# MACHINE LEARNING

*Program computers such that they improve their performance in solving a problem from experience.*

Discover rules, patterns, connections.

- **Data mining**
  Database $\longrightarrow$ knowledge
  Ex.: medicine, bioinformatics

- **Classification / prediction**
  Medical diagnosis
  Detecting defect products
  Prediction of genes, protein structure, . . .

- **Image / speech recognition**
  Objects, faces, fingerprints, handwriting, natural speech

# The Learning Problem

A program **learns** if it improves its ability

- in performing task $T$
- measured by performance measure $P$
- basee on experience $E$

# Examples

**Chess**

- $T$ play chess
- $P$ % games won in tournament
- $E$ playing against itself

**Speech regonition**

- $T$ transform speech to text
- $P$ % correct words
- $E$ training examples with answers

**Gene prediction**

- $T$ find genes in DNA
- $P$ % correctly positioned
- $E$ trainging examples (known genes)

**Protein structure prediction**

- $T$ predict secondary structure given the amino acid sequence
- $P$ $\alpha$-helices and $\beta$-strands correctly positioned
- $E$ training examples with known structure

# Design of a learning system

- $T$: Play chess
- $P$: Percentage of games won in tournament

1. What kind of experience $E$?
2. *What* should we learn? (target function)
3. How to represent the target function?
4. Which mechanism to learn it?

# 1. Experience / training

## Direct/indirect feedback

- **Direct:** Feedback on move: Good/bad
- **Indirect:** Must wait for game outcome
  Problem: distributing blame/credit on moves

## Choice of training examples

With our without teacher

- Teacher chooses examples
- Learner asks teacher for correct answer
- Learner both asks and answers (plays against itself)
  - same game all over, random games
  - varying games/moves by choice

## Agreement between training and real data

Very important in all machine learning applications!

## Chess:

- **Training:** playing against itself
- **Test/use:** playing against world champion

# 2. Choosing the target function

---

$$\begin{aligned} B &= \text{set of all possible board states} \\ M(b) &= \text{set of all legal moves} \\ &\quad \text{for board position } b \in B \end{aligned}$$

Want the **best** move for given $b \in B$.

Natural target function:

$$ChooseMove : B \rightarrow M(b)$$

Difficult to learn!

Simpler: Score function

$$V : B \rightarrow \mathbb{R}$$

Then choose the move giving the highest score.

# An ideal target function

If $b$ is a final board state:

$$V(b) = \begin{cases} 100 & \text{if game is won} \\ -100 & \text{if game is lost} \\ 0 & \text{if game is drawn} \end{cases}$$

If $b$ is not a final state:

$$V(b) = V(b')$$

where $b'$ is the best final state that can be attained if both players play optimally from state $b$.

Theoretically: Can be computed
(adversial searching: minimax / alpha-beta algorithm)

In practise: Not an operational definition!

# Operational target function

- Need an *operational* description of $V$ : $B \to \mathbb{R}$ which the program can evaluate in reasonable time.

- Difficult to learn $V$ exactly.
  Learn *approximation* $\widehat{V}$.

- Need a *representation* of $\widehat{V}$.

# 3. Representation of target function

Some types of representations:

- table of values for all $b \in B$ (not realistic!)
- set of rules (e.g., decision tree)
- artificial neural network
- function $\widehat{V}(x_1, x_2, \ldots, x_k)$ of selected quantities $x_1, x_2, \ldots, x_k$, with parameters that can be learned.

Example of the last type: linear function

$$\widehat{V}(x_1, x_2, \ldots, x_k) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_k x_k$$

# Possible target function for chess

$\hat{V} = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$

der

$x_1$ = points for white pieces
$x_2$ = points for black pieces
$x_3$ = points for black pieces threatened by white
$x_4$ = points for white pieces threatened by black
$x_5$ = number of squares controlled by white
$x_6$ = number of squares controlled by black

$w_1, w_2, w_3, w_4, w_5, w_6$: weights to be learned
$w_0$: additive constant to be learned
(Maybe add symmetry contraints?)

# 4. Algorithm to learn $\hat{V}$

Need training examples $\langle b, V_{\text{train}}(b)\rangle$.

Example (white wins!):

$$\langle\langle 9,0,0,0,28,8\rangle, +100\rangle$$

Must:

- find such pairs from indirect experience (more difficult when $b$ is not a final position!)

- adjust weights to adapt $\hat{V}$ to the examples

# Estimation of training values

$V(b)$: true target function (ukjent)
$\hat{V}(b)$: learned target function
$V_{\text{train}}(b)$: values given in training examples

Self training: how to estimate $V_{\text{train}}(b)$?

Practical possibility:

$$V_{\text{train}}(b) := \hat{V}(\text{next}(b))$$

where next($b$) is the board state following $b$ after the player's and opponent's move in the training game.

# Adjusting weights

In general: updating the learned target function
Here: adjusting the weights $w_0, \ldots, w_6$.

Want to minimize the square error (LMS)

$$E = \sum_{\text{training examples}} \left(V_{\text{train}}(b) - \hat{V}(b)\right)^2$$

**LMS weight update rule:**

For each pair $\langle b, V_{\text{train}}(b)\rangle$:

1. Use current weights to compute $\hat{V}(b)$.

2. For each weight $w_i$:

   $$w_i := w_i + \eta(V_{\text{train}}(b) - \hat{V}(b))x_i$$

   (where $x_0 = 1$)

$\eta$: "small" constant (e.g. 0.1)