Feedforward Multi-Layer Perceptron (MLP) and error backpropagation (Dopredná viacvrstvová sieť a spätné šírenie chýb)

Lubica Benuskova

AIMA 3rd ed. Ch. 18.6.4 – 18.7.5

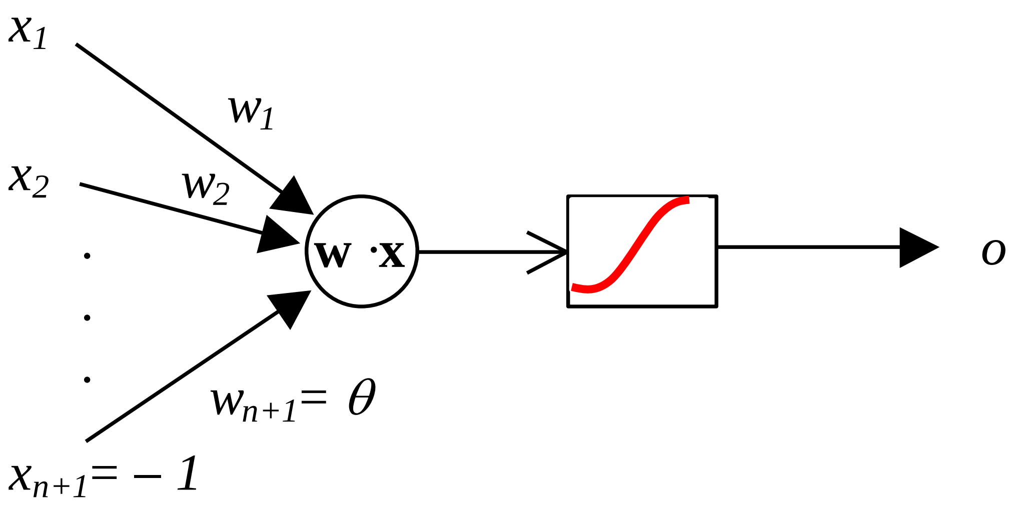
Some historical notes

- *Paul Werbos* : Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science, Ph.D. thesis, Harvard University, 1974.

- *Rumelhart, Hinton, Williams*: Learning internal representations by backpropagating errors, Nature 323(99), pp. 533-536,1986.
  - **Rumelhart Prize** is awarded annually for a significant contribution to the theoretical foundations of human cognition (US$ 100,000).

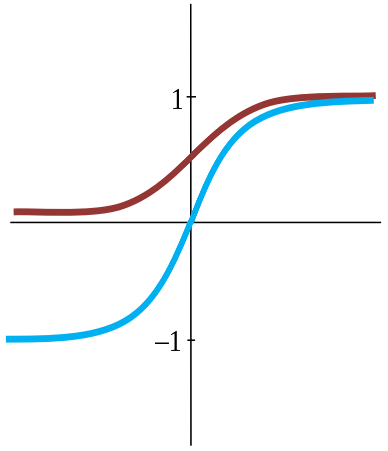
- In 1989, mathematical proof of the theorem of universal approximation of functions by MLP independently by *Kurt Hornik* and *Věra Kůrková*.

Continuous perceptron (sigmoid, tanh)



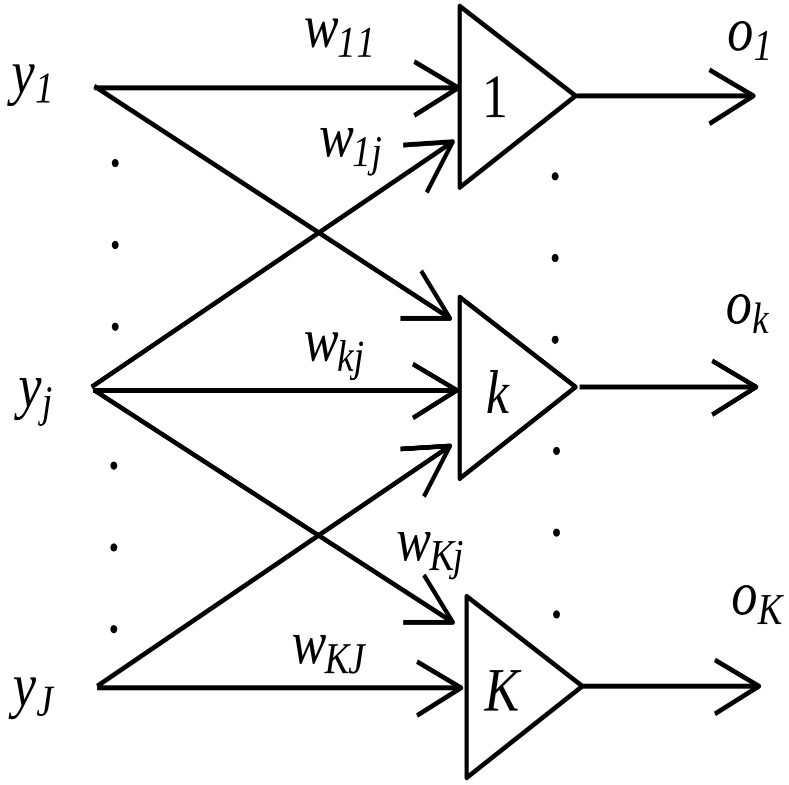
$$o = f(net) = f(\mathbf{w}\cdot\mathbf{x}) = f\left(\sum_{j=1}^{n+1} w_j x_j\right) = f\left(\sum_{j=1}^{n} w_j x_j - \theta\right)$$

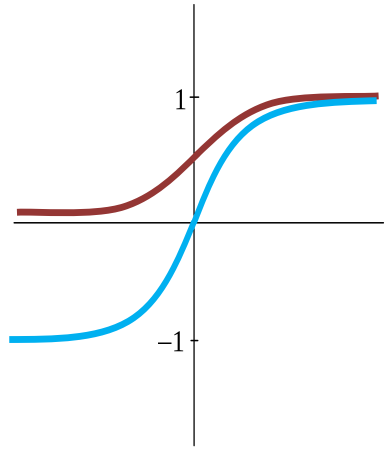
$$f(net) = \frac{1}{1+\exp(-net)}$$



$$f(net) = \tanh(net)$$

Feed-forward perceptron with a single layer of several neurons





$$o_k = f_k(net_k)$$

$$net_k = \sum_{j=1}^{J} w_{kj} y_j$$

Bias: $y_J = -1$ and $w_{kJ} = \theta_k$

Towards δ rule for a single output layer

- Let the training set be

$$A_{train} = \{(\mathbf{y}^1, \mathbf{d}^1)(\mathbf{y}^2, \mathbf{d}^2)...(\mathbf{y}^p, \mathbf{d}^p)....(\mathbf{y}^P, \mathbf{d}^P)\}$$

  - $\mathbf{y}^p$ is the input vector *p* (pattern *p*)
  - $\mathbf{d} = (d_1, d_2, ..., d_k, ..., d_K)$ is the desired output vector

- Error function per pattern *p* = sum of squared errors over *k*

$$E^p = \tfrac{1}{2}\sum_{k=1}^{K}(d_k^p - o_k^p)^2 = \tfrac{1}{2}\sum_{k=1}^{K}(d_k^p - f_k(net_k))^2$$

  - Where $o^p{}_k$ is the actual output of unit *k* for pattern *p*

Gradient descent for an output layer

- After presentation of each pattern we want to minimise E^p

$$\Delta w_{kj} = -\alpha \frac{\partial E^p}{\partial w_{kj}} = -\alpha \frac{\partial E^p}{\partial (net_k)} \frac{\partial (net_k)}{\partial w_{kj}} = \alpha\, \delta_{ok}\, y_j$$

- Here and later we use the chain rule for the derivative of the nested function $f(g(h(x)))$, i.e.: $\dfrac{df}{dx} = \dfrac{df}{dg}\dfrac{dg}{dh}\dfrac{dh}{dx}$

- Let us continue with the derivative

$$\frac{\partial (net_k)}{\partial w_{kj}} = \frac{\partial \left(\sum_{j=1}^{J} w_{kj} y_j \right)}{\partial w_{kj}} = y_j$$

Generalised δ rule for a single output layer

- The negative derivative of error E^p according to the net_k is the generalised error signal δ_{ok} produced by the (output) neuron k

$$\delta_{ok} = -\frac{\partial E^p}{\partial(net_k)} = -\frac{\partial E^p}{\partial o_k}\frac{\partial o_k}{\partial(net_k)} = (d_k^p - o_k^p) f_k'$$

$$-\frac{\partial E^p}{\partial o_k} = -\frac{\partial\left(\frac{1}{2}\sum_{k=1}^{K}(d_k^p - o_k^p)^2\right)}{\partial o_k} = (d_k^p - o_k^p)$$

$$\frac{\partial o_k}{\partial(net_k)} = \frac{\partial\left(f_k(net_k)\right)}{\partial(net_k)} = f_k'$$

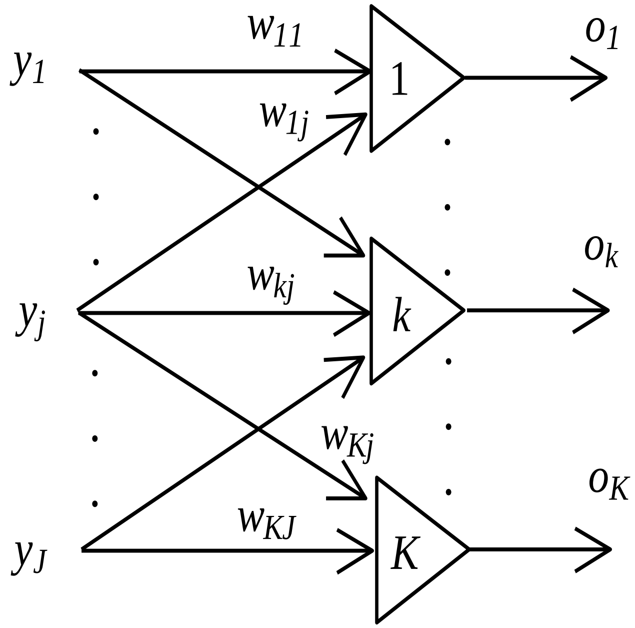
Generalised δ rule for a single output layer

- After presentation of each pattern we adjust the weights of all output neurons to minimise E_p

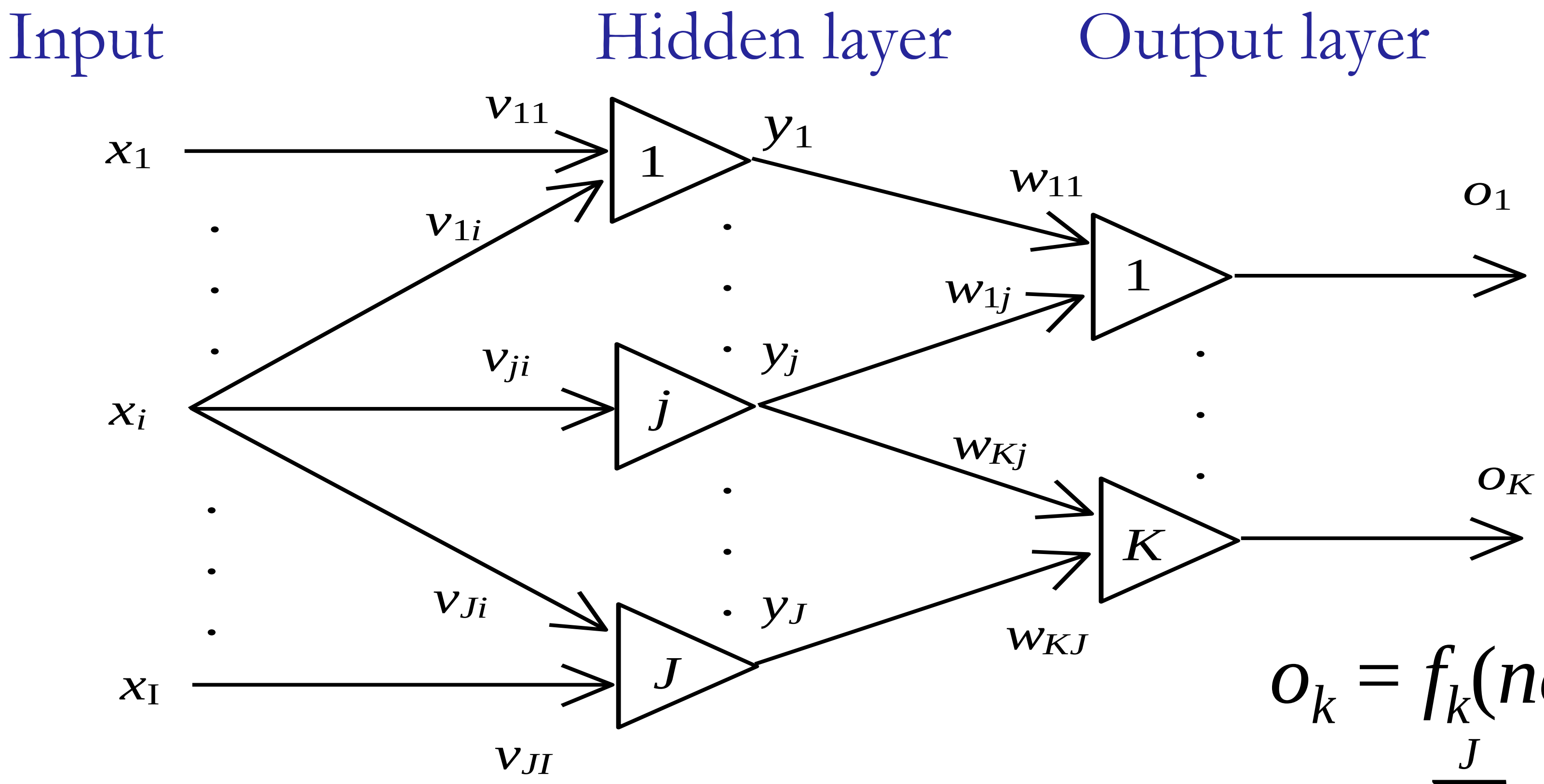
$$\Delta w_{kj} = -\alpha \frac{\partial E^p}{\partial w_{kj}} = \alpha\,(d_k^p - o_k^p)\; f_k'\; y_j$$

- Generalised error signal produced by the (output) neuron *k* is

$$\delta_{ok} = (d_k^p - o_k^p)\; f_k'$$



Feed-forward multi-layer perceptron



$$o_k = f_k(net_k)$$

$$net_k = \sum_{j=1}^{J} w_{kj} y_j$$

$$y_j = f_j(net_j)$$

$$net_j = \sum_{i=1}^{I} v_{ji} x_i$$

Towards error-backpropagation

- Let the training set be

$$A_{train} = \{(\mathbf{x}^1, \mathbf{d}^1)(\mathbf{x}^2, \mathbf{d}^2)...(\mathbf{x}^p, \mathbf{d}^p)....(\mathbf{x}^P, \mathbf{d}^P)\}$$

  - $\mathbf{x}^p$ is the input vector *p* (pattern *p*)
  - $\mathbf{d} = (d_1, d_2, ..., d_k, ..., d_K)$ is the desired output vector

- Error function per pattern *p* = sum of squared errors over *k*

$$E^p = \tfrac{1}{2}\sum_{k=1}^{K}(d_k^p - o_k^p)^2 = \tfrac{1}{2}\sum_{k=1}^{K}(d_k^p - f_k(net_k))^2$$

  - Where $o^p{}_k$ is the actual output of unit *k* for pattern *p*

Gradient descent for a hidden layer

- After presentation of each pattern we want to minimise E^p

$$\Delta v_{ji} = -\alpha \frac{\partial E^p}{\partial v_{ji}} = -\alpha \frac{\partial E^p}{\partial(net_j)} \frac{\partial(net_j)}{\partial v_{ji}} = \alpha\, \delta_{yj}\; x_i$$

- Where the derivative

$$\frac{\partial(net_j)}{\partial v_{ji}} = \frac{\partial\left(\sum_{i=1}^{I} v_{ji} x_i\right)}{\partial v_{ji}} = x_i$$

- Generalised error signal produced by the ***hidden*** neuron *j*

$$-\frac{\partial E^p}{\partial(net_j)} = \delta_{yj}$$

Generalised δ signal for a hidden layer

- Derivation of the generalised error signal for the hidden layer

$$\delta_{yj} = -\frac{\partial E^p}{\partial(net_j)} = -\frac{\partial E^p}{\partial y_j}\frac{\partial y_j}{\partial(net_j)} = -\frac{\partial E^p}{\partial y_j}\frac{\partial\left(f_j(net_j)\right)}{\partial(net_j)} = -\frac{\partial E^p}{\partial y_j} f_j'$$

$$-\frac{\partial E^p}{\partial y_j} = -\frac{\partial\left(\frac{1}{2}\sum_{k=1}^{K}(d_k^p - o_k^p)^2\right)}{\partial y_j} = -\frac{\partial E^p}{\partial o_k}\frac{\partial o_k}{\partial(net_k)}\frac{\partial(net_k)}{\partial y_j}$$

Derivation through $\forall k$

$$-\frac{\partial E^p}{\partial y_j} = \sum_{k=1}^{K}(d_k^p - o_k^p)\frac{\partial\left(f_k(net_k)\right)}{\partial(net_k)}\frac{\partial\left(\sum_{j=1}^{J} w_{kj} y_j\right)}{\partial y_j}$$

Generalised δ signal for a hidden layer

- Continuation of derivation

$$-\frac{\partial E^p}{\partial y_j} = \sum_{k=1}^{K}(d_k^p - o_k^p)\,\frac{\partial\left(f_k(net_k)\right)}{\partial(net_k)}\,\frac{\partial\left(\sum_{j=1}^{J} w_{kj}y_j\right)}{\partial y_j}$$

$$-\frac{\partial E^p}{\partial y_j} = \sum_{k=1}^{K}(d_k^p - o_k^p)\; f_k'\, w_{kj} = \sum_{k=1}^{K}\delta_{ok} w_{kj}$$

- Thus the resulting generalised error signal for the hidden unit *j* is

$$\delta_{yj} = \left(\sum_{k=1}^{K}\delta_{ok}\; w_{kj}\right) f_j'$$

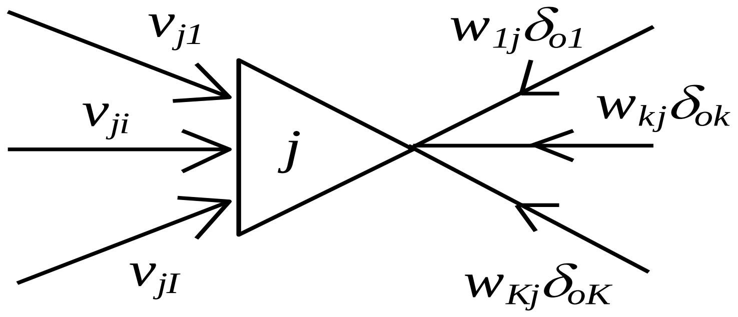
Error back-propagation for a hidden layer

- Generalised error signal for the neuron *j* in the hidden layer

$$\delta_{yj} = \left(\sum_{k=1}^{K} \delta_{ok} \; w_{kj} \right) f'_j$$

- Hidden weights modify as $\Delta v_{ji} = \alpha \, \delta_{yj} \; x_i$

- With the final formula $\Delta v_{ji} = \alpha \left(\sum_{k=1}^{K} \delta_{ok} \; w_{kj} \right) f'_j \; x_i$



Error Back-Propagation Algorithm

1. Choose $\alpha \in (0, 1]$. Generate randomly $\mathbf{w}(0)$ and $\mathbf{v}(0) \in [-0.5, 0.5]$. Set $E = 0$, the counter for patterns $p = 0$, the epoch counter $e = 0$. Choose small $\varepsilon > 0$ as a stopping criterion.
2. For pattern p calculate hidden activities $\mathbf{y}^p$ and actual output $\mathbf{o}^p$
3. For $\forall k$ calculate the learning signal δ_{ok} and for $\forall j$ calculate δ_{yj}
4. Update each **output** weight (including θ): $w_{kj} \leftarrow w_{kj} + \alpha\, \delta_{ok}\, y_j$
5. Update each **hidden** weight (including θ): $v_{ji} \leftarrow v_{ji} + \alpha\, \delta_{yj}\, x_i$
6. If $p < P$, go to step 2. Else continue.
7. Freeze all weights and calculate the total error $E = \Sigma_p E^p$.
8. If $E < \varepsilon$, stop training. Else set $E = 0$, $p = 0$, $e = e + 1$, and go to step 2.

Simple example of MLP

- Inputs are x coordinates of points of some unknown target function $y = f(x)$.

- 2 hidden units (1 and 2) have hyperbolic tanh activation function.

- One output unit (No. 3) sums linearly the outputs of 2 hidden units minus an adjustable bias.

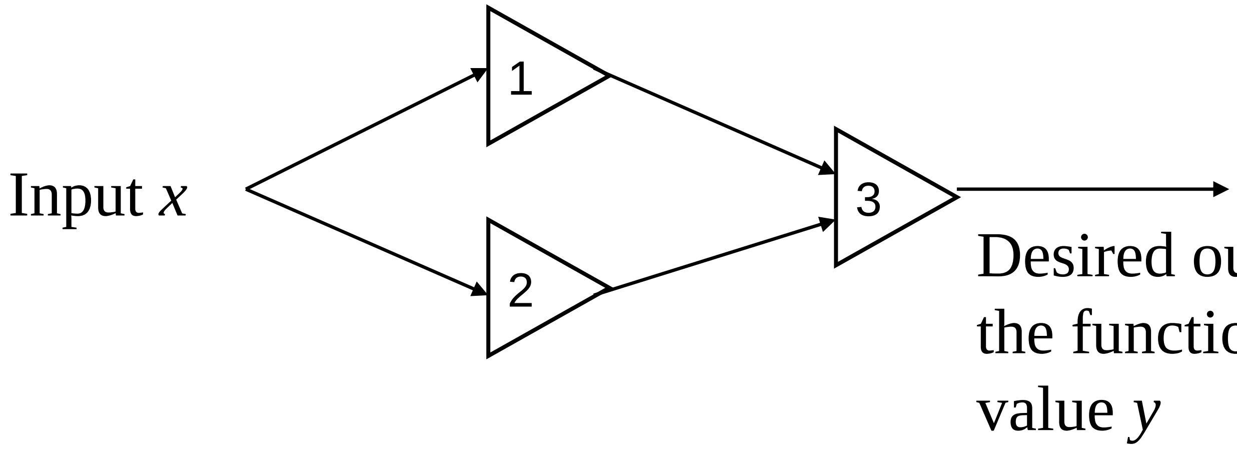
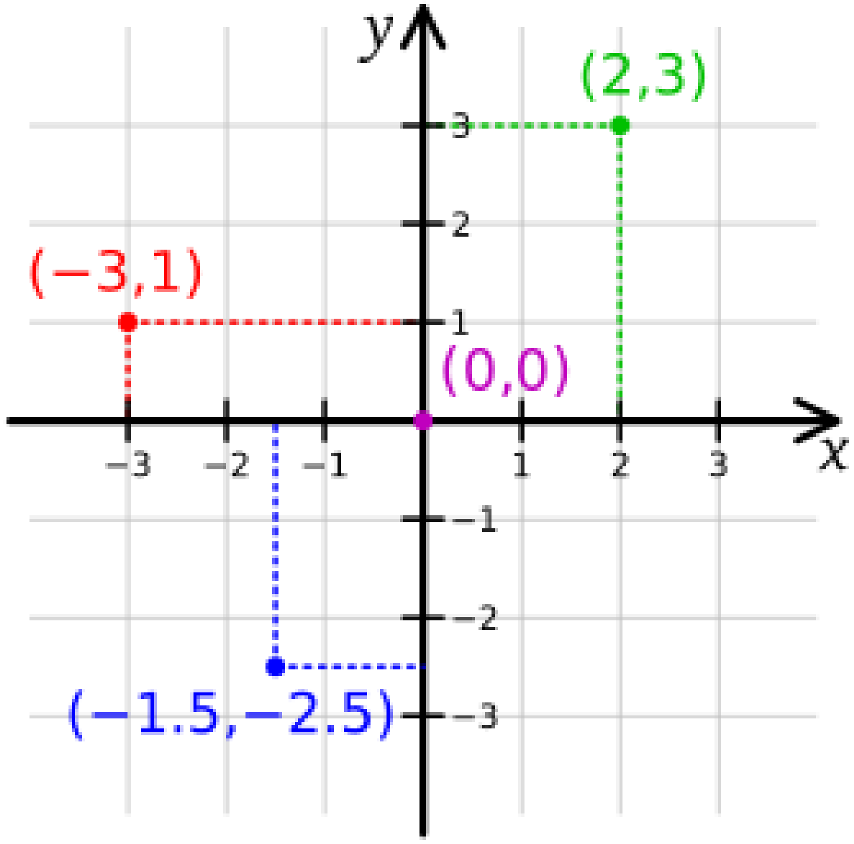


Illustration of the input and target outputs

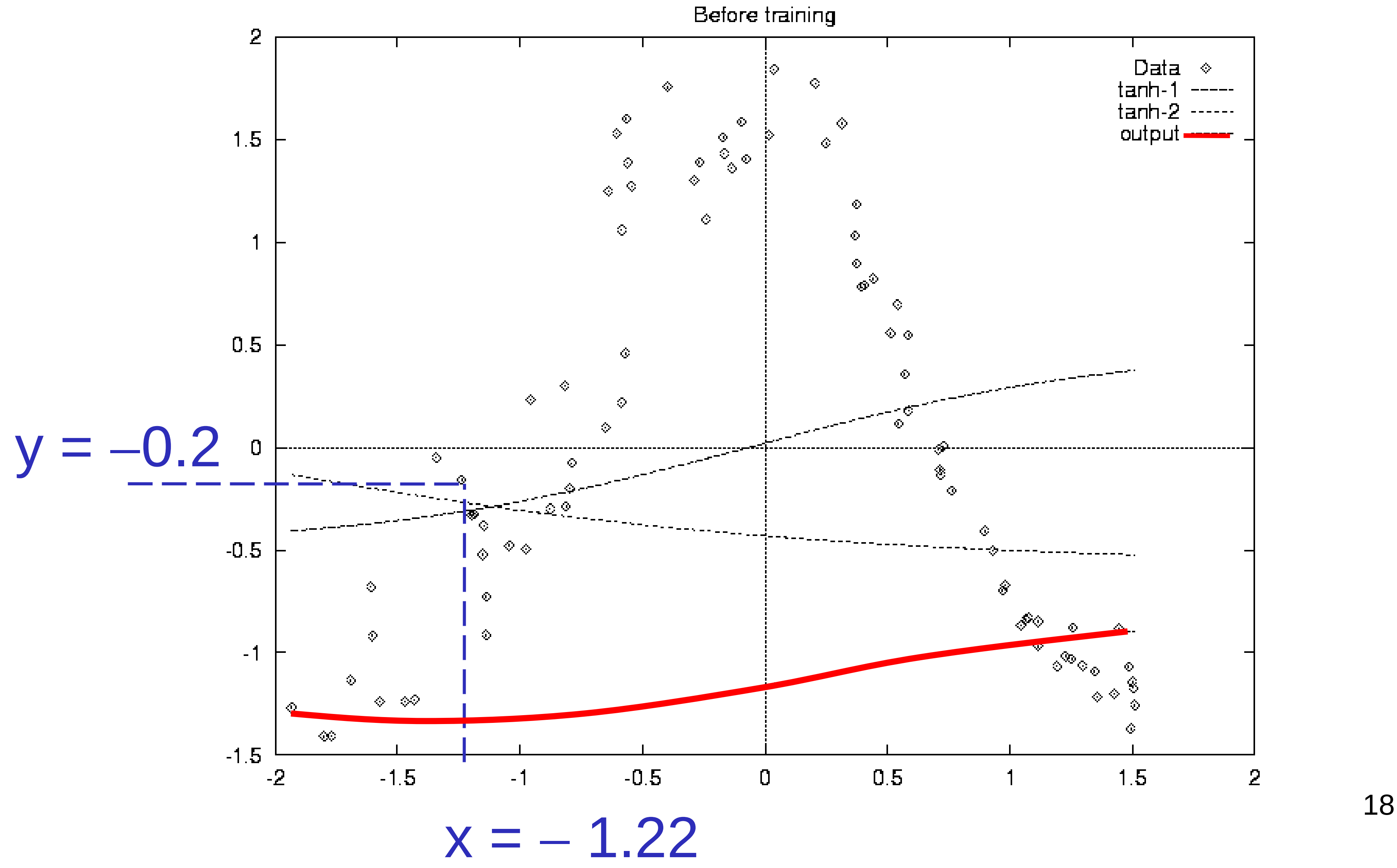
- The input will be a single number, the value of the x coordinate of the point in the 2D space.

- The target output value will be the value of the y coordinate of that point.

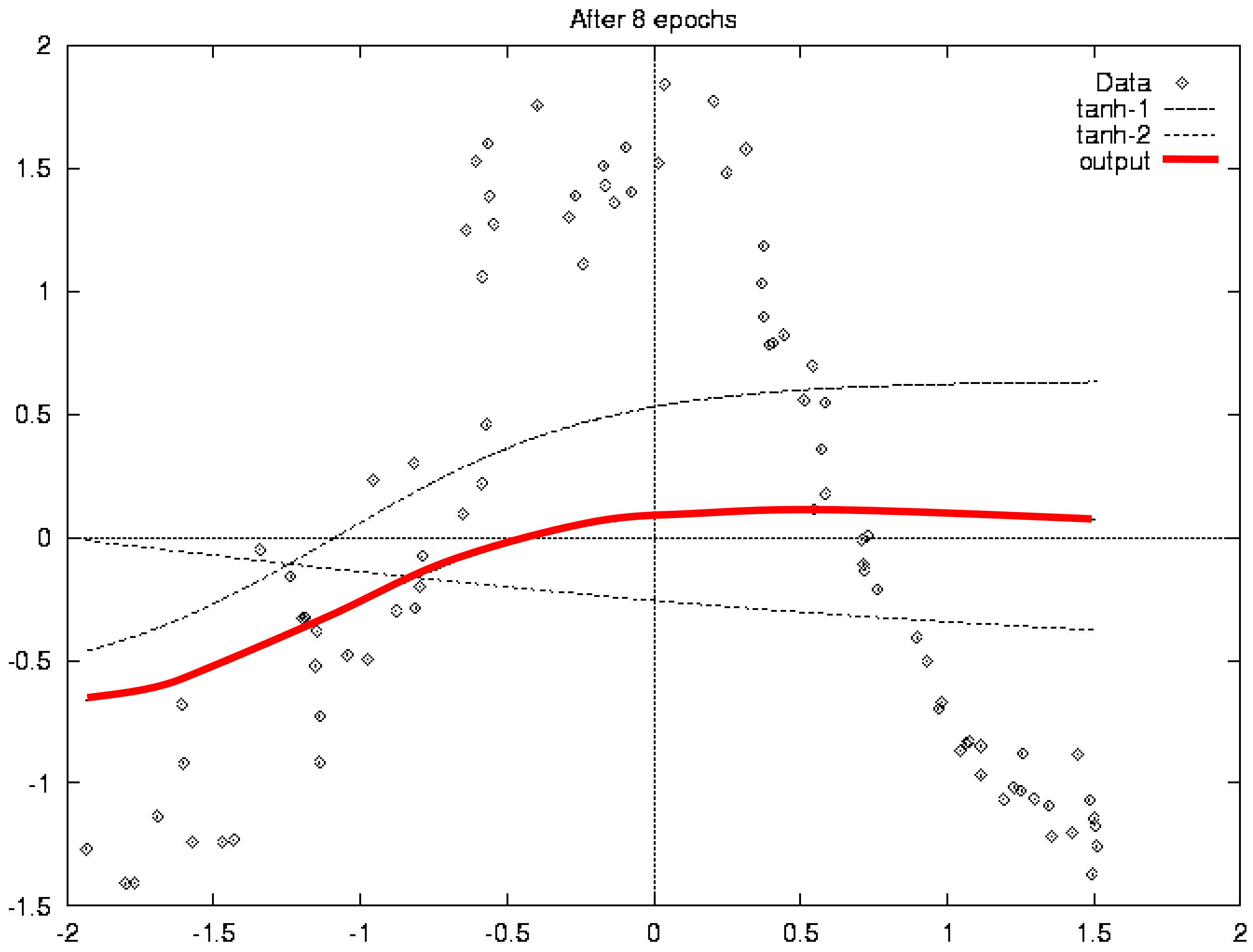


Task: approximate the nonlinear function $y = f(x)$

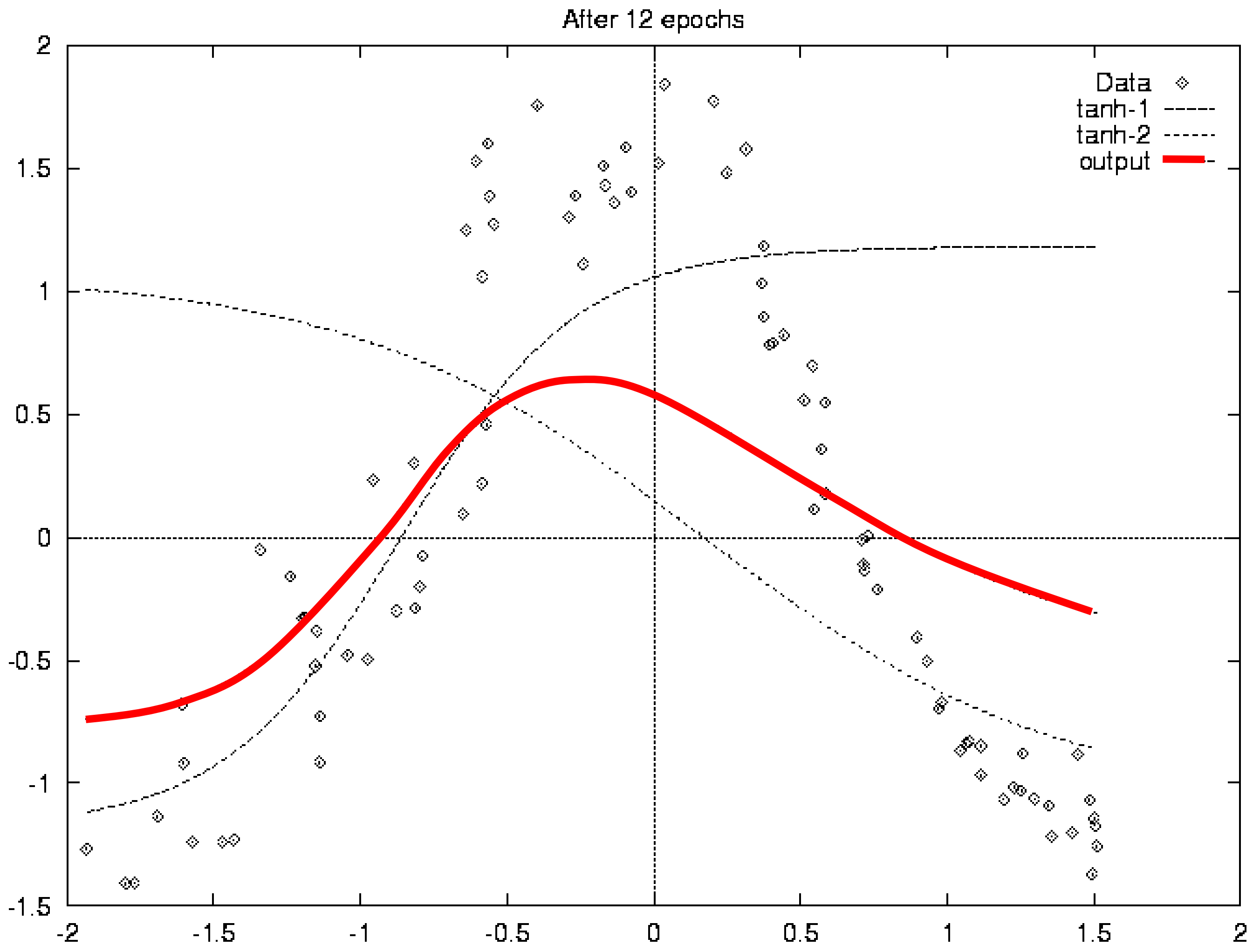
- Input = x coordinate of points, desired output is the value y



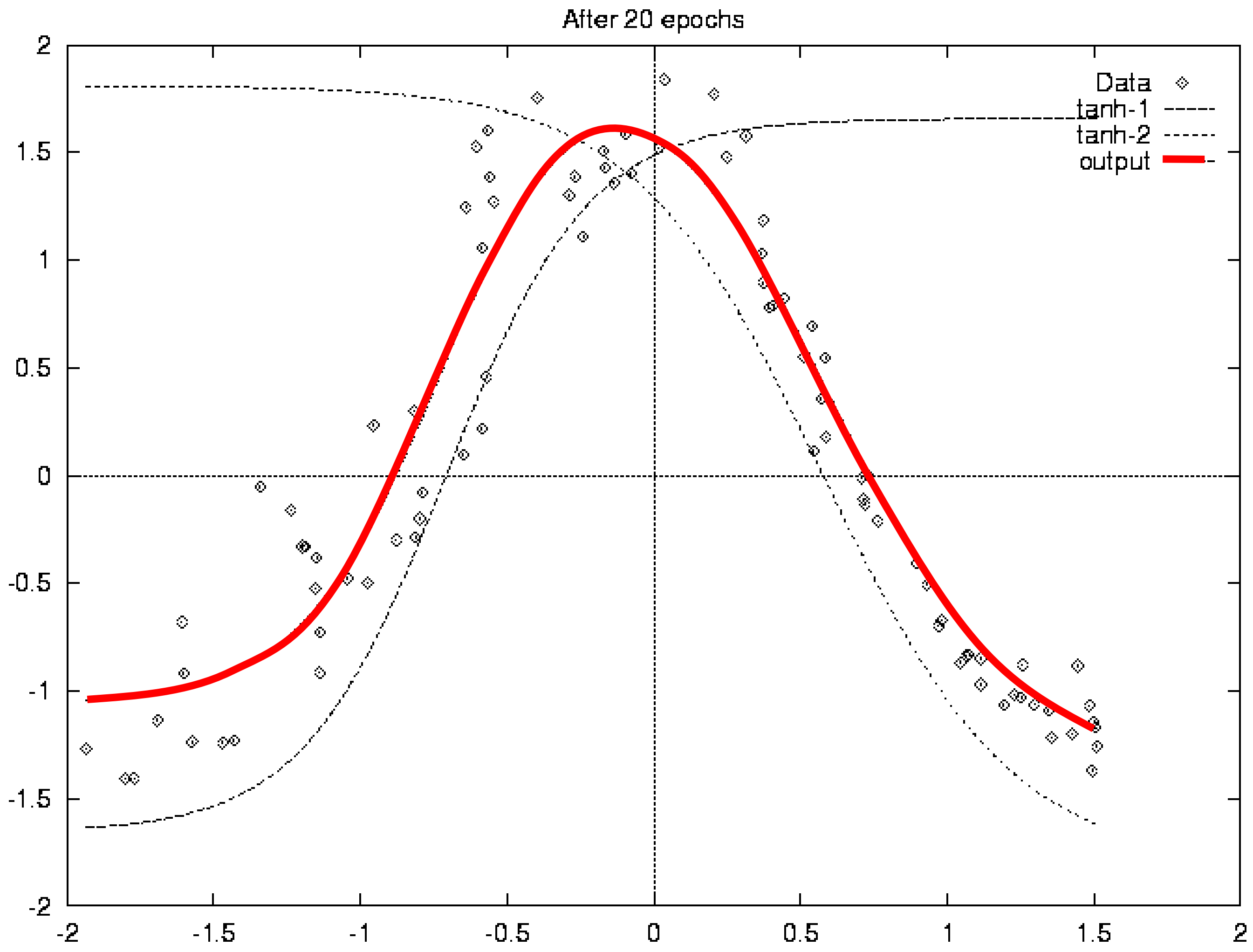
MLP output after 8 epochs through all data points



MLP output after 12 epochs through all data points



Approximation of nonlinear data has been achieved



Generalization (prediction)

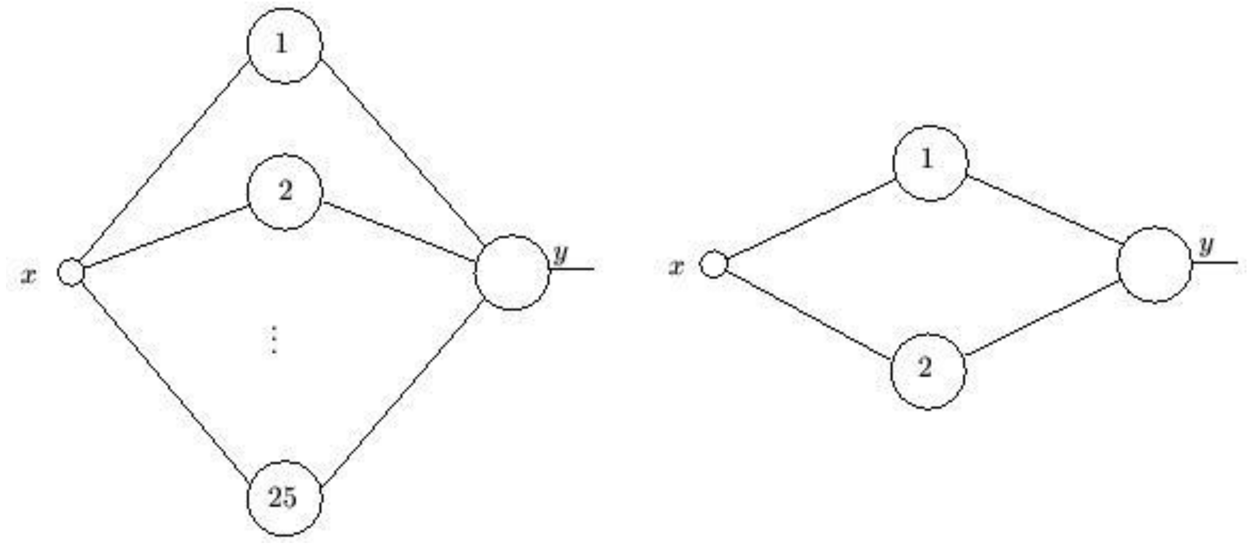
- A network is said to generalise if it gives correct results for inputs not in the training set = prediction of new values.

- Generalization is tested by using an additional **test set**
  - After training, the weights are frozen
  - For each test input we evaluate the MLP *prediction* of the functional value

- The training set and the test set are obtained by separating original data set into 2 parts.
  - For instance, we choose randomly 80% of the data points as a training set and the remaining 20% will constitute the test set

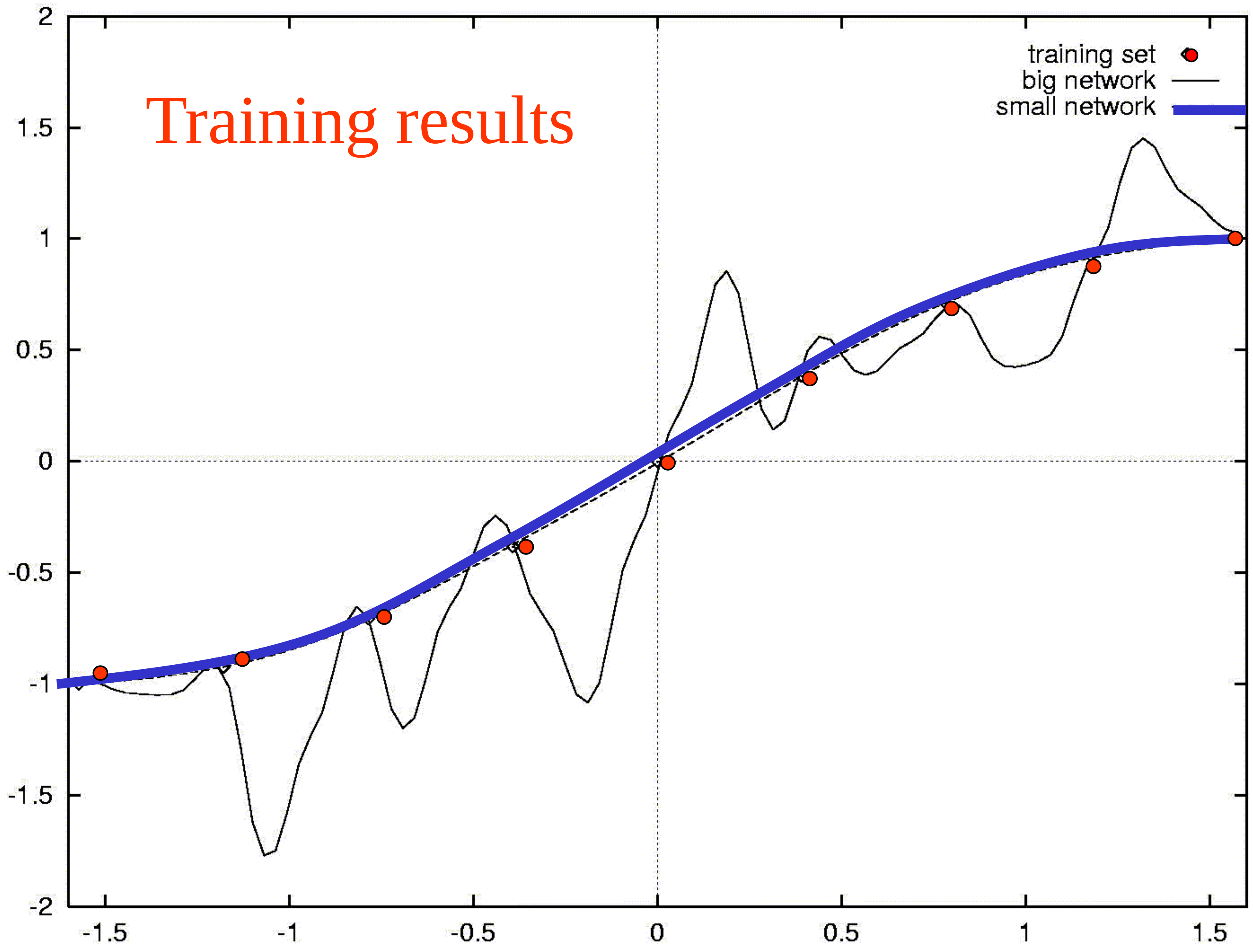
Example of good and bad generalisation

- Two feed-forward MLPs, one with 25 hidden sigmoid neurons and the other one with 2 hidden sigmoid neurons:
  - Output layer : one linear neuron



- Target $f(x) = \sin(x)$, interval $-\pi/2 \le x \le \pi/2$,
- Training set $= \{(-\pi/2 + i\pi/8, \sin(-\pi/2 + i\pi/8)) \mid 0 \le i \le 8\}$.

Training results



Example taken from Alexandra I. Cristea

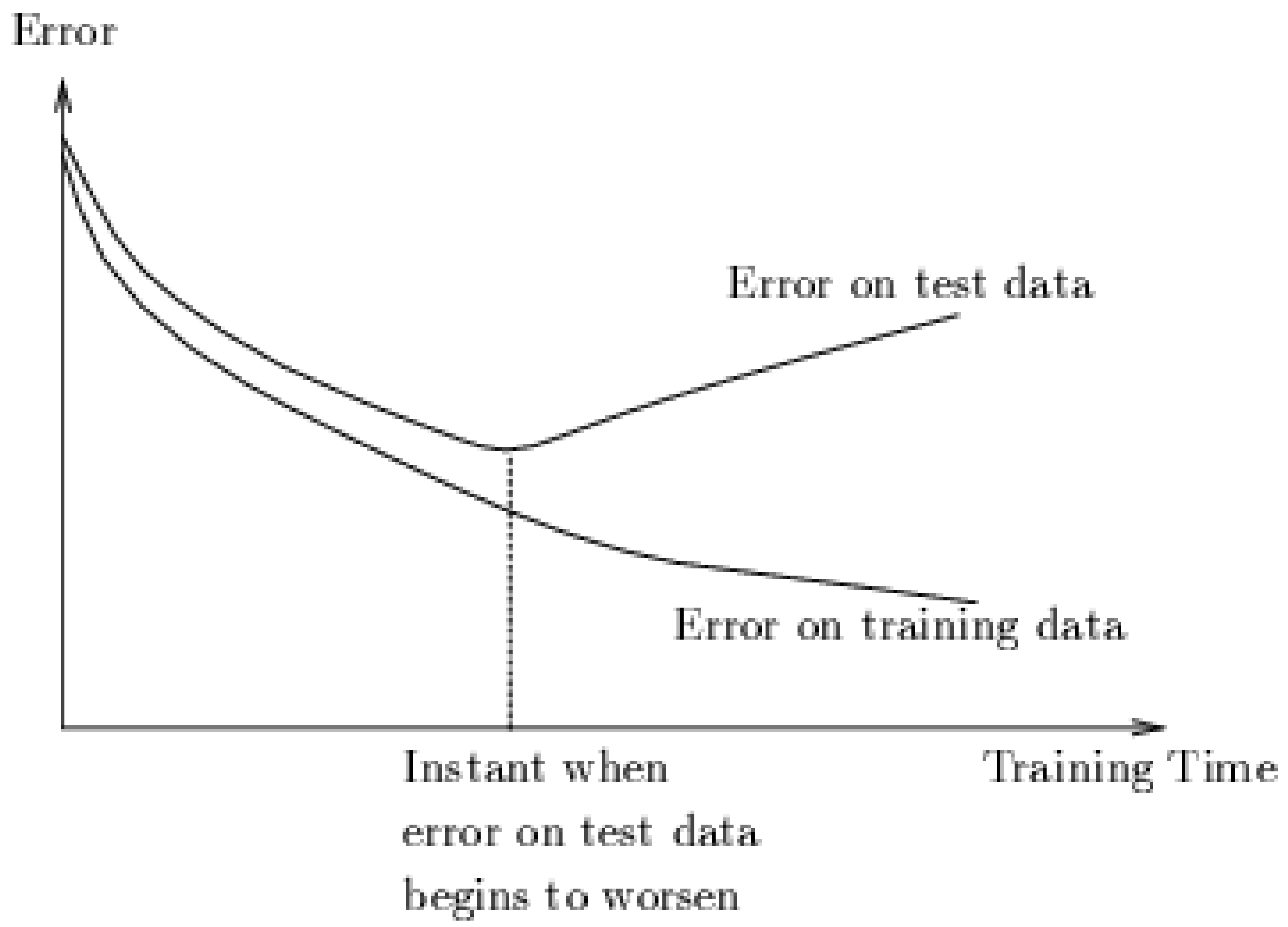
Overtraining (overfitting, preučenie)

- Both MLPs are consistent in all training set points. The complexity of the fitted (hypothesis) curve depends on the number of hidden units J.

- Small MLP generalises well, big MLP very bad:
  - It memorized the training set but gives wrong results for other inputs => overfitting
  - Too many neurons and weights lead to polynomial of a high degree

- ***Overtraining***: a network that performs very well on the training set but very bad on the test points is said to be overtrained.

Early stopping: how to avoid overfitting:



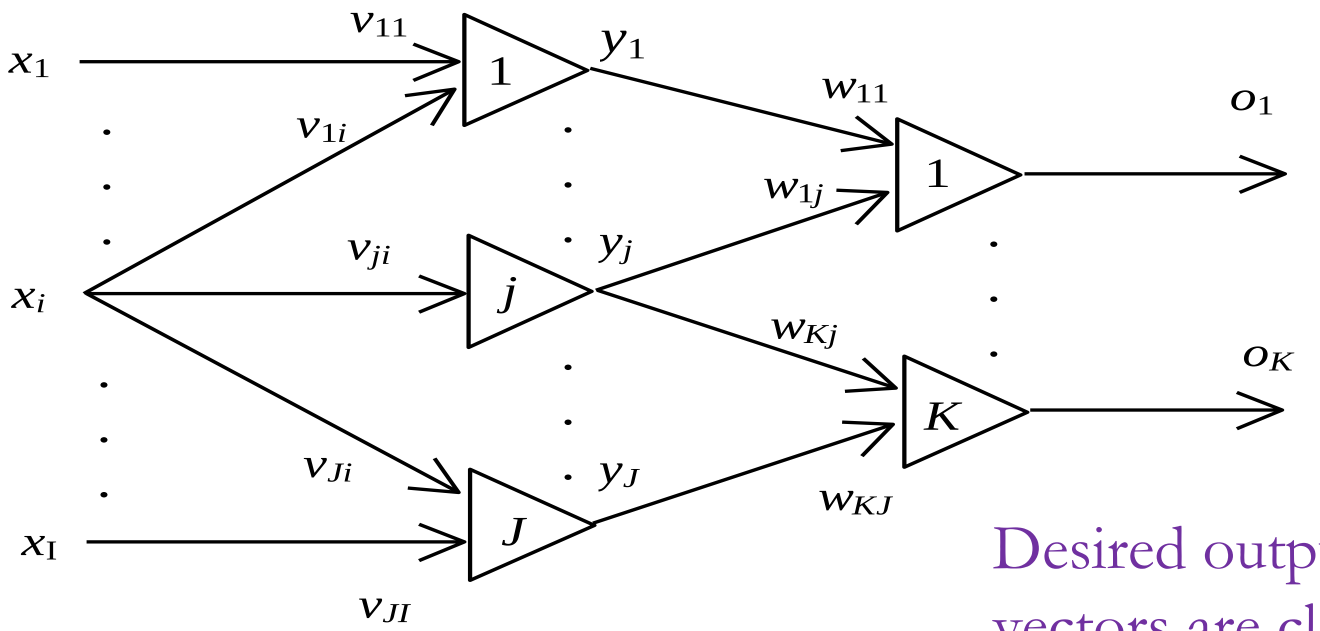
Stop the training here!

Model selection

- We do not know how many hidden units to use for MLP to approximate well the given nonlinear function and obtain a good generalisation.

- *Model selection*, we experimentally evaluate several MLPs with different number of hidden units how well they perform on the **test data**.
  - **K-fold cross-validation**: run K experiments, each time setting aside a different 1/K of the data to test on;
  - **Leave-one-out**, we leave only one example for test, and repeat testing N times (for the set of N examples)

Classification

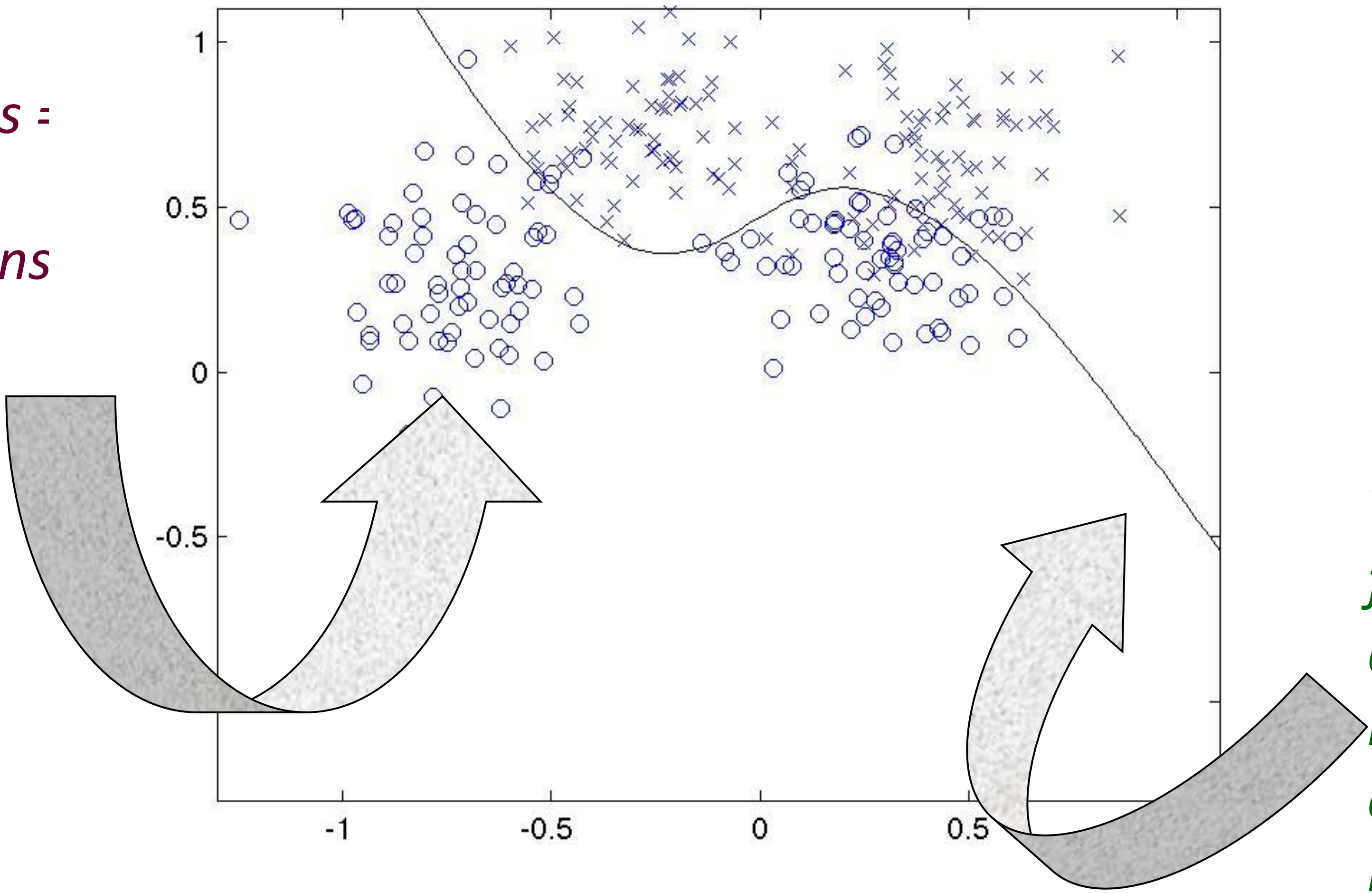


Input vectors are examples of objects from classes, can be either binary coded $x_i \in \{0, 1\}$ or normalised vectors of real values $x_i \in <0, 1>$

Desired output vectors are class labels: one-hot coding, i.e. for each class one unit is on, others are off

Classification task: find a nonlinear class boundary

Input vectors = input patterns



Target function is a nonlinear class boundary

Output desired values

- **Pattern classification:**
  - K classes: K output nodes
  - a class label is represented by one-hot coding

    $\mathbf{d}^{1}$ =(1,…, 0); …; $\mathbf{d}^{K}$ = (0,…, 1)
  - With sigmoid output function, the output values of nodes in output layer will never be 1 or 0, but rather 1 – ε or ε, respectively.
  - Or when classifying an input **x** using a trained MLP, classify it to the k^{th} class when $d_k > d_l$ for all l != k
  - Often the Softmax outputs are used, i.e. the output values o_k are probabilities

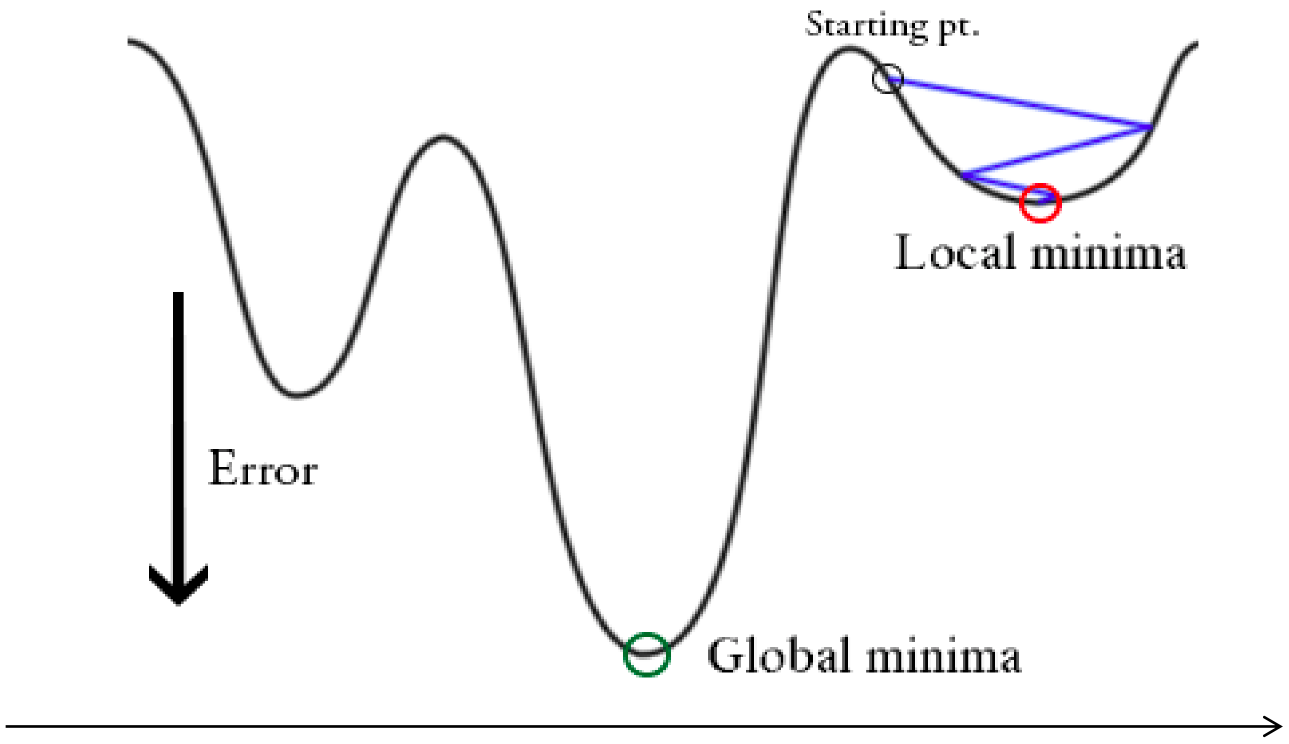
$$P_k = \frac{\exp(net_k)}{\sum_{k}^{K} \exp(net_k)}$$

Strengths of error BP learning

- **Great representation power**
  - Any function can be represented by MLP
- **Wide applicability of error BP**
  - Only requires that a good set of training samples is available
  - Does not require substantial prior knowledge or deep understanding of the domain itself (ill-defined problems)
  - Tolerates noise and missing data in training samples (graceful degrading)
- **Easy to implement** the core of the learning algorithm
- **Good generalization power**
  - Often produces accurate results for new inputs outside the training set

The drawback of error-backpropagation

- Quality of solution depends on the starting values of weights.
- Error-backpropagation ALWAYS converges to the nearest minimum of total Error.
- There are various ways how to improve the chances to find the global minimum, which we are **not** going to deal with in this course.



Problem of local minima of *E*

- Problem with gradient descent approach
  - only guarantees to reduce the total error to a **local minimum** (thus ***E*** can not be reduced to zero)
  - Cannot escape from the local minimum error state
  - How bad: depends on the shape of the error surface. Too many valleys/wells will make it easy to be trapped in local minima
  - *Possible remedies*:
    - Try nets with different # of hidden layers and hidden nodes (they may lead to different error surfaces, some of which might be better than others)
    - Try different initial weights (different starting points on the surface)
    - Forced escape from local minima by random perturbation (e.g., simulated annealing)

More faults of error BP

- Learning often takes a **long time** to converge
  - Complex functions often need *hundreds or thousands* of epochs
- The net is essentially a **black box**
  - It cannot provide an intuitive or causal explanation for the result.
  - This is because the hidden nodes and the learned weights do not have clear semantics.
    - What can be learned are numerical parameters, not general, abstract knowledge of a domain
  - Unlike many statistical methods, there is **no** theoretically well-founded way to **assess the quality** of error BP learning:
    - What is the confidence level one can have for a trained BP net with the final E (which may or may not be close to zero)?

Summary

- Supervised learning by error-backpropagation can be used either for nonlinear regression or pattern (object) classification.

- *Nonlinear regression:*
  - the training set consists of real data values, i.e. the set of pairs {**x**, *f*(**x**)}
  - *f*(**x**) is the unknown function and **x** is the input vector.
  - The task is to **approximate** *f*(**x**) by the output of the MLP: **o(x)**
  - The error signal is calculated based on difference between **o(x)** and *f*(**x**) for the training set.

- *Classification:*
  - the training set consists of pairs: input & desired output (i.e. class labels)
  - The task is to learn how to find the nonlinear classification boundary
  - we know, which class the object/input falls into and we provide *desired or target outputs values.*

Evolutionary robotics

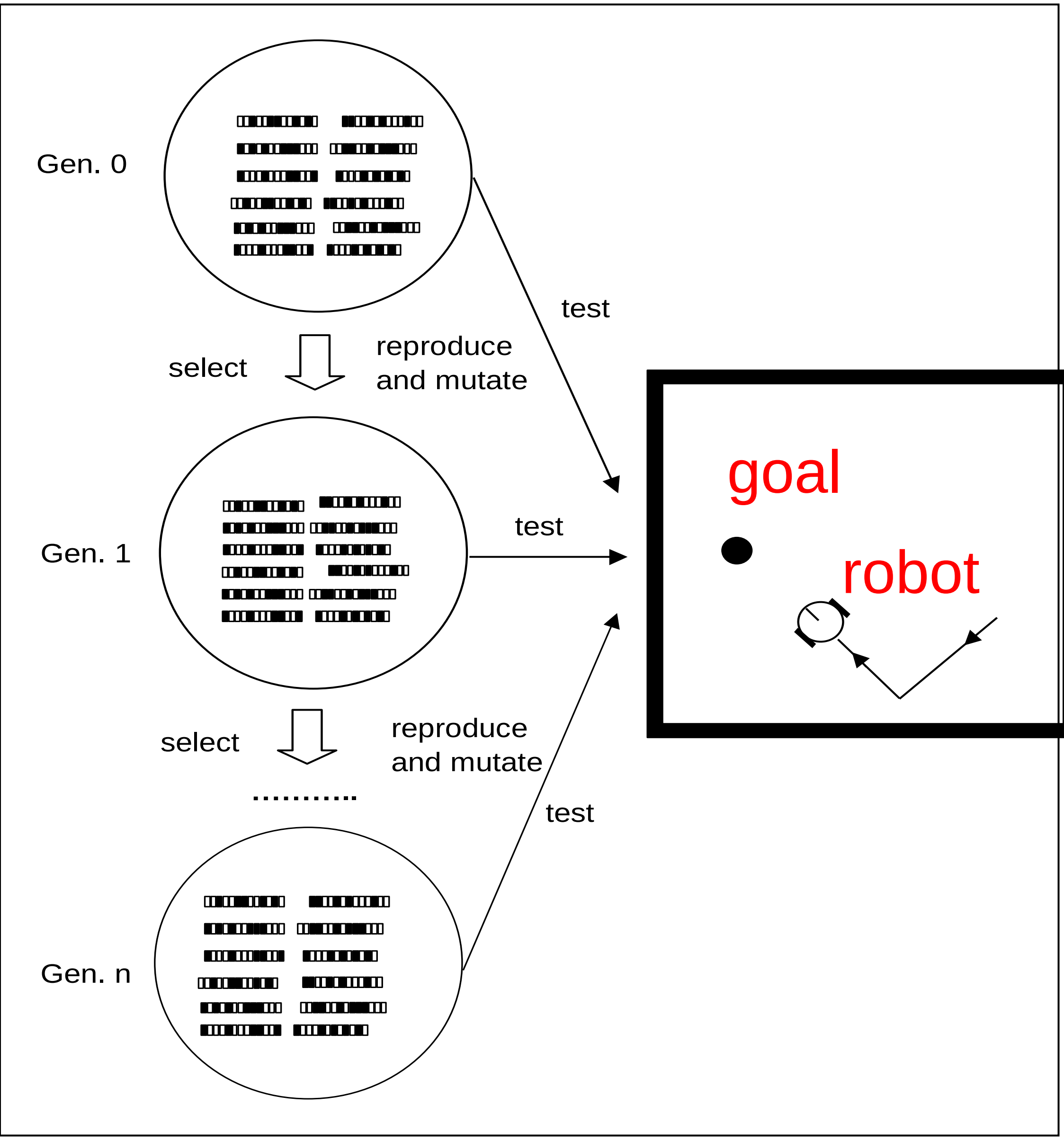
- Initial population of different "genotypes", each representing one possible configuration of the control system (e.g., weights of a neural network) of a robot, are created randomly.

- Then, each genotype is translated into the corresponding phenotype (output of a robot) and tested in the environment for its ability to perform a desired task.

- The robots that have obtained the highest fitness are allowed to reproduce by generating copies of their genotypes with the addition of changes introduced by mutation.

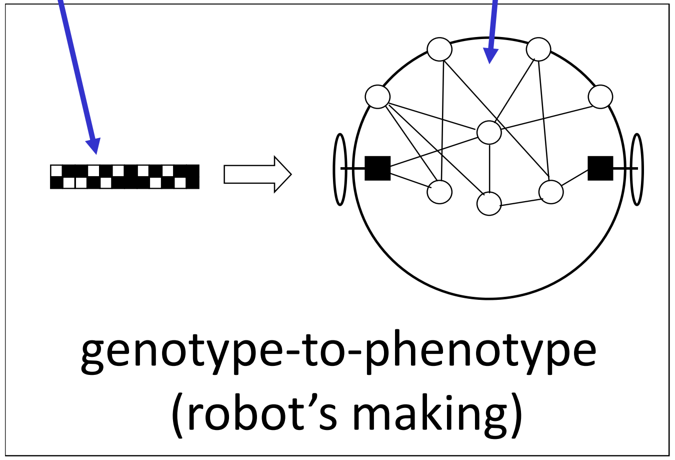
- The process is repeated for a certain number of generations until, hopefully, the desired performances are achieved.

The method of ER: illustration



Genotype: weights of a neural net

Robot's control system



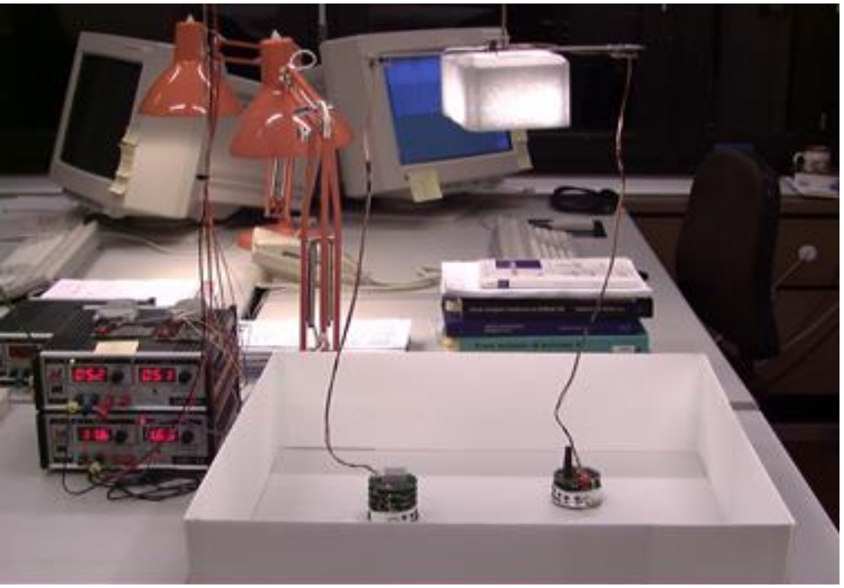
Genotype = chromosome

Phenotype = the visible outcome

The method of ER – contd.

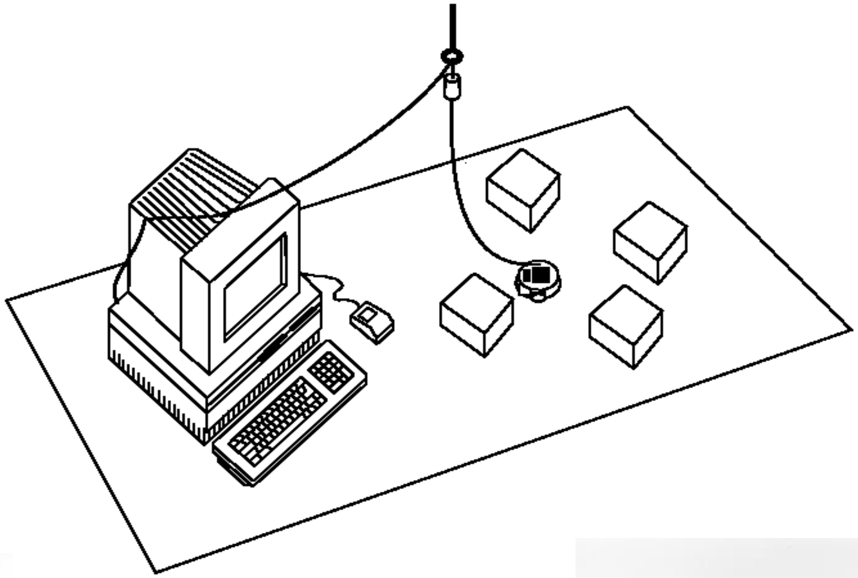
- The designer usually specifies how genetic information is translated into the corresponding phenotype.
  - *i.e., how the output of a neural net is used to navigate the robot*
- The mapping between the genotype and phenotype is usually task independent. It depends on a particular construction of a robot.
- The experimenter designs the ***fitness function***, that is some formula, which is used to measure how much an individual robot is able to accomplish the desired task.
  - *Design of an appropriate fitness function is the most difficult and nontrivial part of the evolutionary robotics approach.*

Setting for ER



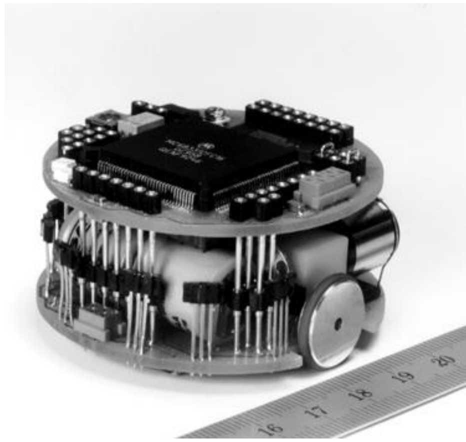
[Floreano, Nolfi, & Mondada, 1998]

Evolvable neural network in the computer navigates the robot



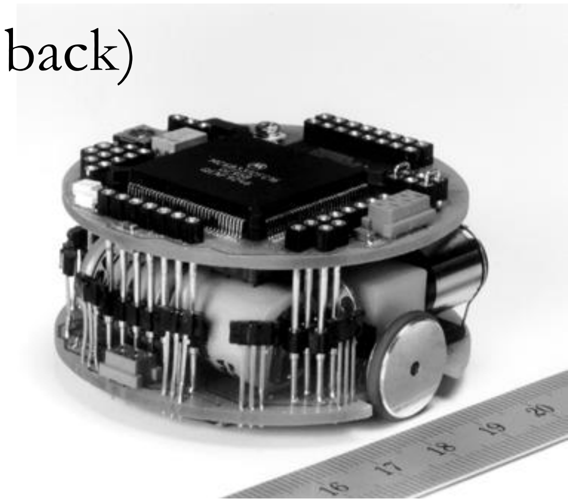
[Floreano and

Mondada 1994]



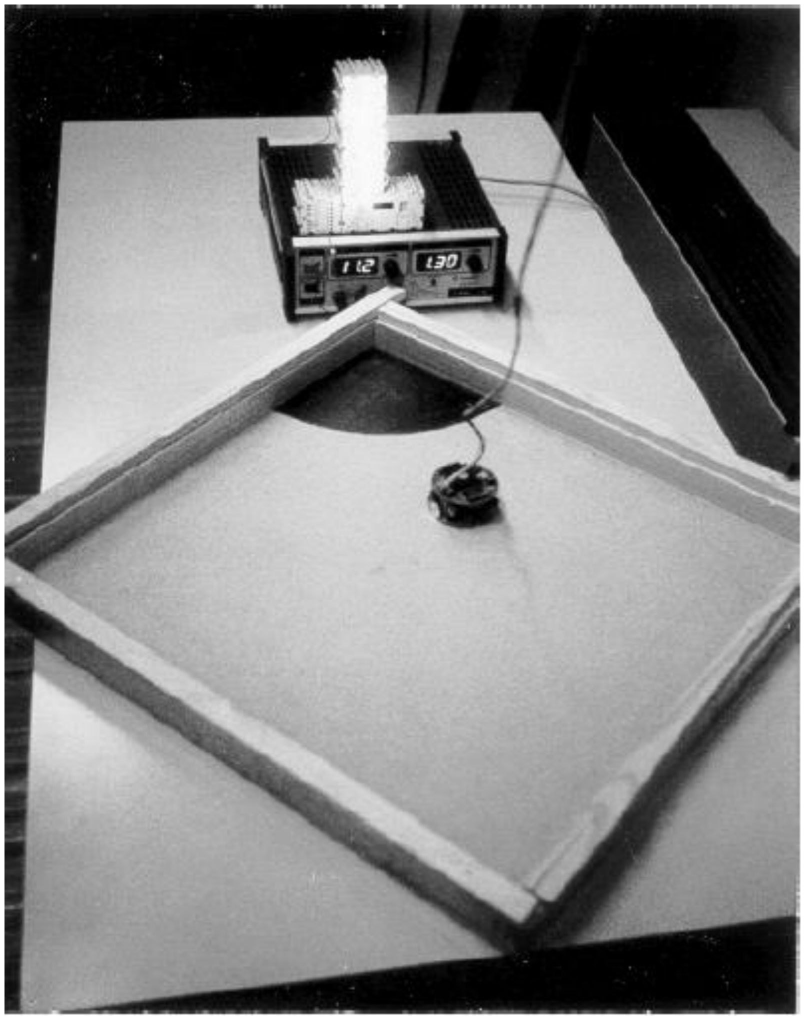
Robot Kephera^{TM}

- Developed at the Swiss Federal Institute of Technology in Lausanne, Switzerland
- Diameter = 55 mm, hight = 30 mm, weight = 70g
- 2 wheels and two teflon balls under the platform
- Each wheel is controlled by a DC motor and can rotate in both directions
- 8 infrared proximity sensors (6 in front, 2 at the back)
- 2 light sensors (1 in front, 1 in back)
- 1 sensor measuring floor brightness
- 1 sensor of the battery charge level



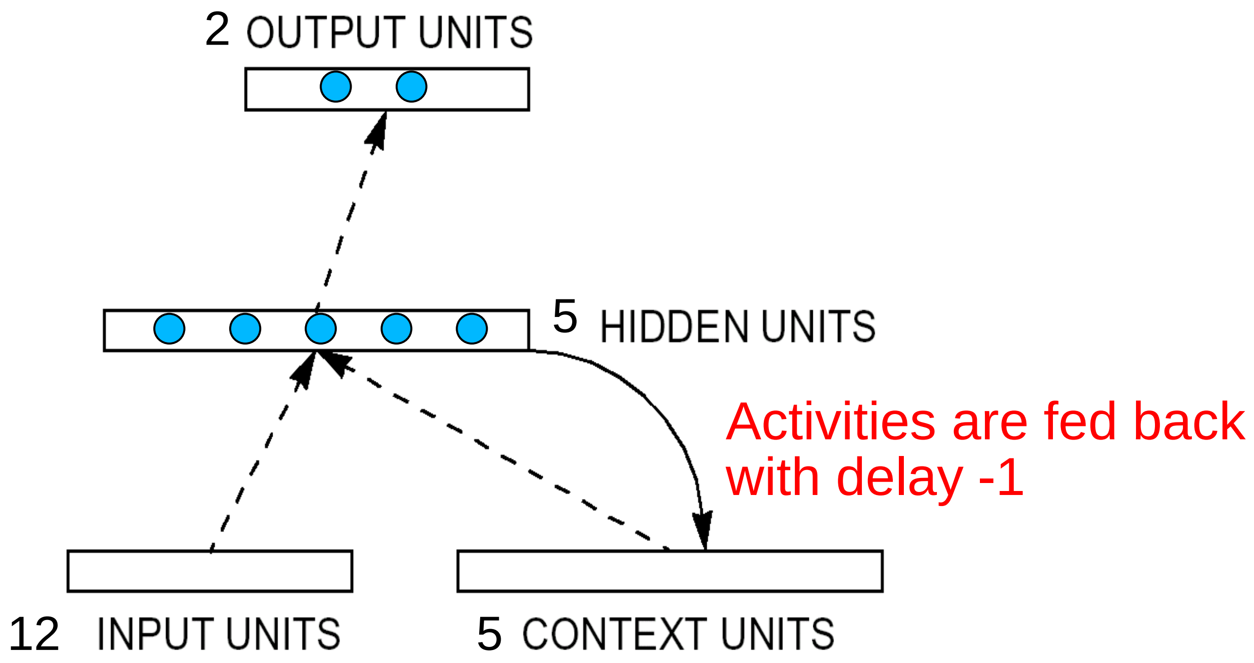
Task: to live as long as possible

- Robot was equipped with a simulated battery that lasted only 50 actions (each action lasts 300 ms).
- It moved in the rectangle with a simulated battery charger in one corner under the lamp. The charging area was black.
- Each individual started with a full battery (life = 50 actions).
- Entering charging area prolonged life for additional 50 actions.



Neuro-controller architecture

- Input: values from 8 proximity sensors + 2 light sensors + 1 sensor measuring floor brightness + 1 sensor of the battery level (all values were normalised to (0, 1) interval);
- 5 hidden sigmoid units with recurrent feedback;
- Two linear output units – each controlling one wheel;



Fitness function

$$F = V(1 - i) \qquad 0 \leq V, i \leq 1$$

- V is the average rotation speed of the two wheels;
- i is the activation value of the sensor with the highest activity
- F was computed after each robot action (i.e. after 300ms) and accumulated during the life.

Fitness function

$$F = V(1 - i) \qquad 0 \leq V, i \leq 1$$

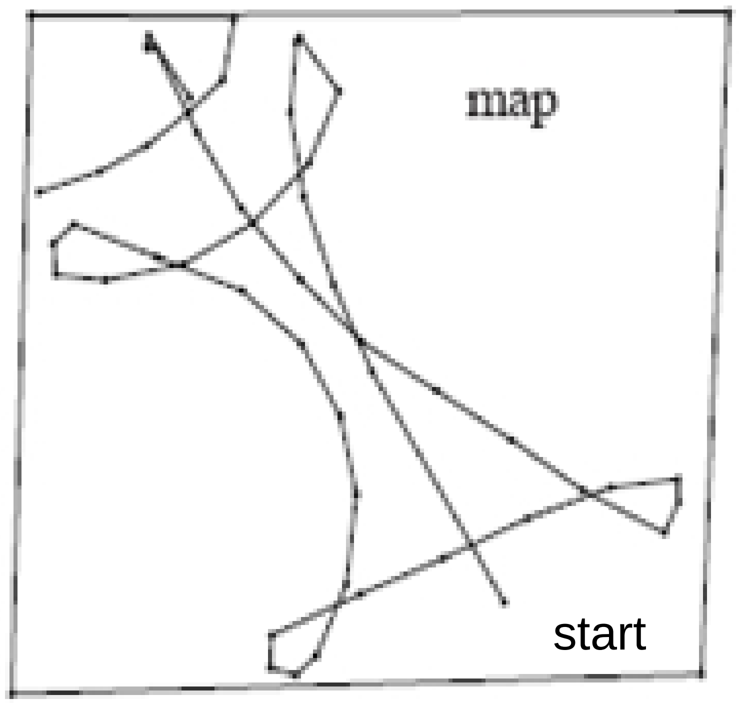
- The task is to **maximise** the total fitness over the whole life.
- First by maximising speed of the wheels V – thus passive behaviour or standing at the charging area is discouraged.
- i is the activation value of the sensor with the highest activity, so the robot avoids walls (high signal from proximity sensors), likes the black painted floor (low signal from the floor brightness sensor), prefers low battery level and avoids light.

Evolving neurocontroller

- Let the architecture of the neurocontroller be fixed, but in general we can evolve also the number of hidden, input and output units.

- Algorithm for evolution of weights for neural network:
  - Encode the weights of a neural net as a binary vector;
  - At $t = 0$ generate a large number of random weight vectors;
  - For each neurocontroller calculate the value of fitness function;
  - Perform parent/survival selection (roulette wheel);
  - Generate the offspring, i.e., new generation of neurocontrollers by one-point crossover and randomly mutate the offspring;
  - Repeat until desired behaviour is achieved.

Results of evolution



- Pictures shows behaviour of the best individual after 240 generations (each of 100 individuals)

- Robot navigated through environment, avoiding the walls, and when the battery was almost empty, rushed to the charging area, which it left immediately.

- This behaviour was maintained indefinitely, wherever in the area the robot was placed at start.

[Floreano D. and Mondada F. (1996) Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics-Part B* 26: 396-407.]