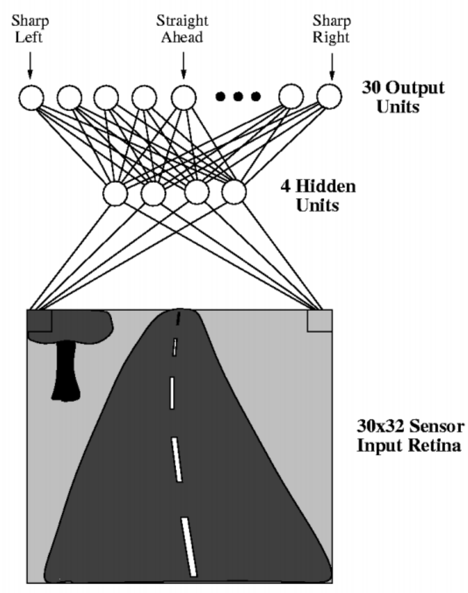
  There are methods that automatically construct the structure of the network based on data, they are not much used though.

Applications

- Text to Speech and vice versa
- Fraud detection
- finance & business predictions
- Game playing (backgammon is a classical example, AlphaGo is the modern one)
- Image recognition

  This is the main area in which the current state-of-the-art deep networks excel.
- (artificial brain and intelligence)
- ...

ALVINN


Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina

ALVINN

- Two layer MLP, $960 - 4 - 30$ (sometimes $960 - 5 - 30$)
- Inputs correspond to pixels.
- Sigmoidal activation function (logistic sigmoid).
- Direction corresponds to the center of gravity.

  I.e., output neurons are considered as points of mass evenly distributed along a line. Weight of each neuron corresponds to its value.

ALVINN – Training

Trained while driving.

- A camera captured the road from the front window, approx. 25 pictures per second
- Training examples $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road
  - $\vec{d}_k \approx$ corresponding direction of the steering wheel set by the driver
- the values $\vec{d}_k$ computed using Gaussian distribution:

$$d_{ki} = e^{-D_i^2/10}$$

where D_i is the distance between the i-th output from the one that corresponds to the real direction of the steering wheel.

(This is better than the binary output because similar road directions induce similar reaction of the driver.)

Selection of Training Examples

Naive approach: just take images from the camera.

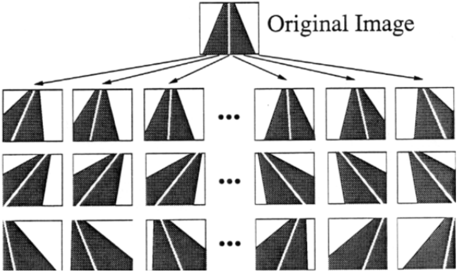
Problems:

- A too good driver never teaches the network how to solve deviations from the right track. Couple of harsh solutions:
  - turn the learning off for a moment, deviate from the right track, then turn on the learning and let the network learn how to solve the situation.
  - let the driver go crazy! (a bit dangerous, expensive, unreliable)
- Images are very similar (the network basically sees the road from the right lane), may be overtrained.

Selection of Training Examples

Problem with too good driver were solved as follows:

- every image of the road has been has been transformed to 15 slightly different copies



Repetitiveness of images was solved as follows:

- the system has a buffer of 200 images (including the 15 copies of the current one), in every round trains on these images
- afterwards, a new image is captured, 15 copies made, and these new 15 substitute 15 selected from the buffer (10 with the smallest training error, 5 randomly)

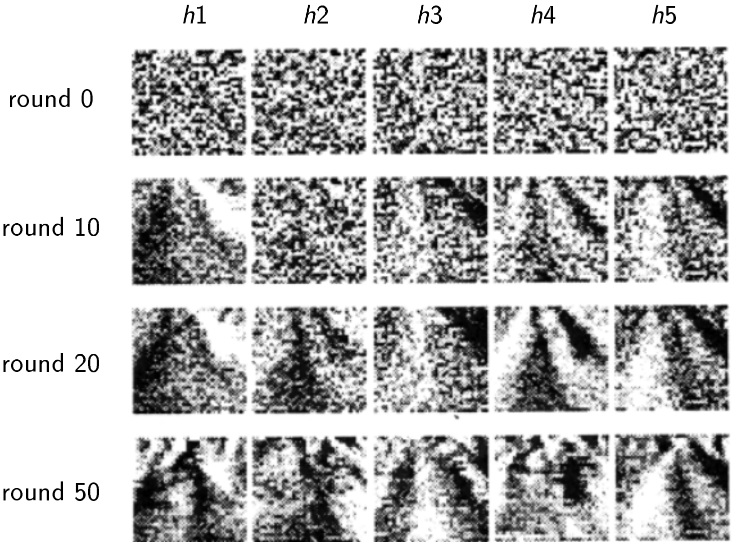
ALVINN – Training

- standard backpropagation
- constant speed of learning (possibly different for each neuron – see PV021)
- some other optimizations (see PV021)

Výsledek:

- Training took 5 minutes, the speed was 4 miles per hour
  (The speed was limited by the hydraulic controller of the steering wheel not the learning algorithm.)
- ALVINN was able to go through roads it never "seen" and in different weather

ALVINN – Weight Learning



Here $h1, \ldots, h5$ are values of hidden neurons.

Extensions and Directions (PV021)

- Other types of learning inspired by neuroscience – Hebbian learning
- More biologically plausible models of neural networks – spiking neurons

  This goes into the direction of HUGE area of (computational) neuroscience, only very lightly touched in PV021.
- Unsupervised learning – Self-Organizing Maps
- Reinforcement learning
  - learning to make decisions, or play games, sequentially
  - neural networks have been used – temporal difference learning

Deep Learning

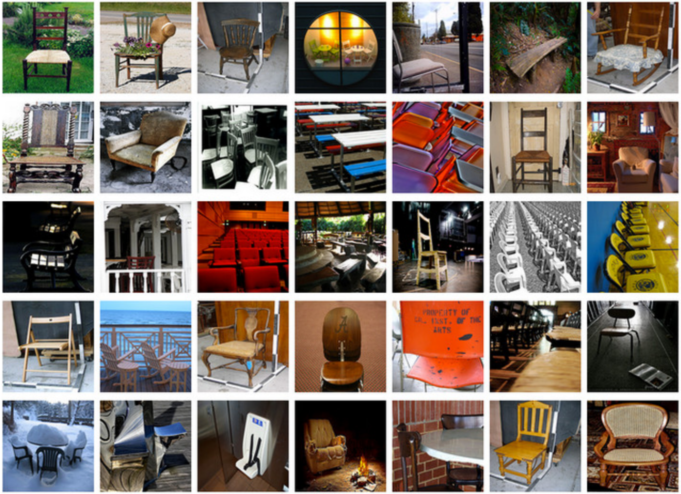
- Cybenko's theorem shows that two-layer networks are omnipotent – such results nearly killed NN when support vector machines were found to be easier to train in 00's.
- Later, it has been shown (experimentally) that deep networks (with many layers) have better represenational properties.
- ... but how to train them? The backpropagation suffers from so-called vanishing gradient, intuitively, updates of weights in lower layers are *very* slow.
- In 2006 a solution was found by Hinton et al:
  - Use *unsupervised* methods to initialize the weights so that they capture important features in data.

    More precisely: The lowest hidden layer learns patterns in data, second lowest learns patterns in data transformed through the first layer, and so on.
  - Then use a supervised learning algorithm to only *fine tune* the weights to the desired input-output behavior.

A rather heavy machinery is needed to develop this, but you will be rewarded by insight into a *very* modern and expensive technology.

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

ImageNet database (16,000,000 color images, 20,000 categories)



ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

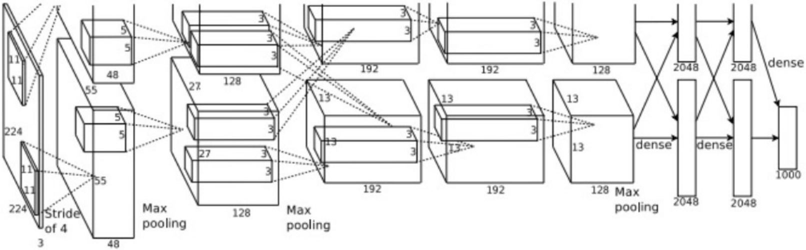
Competition in classification over a subset of images from ImageNet.

In 2012: Training se 1,200,000 images, 1000 categories. Validation set 50,000, Test set 150,000.

Many images contain several objects → typical rule is top-5 highest probability assigned by the net.

KSH síť

ImageNet classification with deep convolutional neural networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton (2012).



Trained on two GPUs (NVIDIA GeForce GTX 580)

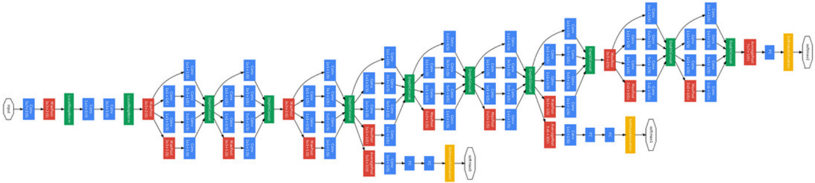
Results:

- Accuracy 84.7% in top-5 (second best alg. at the time: 73.8%)
- 63.3% in "perfect" classification (top-1)

ILSVRC 2014

The same set of images as in 2012, top-5 criterium.

GoogLeNet: deep convolutional net, 22 layers



Results:

- 93.33% in top-5

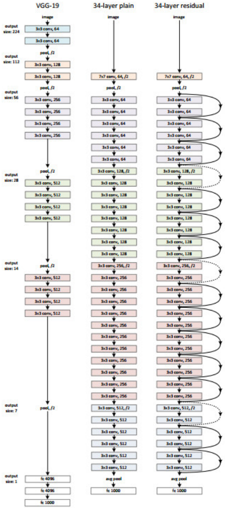
Superhuman power?

Superhuman GoogLeNet?!

Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought we could recruit paid undergrads. Then I organized a labeling party of intense labeling effort only among the (expert labelers) in our lab. Then I developed a modified interface that used GoogLeNet predictions to prune the number of categories from 1000 to only about 100. It was still too hard - people kept missing categories and getting up to ranges of 13-15% error rates. In the end I realized that to get anywhere competitively close to GoogLeNet, it was most efficient if I sat down and went through the painfully long training process and the subsequent careful annotation process myself... The labeling happened at a rate of about 1 per minute, but this decreased over time... Some images are easily recognized, while some images (such as those of fine-grained breeds of dogs, birds, or monkeys) can require multiple minutes of concentrated effort. I became very good at identifying breeds of dogs... Based on the sample of images I worked on, the GoogLeNet classification error turned out to be 6.8%... My own error in the end turned out to be 5.1%, approximately 1.7% better.

ILSVRC 2015

- Microsoft network ResNet: 152 layers, complex architecture
- Trained on 8 GPUs
- 96.43% **accuracy** in top-5



ILSVRC

ilsvrc.png

Deeper Insight into the Logistic Sigmoid

Consider a perceptron (that is a linear classifier):

$$\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

and $y = sgn(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$

Recall, that the *signed distance* from the decision boundary determined by $\xi = 0$ is (here $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{w} = (w_1, \ldots, w_n)$)

$$\frac{w_0 + \sum_{i=1}^{n} w_i \cdot x_i}{\sqrt{\sum_{i=1}^{n} w_i^2}} = \frac{\xi}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

This value is positive for $\vec{x}$ on the side of $\vec{w}$ and negative on the opposite.

For simplicity, assume that $\sqrt{\sum_{i=1}^{n} w_i^2} = 1$, and thus that the potential ξ is *equal to the signed distance of $\vec{x}$ from the boundary*.

Deeper Insight into the Logistic Sigmoid

Assume that training examples $(\vec{x}, c(\vec{x}))$ are randomly generated.

Denote:

- ξ^1 mean signed distance from the boundary of points classified 1.
- ξ^0 mean signed distance from the boundary of points classified 0.

It is not unreasonable to assume that

- conditioned on $c = 1$, the signed distance ξ is normally distributed with the mean ξ^1 and variance (for simplicity) 1,
- conditioned on $c = 0$, the signed distance ξ is normally distributed with the mean ξ^0 and variance (for simplicity) 1.

(Notice that ξ may be negative, which means that such point is on the wrong side of the boundary (the same for $\xi > 0$).)

Now, can we decide what is the probability of $c = 1$ given a distance?

Deeper Insight into the Logistic Sigmoid

For simplicity, assume that $\xi^1 = -\xi^0 = 1/2$.

$$\begin{aligned} P(1 \mid \xi) &= \frac{p(\xi \mid 1)P(1)}{p(\xi \mid 1)P(1) + p(\xi \mid 0)P(0)} \\ &= \frac{LR}{LR + 1/clr} \end{aligned}$$

where

$$LR = \frac{p(\xi \mid 1)}{p(\xi \mid 0)} = \frac{\exp(-(\xi - 1/2)^2/2)}{\exp(-(\xi + 1/2)^2/2)} = \exp(\xi)$$

and

$$clr = \frac{P(1)}{P(0)} \text{ which we assume (for simplicity) } = 1$$

So

$$P(1 \mid \xi) = \frac{\exp(\xi)}{\exp(\xi) + 1} = \frac{1}{1 + e^{-\xi}}$$

Thus the logistic sigmoid applied to $\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$ gives *the probability* of $c = 1$ given the input!

Deeper Insight into the Logistic Sigmoid

So if we use the logistic sigmoid as an activation function,

and turn the neuron into a classifier as follows:

classify a given input $\vec{x}$ as 1 iff $y \geq 1/2$

Then the neuron basically works as the *Bayes classifier!*

This is the basis of logistic regression.

Given training data, we may compute the weights $\vec{w}$ that maximize the likelihood of the training data (w.r.t. the probabilities returned by the neuron).

An extremely interesting observation is that such $\vec{w}$ maximizing the likelihood coincides with the minimum of least squares for the corresponding linear function (that is the same neuron but with identity as the activation function).

Kernel Methods & SVM

Partially based on the ML lecture by Raymond J. Mooney
University of Texas at Austin

Back to Linear Classifier (Slightly Modified)

A linear classifier $h[\vec{w}]$ is determined by a vector of weights $\vec{w} = (w_0, w_1, \ldots, w_n) \in \mathbb{R}^{n+1}$ as follows:

Given $\vec{x} = (x_1, \ldots, x_n) \in X \subseteq \mathbb{R}^n$,

$$h[\vec{w}](\vec{x}) := \begin{cases} 1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i \geq 0 \\ -1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i < 0 \end{cases}$$

For convenience, we use values $\{-1, 1\}$ instead of $\{0, 1\}$. Note that this is not a principal modification, the linear classifier works exactly as the original one.

Recall that given $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, the *augmented feature vector* is

$$\tilde{\mathbf{x}} = (x_0, x_1, \ldots, x_n) \quad \text{where } x_0 = 1$$

This makes the notation for the linear classifier more succinct:

$$h[\vec{w}](\vec{x}) = sig(\vec{w} \cdot \tilde{\mathbf{x}}) \text{ where } sig(y) = \begin{cases} 1 & y \geq 0 \\ -1 & y < 0 \end{cases}$$

Perceptron Learning Revisited

- Given a training set

  $$D = \{(\vec{x}_1, y(\vec{x}_1)), (\vec{x}_2, y(\vec{x}_2)), \ldots, (\vec{x}_p, y(\vec{x}_p))\}$$

  Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in X \subseteq \mathbb{R}^n$ and $y(\vec{x}_k) \in \{-1, 1\}$.

  We write y_k instead of $y(\vec{x}_k)$.
  Note that $\tilde{\mathbf{x}}_k = (x_{k0}, x_{k1} \ldots, x_{kn})$ where $x_{k0} = 1$.

- A weight vector $\vec{w} \in \mathbb{R}^{n+1}$ is **consistent with** D if

  $$h[\vec{w}](\vec{x}_k) = sig(\vec{w} \cdot \tilde{\mathbf{x}}_k) = y_k \quad \text{for all } k = 1, \ldots, p$$

  D is **linearly separable** if there is a vector $\vec{w} \in \mathbb{R}^{n+1}$ which is consistent with D.

Perceptron Learning Revisited

Perceptron learning algorithm (slightly modified):

Consider training examples cyclically. Compute a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- $\vec{w}^{(0)}$ is initialized to $\vec{0} = (0, \ldots, 0)$.

  (This is a slight but harmless modification of the traditional algorithm.)

- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:
  - If $sig(\vec{w} \cdot \tilde{\mathbf{x}}_k) \neq y_k$, then $\vec{w}^{(t+1)} = \vec{w}^{(t)} + y_k \cdot \tilde{\mathbf{x}}_k$.
  - Otherwise, $\vec{w}^{(t+1)} = \vec{w}^{(t)}$.

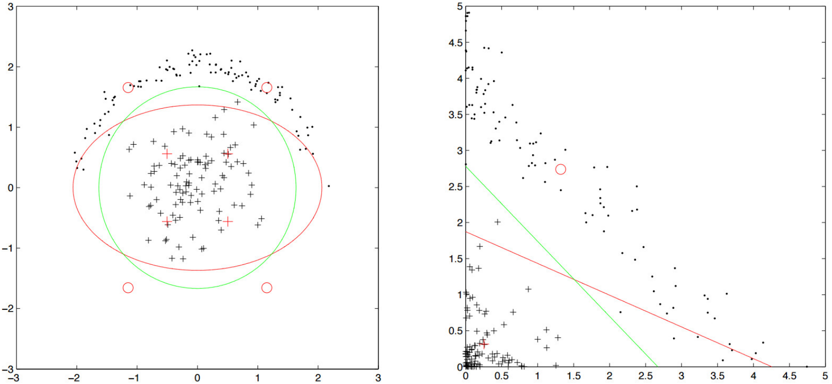
  Here $k = (t \mod p) + 1$, i.e. the examples are considered cyclically.

(Note that this algorithm corresponds to the perceptron learning with the learning speed $\varepsilon = 1$.)

We know: if D is linearly separable, then there is t^* such that $\vec{w}^{(t^*)}$ is consistent with D.

But what can we do if D is not linearly separable?

Quadratic Decision Boundary



Left: The original set, Right: Transformed using the square of features.

Right: the green line is the decision boundary learned using the perceptron algorithm.

(The red boundary corresponds to another learning algorithm.)

Left: the green ellipse maps exactly to the green line.

How to classify (in the original space): First, transform a given feature vector by squaring the features, then use the linear classifier.

Do We Need to Map Explicitly?

In general, mapping to (much) higher feature space helps (there are more "degrees of freedom" so linear separability might get a chance).

However, complexity of learning grows (quickly) with dimension.

Sometimes its even beneficial to map to infinite-dimensional spaces.

To avoid explicit construction of the higher dimensional feature space, we use so called *kernel trick*.

But first we need to *dualize* our learning algorithm.

Perceptron Learning Revisited

Perceptron learning algorithm once more:

Compute a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- $\vec{w}^{(0)}$ is initialized to $\vec{0} = (0, \ldots, 0)$.
- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:
  - If $sig(\vec{w} \cdot \tilde{\mathbf{x}}_k) \neq y_k$, then $\vec{w}^{(t+1)} = \vec{w}^{(t)} + y_k \cdot \tilde{\mathbf{x}}_k$.
  - Otherwise, $\vec{w}^{(t+1)} = \vec{w}^{(t)}$.

  Here $k = (t \mod p) + 1$, i.e. the examples are considered cyclically.

Crucial observation:

Note that $\vec{w}^{(t)} = \sum_{\ell=1}^{p} n_\ell^{(t)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell$ for suitable $n_1^{(t)}, \ldots, n_p^{(t)} \in \mathbb{N}$.

Intuitively, $n_\ell^{(t)}$ counts how many times $\tilde{\mathbf{x}}_\ell$ was added to (if $y_\ell = 1$), or subtracted from (if $y_\ell = -1$) weights.

Dual Perceptron Learning

Dual Perceptron learning algorithm :

Compute a sequence of vectors of numbers $\vec{n}^{(0)}, \vec{n}^{(1)}, \ldots$ where each $\vec{n}^{(t)} = (n_1^{(t)}, \ldots, n_p^{(t)}) \in \mathbb{N}^p$.

- $\vec{n}^{(0)}$ is initialized to $\vec{0} = (0, \ldots, 0)$.
- In $(t+1)$-th step, $(n_1^{(t+1)}, \ldots, n_p^{(t+1)})$ is computed as follows:
    - If $sig(\sum_{\ell=1}^{p} n_\ell^{(t)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_k) \neq y_k$, then $n_k^{(t+1)} := n_k^{(t)} + 1$, else, $n_k^{(t+1)} := n_k^{(t)}$.
    - $n_\ell^{(t+1)} := n_\ell^{(t)}$ for all $\ell \neq k$.

  Here $k = (t \mod p) + 1$, the examples are considered cyclically.

If D is linearly separable, there exists t^* such that $\sum_{\ell=1}^{p} n_\ell^{(t^*)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell$ is consistent with D. The algorithm stops at such t^* and returns $(n_1^{(t^*)}, \ldots, n_p^{(t^*)})$ so that $\sum_{\ell=1}^{p} n_\ell^{(t^*)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell$ is consistent with D.

Example

Training set:

$$D = \{((2,-1),1),((2,1),1),((1,3),-1)\}$$

That is

$$\begin{aligned} \vec{x}_1 &= (2,-1) & \qquad \tilde{\mathbf{x}}_1 &= (1,2,-1) \\ \vec{x}_2 &= (2,1) & \qquad \tilde{\mathbf{x}}_2 &= (1,2,1) \\ \vec{x}_3 &= (1,3) & \qquad \tilde{\mathbf{x}}_3 &= (1,1,3) \end{aligned}$$

$$\begin{aligned} y_1 &= 1 \\ y_2 &= 1 \\ y_3 &= -1 \end{aligned}$$

The initial values $n_1^{(0)} = n_2^{(0)} = n_3^{(0)} = 0$.

- $\sum_{\ell=1}^{3} n_\ell^{(0)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_1 = 0$, thus $sig(\sum_{\ell=1}^{3} n_\ell^{(0)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_1) = 1 = y_1$. Hence, $\vec{n}^{(1)} = (0,0,0)$.
- $\sum_{\ell=1}^{3} n_\ell^{(1)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_2 = 0$, thus $sig(\sum_{\ell=1}^{3} n_\ell^{(1)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_2) = 1 = y_2$. Hence, $\vec{n}^{(2)} = (0,0,0)$.
- $\sum_{\ell=1}^{3} n_\ell^{(2)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_3 = 0$, thus $sig(\sum_{\ell=1}^{3} n_\ell^{(2)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_3) = 1 \neq y_3$. Hence, $\vec{n}^{(3)} = (0,0,1)$.
- $\sum_{\ell=1}^{3} n_\ell^{(3)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_1 = -1 \cdot \tilde{\mathbf{x}}_3 \cdot \tilde{\mathbf{x}}_1 = -1 \cdot (1,1,3) \cdot (1,2,-1) = -1 \cdot 0 = 0$, thus $sig(\sum_{\ell=1}^{3} n_\ell^{(3)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_1) = 1 = y_1$. Hence, $\vec{n}^{(4)} = (0,0,1)$.
- $\sum_{\ell=1}^{3} n_\ell^{(4)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_2 = -1 \cdot \tilde{\mathbf{x}}_3 \cdot \tilde{\mathbf{x}}_2 = -1 \cdot (1,1,3) \cdot (1,2,1) = -1 \cdot 6 = -6$, thus $sig(\sum_{\ell=1}^{p} n_\ell^{(4)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_2) = -1 \neq y_2$. Hence, $\vec{n}^{(5)} = (0,1,1)$.
- $\sum_{\ell=1}^{p} n_\ell^{(5)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_3 = 1 \cdot \tilde{\mathbf{x}}_2 \cdot \tilde{\mathbf{x}}_3 - 1 \cdot \tilde{\mathbf{x}}_3 \cdot \tilde{\mathbf{x}}_3 = -5$, thus $\vec{n}^{(6)} = (0,1,1)$. This is OK.
- $\sum_{\ell=1}^{p} n_\ell^{(6)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_1 = 1 \cdot \tilde{\mathbf{x}}_2 \cdot \tilde{\mathbf{x}}_1 - 1 \cdot \tilde{\mathbf{x}}_3 \cdot \tilde{\mathbf{x}}_1 = 4$, thus $\vec{n}^{(7)} = (0,1,1)$. This is OK.
- $\sum_{\ell=1}^{p} n_\ell^{(6)} \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}_2 = 1 \cdot \tilde{\mathbf{x}}_2 \cdot \tilde{\mathbf{x}}_2 - 1 \cdot \tilde{\mathbf{x}}_3 \cdot \tilde{\mathbf{x}}_2 = 0$, thus $\vec{n}^{(7)} = (0,1,1)$. This is OK.

The result: $\tilde{\mathbf{x}}_2 - \tilde{\mathbf{x}}_3$.

Dual Perceptron Learning – Output

Let $\sum_{\ell=1}^{p} n_\ell \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell$ result from the dual perceptron learning algorithm.

I.e., each $n_\ell = n_\ell^{(t^*)} \in \mathbb{N}$ for suitable t^* in which the algorithm found a consistent vector.

This vector of weights determines a linear classifier that for a given $\vec{x} \in \mathbb{R}^n$ gives

$$h[\vec{w}](\vec{x}) = sig\left(\sum_{\ell=1}^{p} n_\ell \cdot y_\ell \cdot \tilde{\mathbf{x}}_\ell \cdot \tilde{\mathbf{x}}\right)$$

(Here $\tilde{\mathbf{x}}$ is the augmented feature vector obtained from $\vec{x}$.)

Crucial observation: The (augmented) feature vectors $\tilde{\mathbf{x}}_\ell$ and $\tilde{\mathbf{x}}$ occur *only* in scalar products!

Kernel Trick

For simplicity, assume bivariate data: $\tilde{\mathbf{x}}_k = (1, x_{k1}, x_{k2})$.

The *corresponding instance* in the quadratic feature space is $(1, x_{k1}^2, x_{k2}^2)$.

Consider two instances $\tilde{\mathbf{x}}_k = (1, x_{k1}, x_{k2})$ and $\tilde{\mathbf{x}}_\ell = (1, x_{\ell 1}, x_{\ell 2})$. Then the scalar product of their corresponding instances $(1, x_{k1}^2, x_{k2}^2)$ and $(1, x_{\ell 1}^2, x_{\ell 2}^2)$, resp., in the quadratic feature space is

$$1 + x_{k1}^2 x_{\ell 1}^2 + x_{k2}^2 x_{\ell 2}^2$$

which resembles (but is not equal to)

$$(\tilde{\mathbf{x}}_k \cdot \tilde{\mathbf{x}}_\ell)^2 = (1 + x_{k1} x_{\ell 1} + x_{k2} x_{\ell 2})^2 =$$
$$= 1 + x_{k1}^2 x_{\ell 1}^2 + x_{k2}^2 x_{\ell 2}^2 + 2 x_{k1} x_{\ell 1} x_{k2} x_{\ell 2} + 2 x_{k1} x_{\ell 1} + 2 x_{k2} x_{\ell 2}$$

But now consider a mapping ϕ to $\mathbb{R}^6$ defined by

$$\phi(\tilde{\mathbf{x}}_k) = (1, x_{k1}^2, x_{k2}^2, \sqrt{2} x_{k1} x_{k2}, \sqrt{2} x_{k1}, \sqrt{2} x_{k2})$$

Then

$$\phi(\tilde{\mathbf{x}}_k) \cdot \phi(\tilde{\mathbf{x}}_\ell) = (\tilde{\mathbf{x}}_k \cdot \tilde{\mathbf{x}}_\ell)^2$$

THE Idea: Define a *kernel* $\kappa(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_\ell) = (\tilde{\mathbf{x}}_k \cdot \tilde{\mathbf{x}}_\ell)^2$ and replace $\tilde{\mathbf{x}}_k \cdot \tilde{\mathbf{x}}_\ell$ in the dual perceptron algorithm with $\kappa(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_\ell)$.

Kernel Perceptron Learning

Kernel Perceptron learning algorithm :

Compute a sequence of vectors of numbers $\vec{n}^{(0)}, \vec{n}^{(1)}, \ldots$ where each $\vec{n}^{(t)} = (n_1^{(t)}, \ldots, n_p^{(t)}) \in \mathbb{N}^p$.

- $\vec{n}^{(0)}$ is initialized to $\vec{0} = (0, \ldots, 0)$.
- In $(t+1)$-th step, $(n_1^{(t+1)}, \ldots, n_p^{(t+1)})$ is computed as follows:
  - If $sig\left(\sum_{\ell=1}^{p} n_\ell^{(t)} \cdot y_\ell \cdot \kappa(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_\ell)\right) \neq y_k$, then $n_k^{(t+1)} := n_k^{(t)} + 1$,

    else, $n_k^{(t+1)} := n_k^{(t)}$.
  - $n_\ell^{(t+1)} := n_\ell^{(t)}$ for all $\ell \neq k$.

  Here $k = (t \mod p) + 1$, the examples are considered cyclically.

Intuition: The algorithm computes a linear classifier in $\mathbb{R}^6$ for training examples transformed using ϕ.

The trick is that the transformation ϕ itself *does not have to be explicitly computed!*

Dual Perceptron Learning

Let $\vec{n} = (n_1, \ldots, n_p)$ result from the kernel perceptron learning algorithm.
I.e., each $n_\ell = n_\ell^{(t^*)} \in \mathbb{N}$ for suitable t^* such that
$sig\left(\sum_{\ell=1}^{p} n_\ell^{(t^*)} \cdot y_\ell \cdot \kappa(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_\ell)\right) = y_k$ for all $k = 1, \ldots, p$.

We obtain a *non-linear classifier* that for a given $\vec{x} \in \mathbb{R}^n$ gives

$$h[\vec{w}](\vec{x}) = sig\left(\sum_{\ell=1}^{p} n_\ell \cdot y_\ell \cdot \kappa(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}_\ell)\right)$$

(Here $\tilde{\mathbf{x}}$ is the augmented feature vector obtained from $\vec{x}$.)

Are there other kernels that correspond to the scalar product in higher dimensional spaces?

Kernels

Given a (potential) kernel $\kappa(\vec{x}_\ell, \vec{x}_k)$ we need to check whether $\kappa(\vec{x}_\ell, \vec{x}_k) = \phi(\vec{x}_\ell) \cdot \phi(\vec{x}_k)$ for a function ϕ. This might be very difficult.

Věta (Mercer's)

κ is a kernel if the corresponding Gram matrix K of the training set D, whose each ℓk-th element is $\kappa(\vec{x}_\ell, \vec{x}_k)$, is positive semi-definite for all possible choices of the training set D.

Kernels can be constructed from existing kernels by several operations

- linear combination (i.e. multiply by a constant, or sum),
- multiplication,
- exponentiation,
- multiply by a polynomial with non-negative coefficients,
- ...

(see e.g. "Pattern Recognition and Machine Learning" by Bishop)

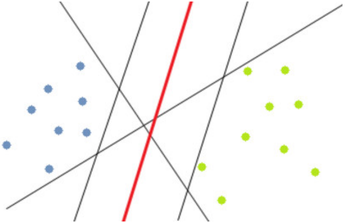
Examples of Kernels

- Linear: $\kappa(\vec{x}_\ell, \vec{x}_k) = \vec{x}_\ell \cdot \vec{x}_k$
  The corresponding mapping $\phi(\vec{x}) = \vec{x}$ is identity (no transformation).
- Polynomial of power m: $\kappa(\vec{x}_\ell, \vec{x}_k) = (1 + \vec{x}_\ell \cdot \vec{x}_k)^m$
  The corresponding mapping assigns to $\vec{x} \in \mathbb{R}^n$ the vector $\phi(\vec{x})$ in $\mathbb{R}^{\binom{n+m}{m}}$.
- Gaussian (radial-basis function): $\kappa(\vec{x}_\ell, \vec{x}_k) = e^{-\frac{\|\vec{x}_\ell - \vec{x}_k\|^2}{2\sigma^2}}$
  The corresponding mapping ϕ maps $\vec{x}$ to an *infinite-dimensional* vector $\phi(\vec{x})$ which is, in fact, a Gaussian function; combination of such functions for support vectors is then the separating hypersurface.
- ...

Choosing kernels remains to be black magic of kernel methods. They are usually chosen based on trial and error (of course, experience and additional insight into data helps).

Now let's go on to the main area where kernel methods are used: to enhance support vector machines.

SVM Idea – Which Linear Classifier is the Best?



Benefits of maximum margin:

- Intuitively, maximum margin is good w.r.t. generalization.
- Only the *support vectors* (those on the magin) matter, others can, in principle, be ignored.

Support Vector Machines (SVM)

Notation:

- $\vec{w} = (w_0, w_1, \ldots, w_n)$ a vector of weights,
- $\underline{\vec{w}} = (w_1, \ldots, w_n)$ a vector of all weights except w_0,
- $\vec{x} = (x_1, \ldots, x_n)$ a (generic) feature vector.

Consider a linear classifier:

$$h[\vec{w}](\vec{x}) := \begin{cases} 1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i = w_0 + \underline{\vec{w}} \cdot \vec{x} \geq 0 \\ -1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i = w_0 + \underline{\vec{w}} \cdot \vec{x} < 0 \end{cases}$$

The *signed distance* of $\vec{x}$ from the decision boundary determined by $\vec{w}$ is

$$d[\vec{w}](\vec{x}) = \frac{w_0 + \underline{\vec{w}} \cdot \vec{x}_k}{\|\underline{\vec{w}}\|}$$

Here $\|\underline{\vec{w}}\| = \sqrt{\sum_{i=1}^{n} w_i^2}$ is the Euclidean norm of $\underline{\vec{w}}$.

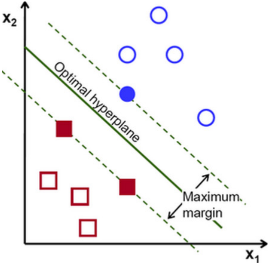
$|d[\vec{w}](\vec{x})|$ is the distance of $\vec{x}$ from the decision boundary.
$d[\vec{w}](\vec{x})$ is positive for $\vec{x}$ on the side to which $\underline{\vec{w}}$ points and negative on the opposite side.

Support Vectors & Margin

- Given a training set
  $$D = \{(\vec{x}_1, y(\vec{x}_1)), (\vec{x}_2, y(\vec{x}_2)), \ldots, (\vec{x}_p, y(\vec{x}_p))\}$$
  Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in X \subseteq \mathbb{R}^n$ and $y(\vec{x}_k) \in \{-1, 1\}$.

  We write y_k instead of $y(\vec{x}_k)$.

- Assume that D is linearly separable, let $\vec{w}$ *be consistent with* D *so that the distance of the decision boundary from the nearest examples on both sides is the same* (if not, it suffices to adjust w_0).



- *Support vectors* are those $\vec{x}_k$ that minimize $|d[\vec{w}](\vec{x}_k)|$.
- *Margin* ρ of $\vec{w}$ is twice the distance between support vectors and the decision boundary.

Our goal is to find a classifier that maximizes the margin.

Maximizing the Margin

For $\vec{w}$ consistent with D (such that no $\vec{x}_k$ lies on the decision boundary) we have

$$\varrho = 2 \cdot \frac{|w_0 + \underline{\vec{w}} \cdot \vec{x}_k|}{\|\underline{\vec{w}}\|} = 2 \cdot \frac{y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k)}{\|\underline{\vec{w}}\|} > 0$$

where $\vec{x}_k$ is a support vector.

We may safely consider only $\vec{w}$ such that $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$ for the support vectors.

Just adjust the length of $\vec{w}$ so that $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$, the denominator $\|\underline{\vec{w}}\|$ will compensate.

Then maximizing ϱ is equivalent to maximizing $2/\|\underline{\vec{w}}\|$.

(In what follows we use a bit looser constraint:

$$y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1 \text{ for all } \vec{x}_k$$

However, the result is the same since even with this looser condition, the support vectors always satisfy $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$ whenever $2/\|\underline{w}\|$ is maximal.)

SVM – Optimization

Margin maximization can be formulated as a *quadratic optimization problem:*

Find $\vec{w} = (w_0, \ldots, w_n)$ such that

$$\rho = \frac{2}{\|\underline{\vec{w}}\|} \text{ is maximized}$$

and for all $(\vec{x}_k, y_k) \in D$ we have $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1.$

which can be reformulated as:

Find $\vec{w}$ such that

$$\Phi(\vec{w}) = \|\underline{\vec{w}}\|^2 = \underline{\vec{w}} \cdot \underline{\vec{w}} \text{ is minimized}$$

and for all $(\vec{x}_k, y_k) \in D$ we have $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1.$

SVM – Optimization

- Need to optimize a quadratic function subject to linear constraints.
- Quadratic optimization problems are a well-known class of mathematical programming problems for which efficient methods (and tools) exist.
- The solution usually involves construction of a *dual problem* where *Lagrange multipliers* α_i are associated with every inequality (constraint) in the original problem:

Find $\alpha = (\alpha_1, \ldots, \alpha_p)$ such that

$$\Psi(\alpha) = \sum_{\ell=1}^{p} \alpha_\ell - \frac{1}{2} \sum_{\ell=1}^{p} \sum_{k=1}^{p} \alpha_\ell \cdot \alpha_k \cdot y_\ell \cdot y_k \cdot \vec{x}_\ell \cdot \vec{x}_k \text{ is maximized}$$

so that the following constraints are satisfied:

- $\sum_{\ell=1}^{p} \alpha_\ell y_\ell = 0$
- $\alpha_\ell \geq 0$ for all $1 \leq \ell \leq p$

The Optimization Problem Solution

- Given a solution $\alpha_1, \ldots, \alpha_n$ to the dual problem, solution $\vec{w} = (w_0, w_1, \ldots, w_n)$ to the original one is:

$$\underline{\vec{w}} = (w_1, \ldots, w_n) = \sum_{\ell=1}^{p} \alpha_\ell \cdot y_\ell \cdot \vec{x}_\ell$$

$$w_0 = y_k - \sum_{\ell=1}^{p} \alpha_\ell \cdot y_\ell \cdot \vec{x}_\ell \cdot \vec{x}_k \text{ for an arbitrary } \alpha_k > 0$$

  Note that $\alpha_k > 0$ iff $\vec{x}_k$ is a support vector. Hence it does not matter which $\alpha_k > 0$ is chosen in the above definition of w_0.

- The classifier is then

$$\boxed{\begin{aligned} h(\vec{x}) &= sig(w_0 + \underline{\vec{w}} \cdot \vec{x}) \\ &= sig\left(y_k - \textstyle\sum_\ell \alpha_\ell \cdot y_\ell \cdot \vec{x}_\ell \cdot \vec{x}_k + \sum_\ell \alpha_\ell \cdot y_\ell \cdot \vec{x}_\ell \cdot \vec{x}\right) \end{aligned}}$$

Note that both the dual optimization problem as well as the classifier contain training feature vectors only in the scalar product! We may apply the kernel trick!

Kernel SVM

- Choose your favourite kernel κ.
- Solve the *dual problem* with kernel κ:

Find $\alpha = (\alpha_1, \ldots, \alpha_p)$ such that

$$\Psi(\alpha) = \sum_{\ell=1}^{p} \alpha_\ell - \frac{1}{2} \sum_{\ell=1}^{p} \sum_{k=1}^{p} \alpha_\ell \cdot \alpha_k \cdot y_\ell \cdot y_k \cdot \kappa(\vec{x}_\ell, \vec{x}_k) \text{ is maximized}$$

so that the following constraints are satisfied:

  - $\sum_\ell \alpha_\ell y_\ell = 0$
  - $\alpha_\ell \geq 0$ for all $1 \leq \ell \leq p$

- Then use the classifier:

$$h(\vec{x}) = sig\left(y_k - \sum_\ell \alpha_\ell \cdot y_\ell \cdot \kappa(\vec{x}_\ell, \vec{x}_k) + \sum_\ell \alpha_\ell \cdot y_\ell \cdot \kappa(\vec{x}_\ell, \vec{x})\right)$$

- Note that the optimization techniques remain the same as for the linear SVM without kernels!

Comments on Algorithms

- The main bottleneck of SVM's is in complexity of quadratic programming (QP). A naive QP solver has cubic complexity.
- For small problems any general purpose optimization algorithm can be used.
- For large problems this is usually not possible, many methods avoiding direct solution have been devised.
- These methods usually decompose the optimization problem into a sequence of smaller ones. Intuitively,
  - start with a (smaller) subset of training examples.
  - Find an optimal solution using any solver.
  - Afterwards, only support vectors matter in the solution! Leave only them in the training set, and add new training examples.
  - This iterative procedure decreases the (general) cost function.

SVM in Applications (Mooney's lecture)

- SVMs were originally proposed by Boser, Guyon and Vapnik in 1992 and gained increasing popularity in late 1990s.
- SVMs are currently among the best performers for a number of classification tasks ranging from text to genomic data.
- SVMs can be applied to complex data types beyond feature vectors (e.g. graphs, sequences, relational data) by designing kernel functions for such data.
- SVM techniques have been extended to a number of tasks such as regression [Vapnik et al. '97], principal component analysis [Schölkopf et al. '99], etc.
- Most popular optimization algorithms for SVMs use decomposition to hillclimb over a subset of α_i's at a time, e.g. SMO [Platt '99] and [Joachims '99]
- Tuning SVMs remains a black art: selecting a specific kernel and parameters is usually done in a try-and-see manner.