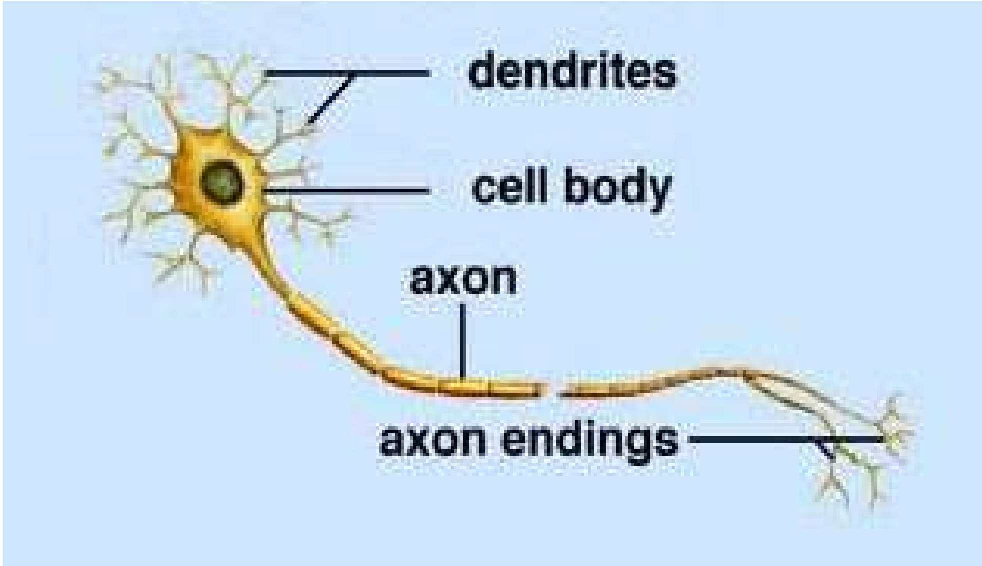
Artificial Neural Networks
Part I

Beatrice Lazzerini
Department of Information Engineering
University of Pisa
ITALY

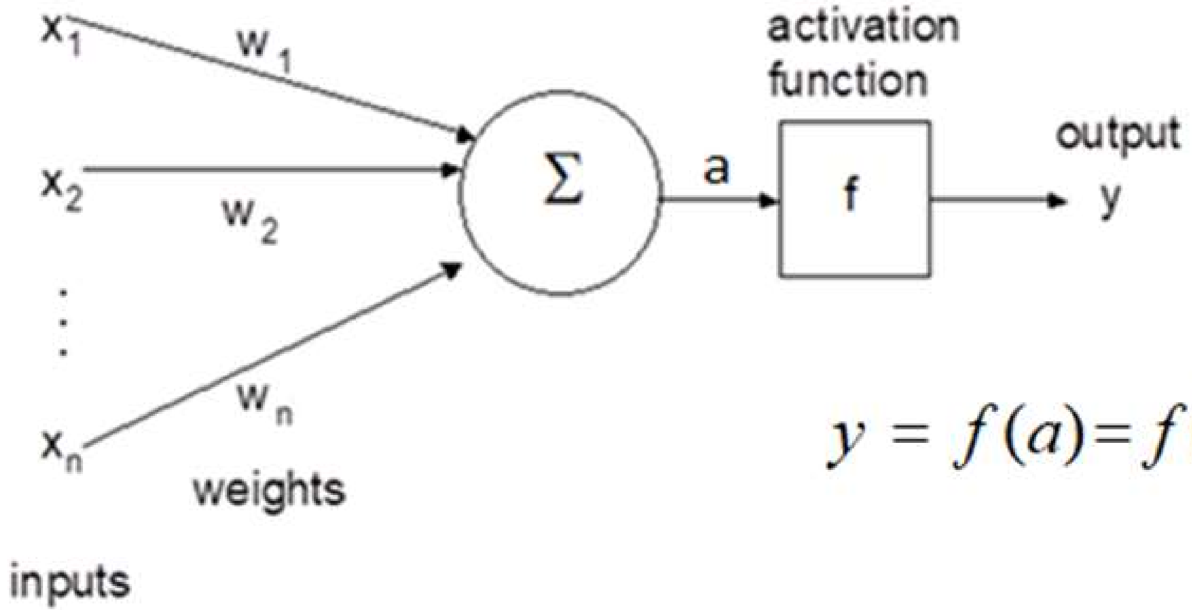
Biological neuron



A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

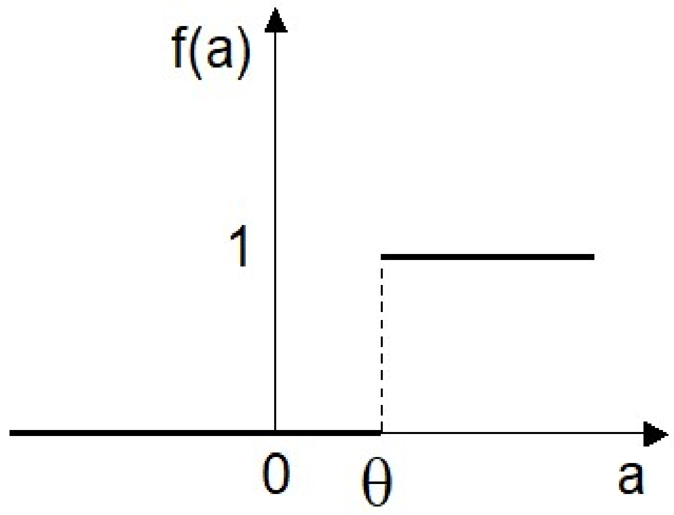
Artificial neuron



$$y = f(a) = f\left(\sum_{j=1}^{n} w_j x_j\right)$$

- An artificial neuron has many inputs but only one output.
- Each input connection has an associated adjustable *weight* (a real number). A positive (negative) weight has an excitatory (inhibitory) effect on the neuron.
- A neuron computes some function *f* (called *activation* function) of the weighted sum of its inputs.
- The output can be a real number, a real number restricted to some interval (e.g., [0,1] or [-1,1]), a discrete number (e.g., {0, 1} or {+1, -1}).

- Binary threshold function

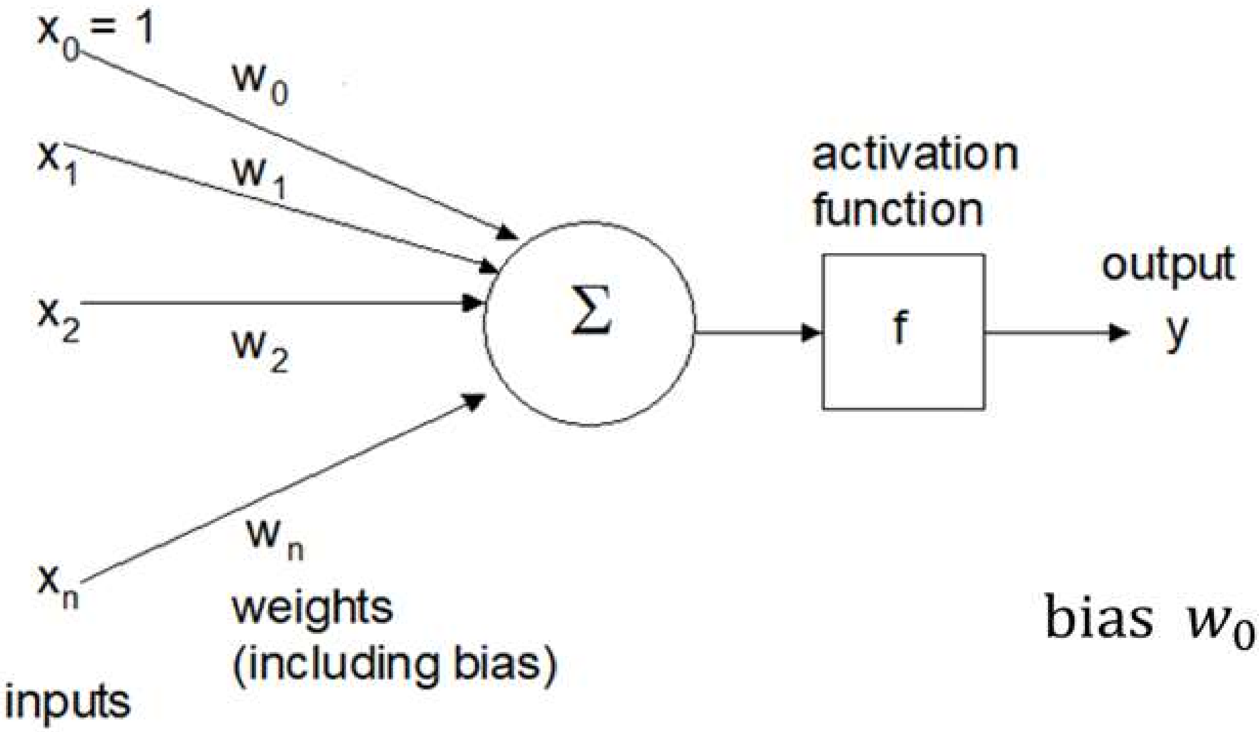


$$y = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{if } a < \theta \end{cases}$$

- Typically, the *threshold* is subtracted from the weighted sum of the inputs

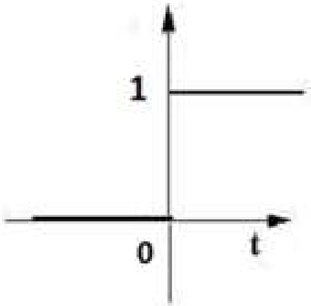
$$y = f\left(\sum_{j=1}^{n} w_j x_j - \theta\right) = f\left(\sum_{j=0}^{n} w_j x_j\right)$$

and the negative of the threshold (called *bias*) is considered as a weight connected to a unit that always has an output of 1 ($x_0 = 1$).

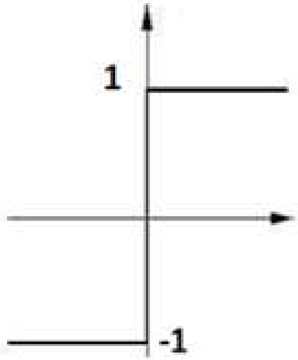


$$\text{bias } w_0 = -\theta$$

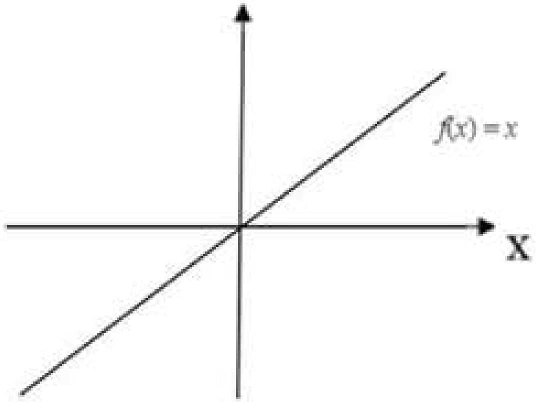
Some activation functions



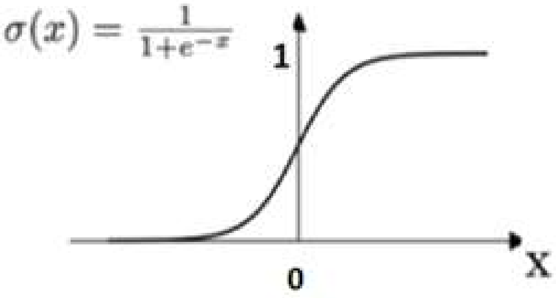
Unit step



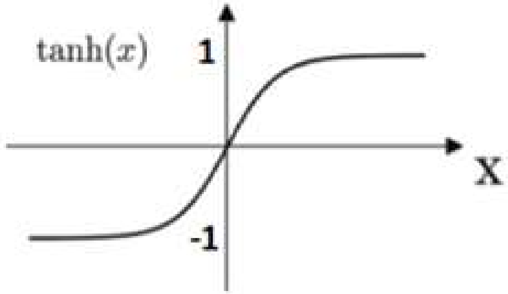
Sign



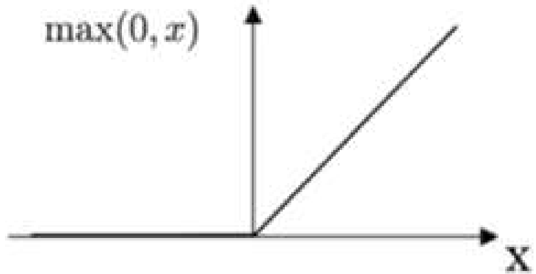
Linear



Sigmoid (Logistic)



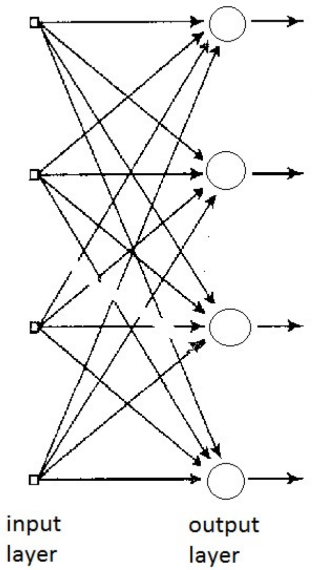
Tanh



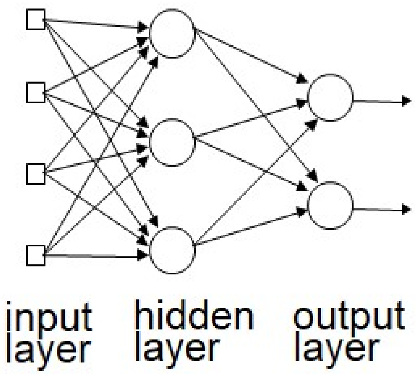
ReLU

Network topology

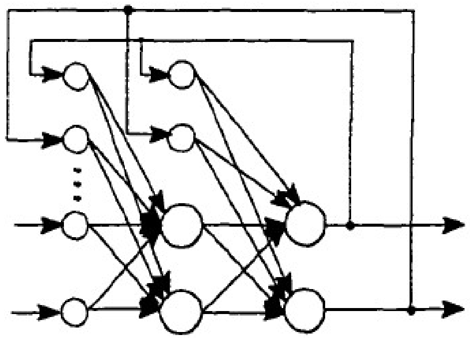
- There are many types of NNs depending on how neurons are connected to each other:
  - neurons are arranged in an ordered hierarchy of layers in which connections are only allowed between neurons in immediately adjacent layers,
  - connections back from later to earlier neurons are allowed,
  - each neuron can connect to every other neuron,
  - …



one-layer feedforward network



two-layer feedforward network



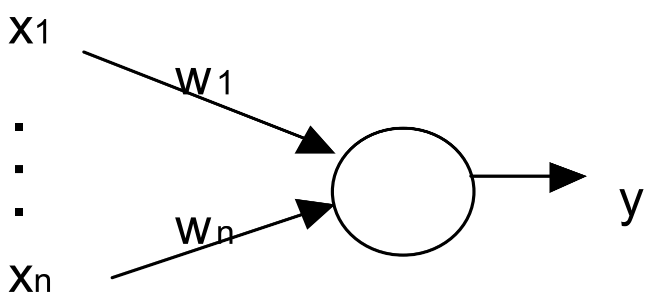
Recurrent network

Training

- A popular learning paradigm is *supervised learning*, which changes the weights by applying a set of labeled *training samples*. Each sample consists of an *input* and the corresponding *target* (or *desired*) output.
- The network is presented with a sample chosen at random from the *training set* and the weights are modified to minimize the error, i.e., the difference between the desired response and the actual response. The entire training set is presented to the network. The training is repeated until the network reaches a state where there are no further significant changes in the weights.
- Therefore, the network learns from the training examples by building an *input-output mapping* for the problem under consideration.

Delta rule

- An example of an error correction rule is the *delta rule* (also called *Widrow-Hoff rule*).



- The delta rule states that the adjustment to be made is

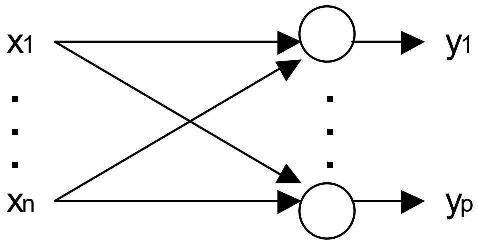
$$\Delta w_i = \eta \delta x_i \quad \text{with} \quad \delta = t - y \quad \text{(error)}$$

where t is the target output, y is the actual output and the real number η is the *learning rate*.

Therefore $$w_i(n+1) = w_i(n) + \Delta w_i(n)$$

- The original delta rule is used for one-layer neural networks with linear output neurons.

- Consider the following network



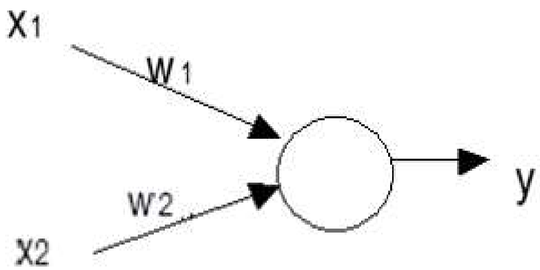
- t_j = target (or desired) output
- y_j = actual output

- Error for the *k*–th training sample

$$E_k = \frac{1}{2}\sum_{j=1}^{p}(t_j - y_j)^2$$

- Overall error on *m* training samples

$$E = \sum_{k=1}^{m} E_k$$



$$y = w_1 x_1 + w_2 x_2$$

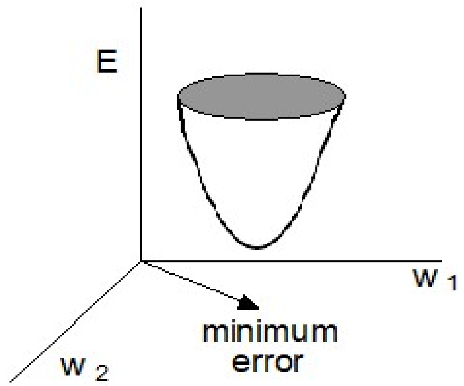
- For a linear neuron (i.e., $f(a) = a$) and a single sample the error is

$$E = \frac{1}{2}(t - a)^2$$

- Expanding gives

$$E = \frac{1}{2}[t^2 - 2ta + a^2] =$$

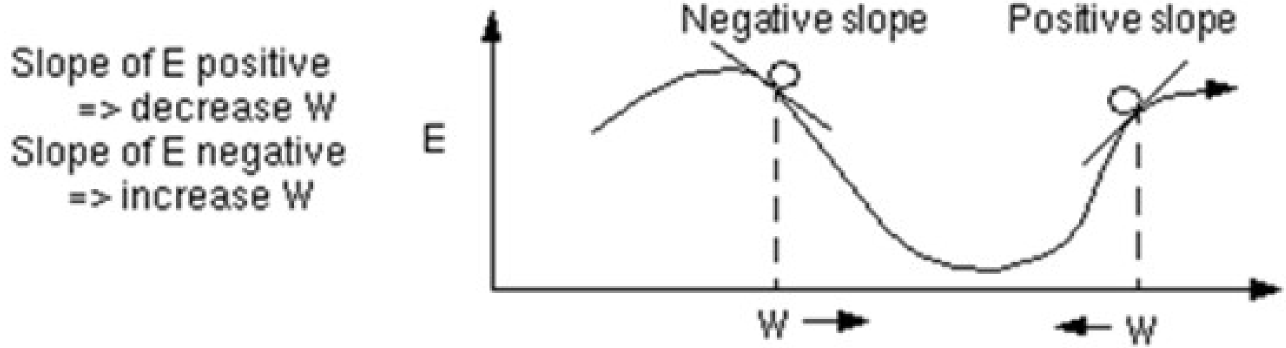
$$= \frac{1}{2}[t^2 - 2t(x_1 w_1 + x_2 w_2) + x_1^2 w_1^2 + 2x_1 w_1 x_2 w_2 + x_2^2 w_2^2]$$



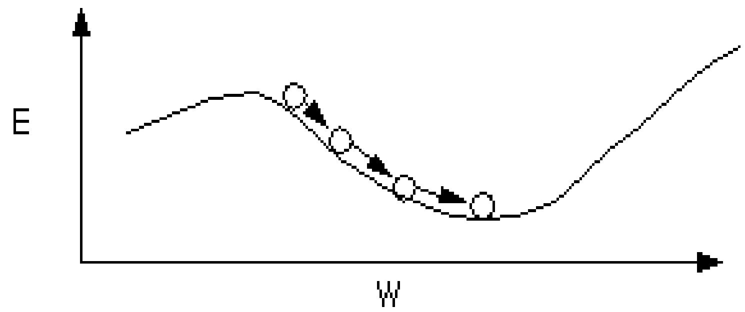
- Before training, the weights start at some random values and therefore the initial state of the network could be anywhere on the error surface.
- During training, the weights are adjusted in a direction towards a lower overall error (i.e., in the direction of the steepest descent of the error surface).

To prove the convergence of the delta rule, we just need to show that it can be expressed as

$$\Delta w_{ij} = -G = -\frac{\partial E}{\partial w_{ij}}$$



By repeating this over and over, we move "downhill" in *E* until we reach a minimum ($G = 0$)



- delta rule

$$\Delta w_{ij} = \eta \delta_j x_i \qquad \delta_j = (t_j - o_j)$$

where t_j is the desired output from neuron j, o_j is the actual output, x_i is the signal coming from neuron i, η is the learning rate and Δw_{ij} is the weight change.

- We deal with a linear unit with the output defined as

$$o_j = \sum_i x_i w_{ij}$$

- Applying the chain rule $$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial o_j} = -\delta_j \qquad \frac{\partial o_j}{\partial w_{ij}} = x_i$$

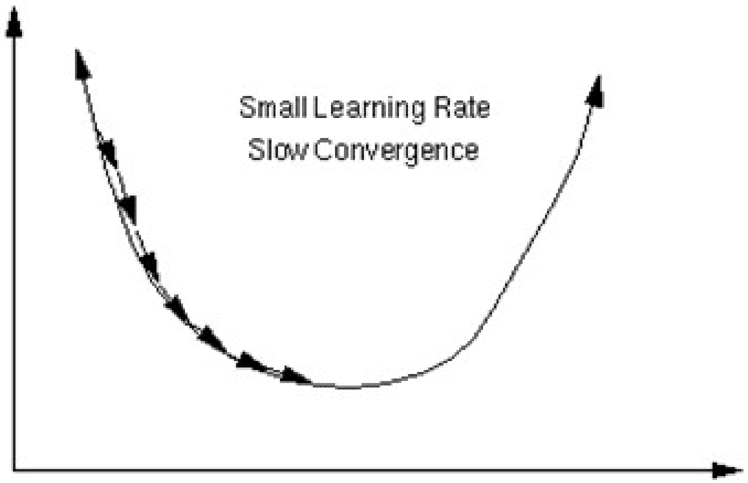
$$-\frac{\partial E}{\partial w_{ij}} = \delta_j x_i$$

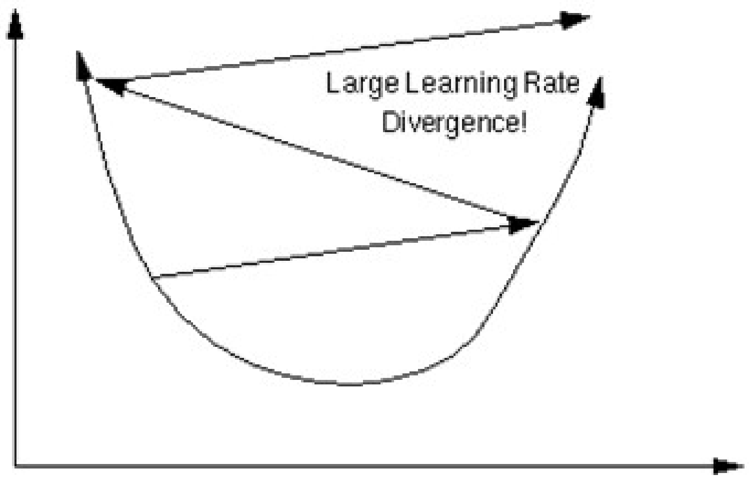
- By inserting the learning rate, the formula for the delta rule is obtained:

$$\Delta w_{ij} = \eta \delta_j x_i$$

The learning rate

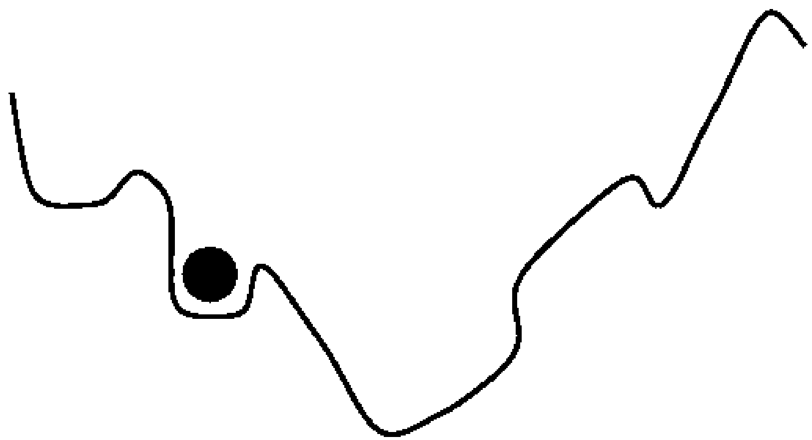
- The learning rate cannot be too low (to avoid too long training times) nor too high (to avoid oscillations around the minimum).
- Typically, the learning rate is chosen experimentally; it can also vary over time, getting smaller as the training algorithm progresses.





Local minima

- For real-world problems, the error surfaces can have several *local minima*, and, during training, the network can get trapped in one of these minima.



Momentum

One technique that can help the network get out of local minima is the use of a *momentum* term. With momentum *m*, the weight update at a given time becomes

$$\Delta w_{ij}(n+1) = \eta \delta_j x_i + m \Delta w_{ij}(n)$$

where $0 < m < 1$ is a new global parameter that must be determined by trial and error.

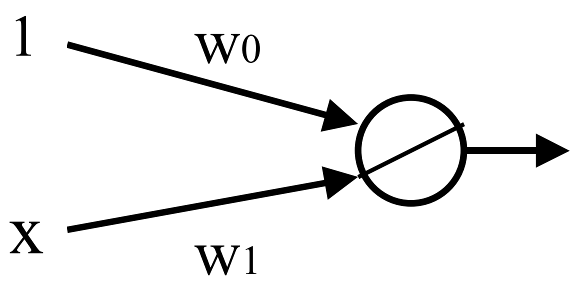
Learning algorithms

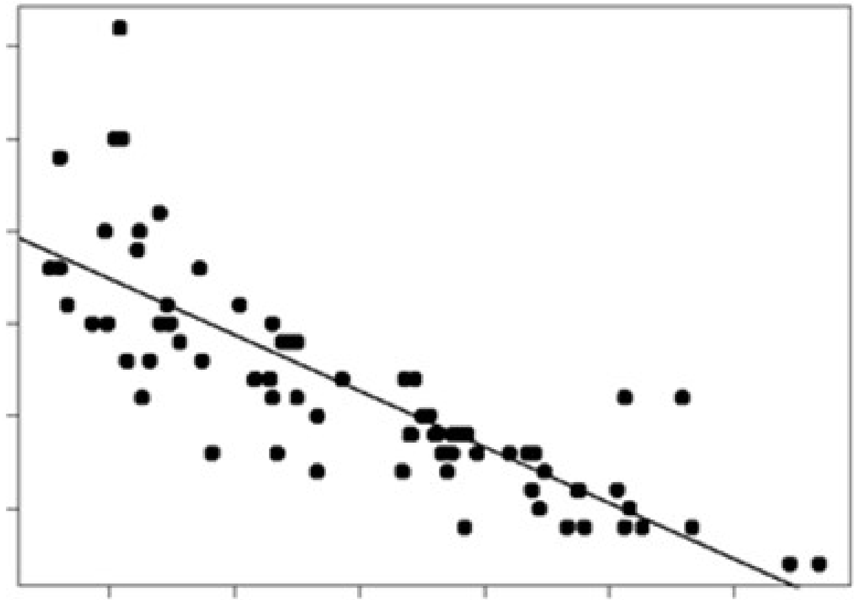
- In *online learning*, the weights are updated immediately after considering each training sample.
- In *batch* (or *off-line*) *learning*, we accumulate the gradient contributions for all samples in the training set before updating the weights. The weights are updated at the end of each epoch.
- In *mini-batch learning*, the weights are updated every n samples, with $n > 1$ but less than the size of the training set.

- Batch learning has the following advantages:
  - it provides an accurate estimate of the gradient vector;
  - it allows the parallelization of the learning process.
- Online learning has some advantages, in particular:
  - it can be used when there is no fixed training set;
  - it requires much less storage space than batch learning;
  - the noise in the gradient can help escape from local minima, if they are not too severe (in fact, the gradient for a single training sample can be considered a noisy approximation of the overall gradient).

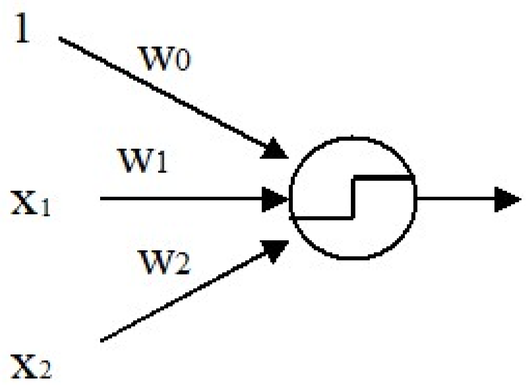
Linear fit

$y = w_1\ x + w_0$





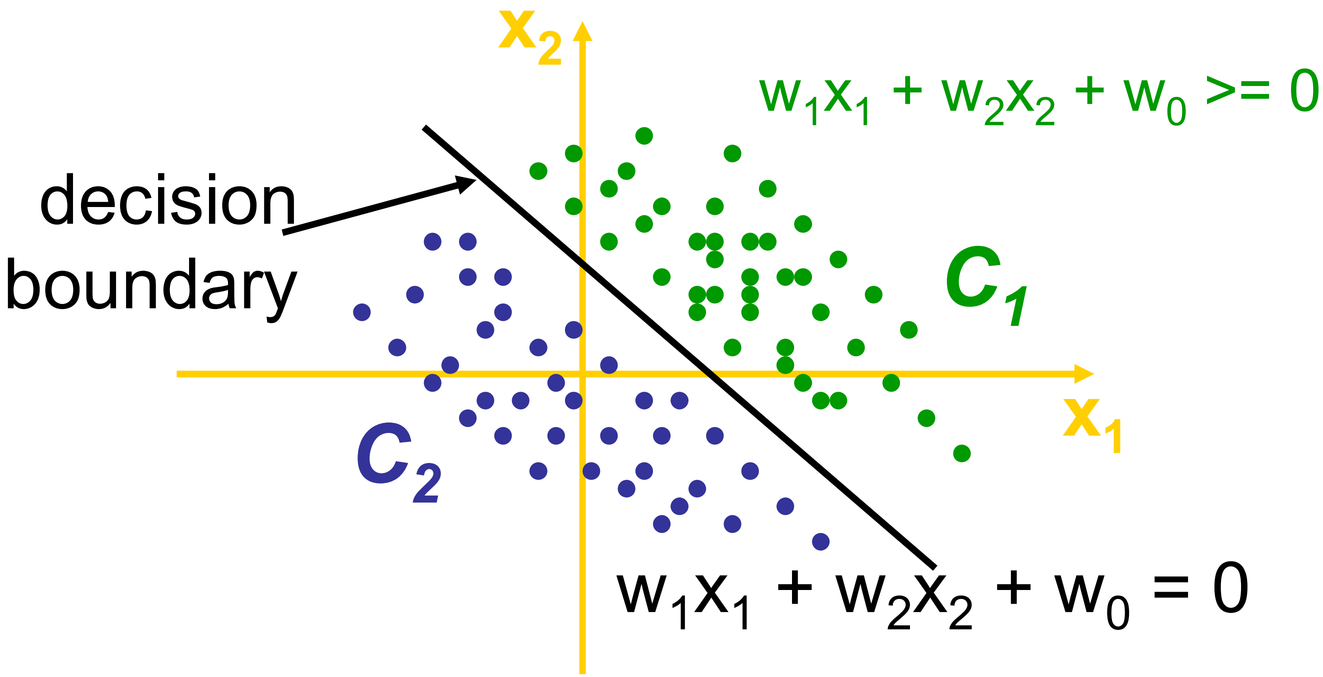
Perceptron

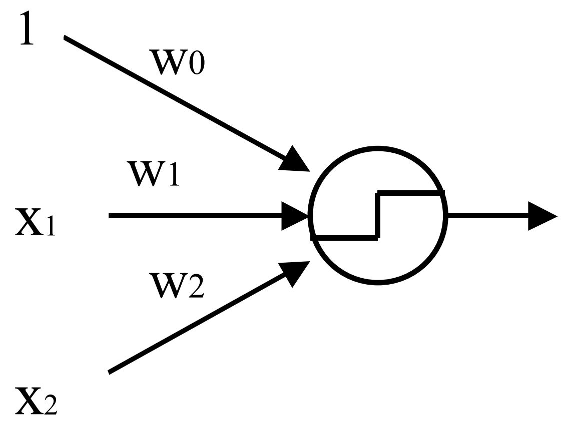


- A perceptron is a single neuron with a hard limiter as activation function (usually, the *sign function* or the *unit step function*). It is used for binary classification.
- Given training examples from classes C_1 and C_2 , the perceptron can be trained to correctly classify the training examples. E.g.,
  - *if the output is +1 then the input is assigned to class C_1*
  - *if the output is -1 then the input is assigned to C_2*

Classification

- The equation below describes a straight line in the input space.
- The line divides the input space into two regions, each corresponding to one class.





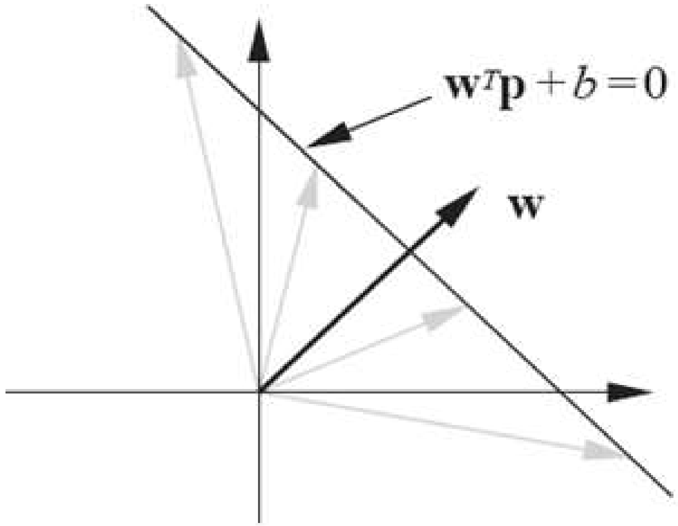
Perceptron: Learning Algorithm

1. initialize the weights (to zero or to a small random value);
2. choose a learning rate η (a number between 0 and 1);
3. until the stop condition is satisfied (e.g., weights don't change):

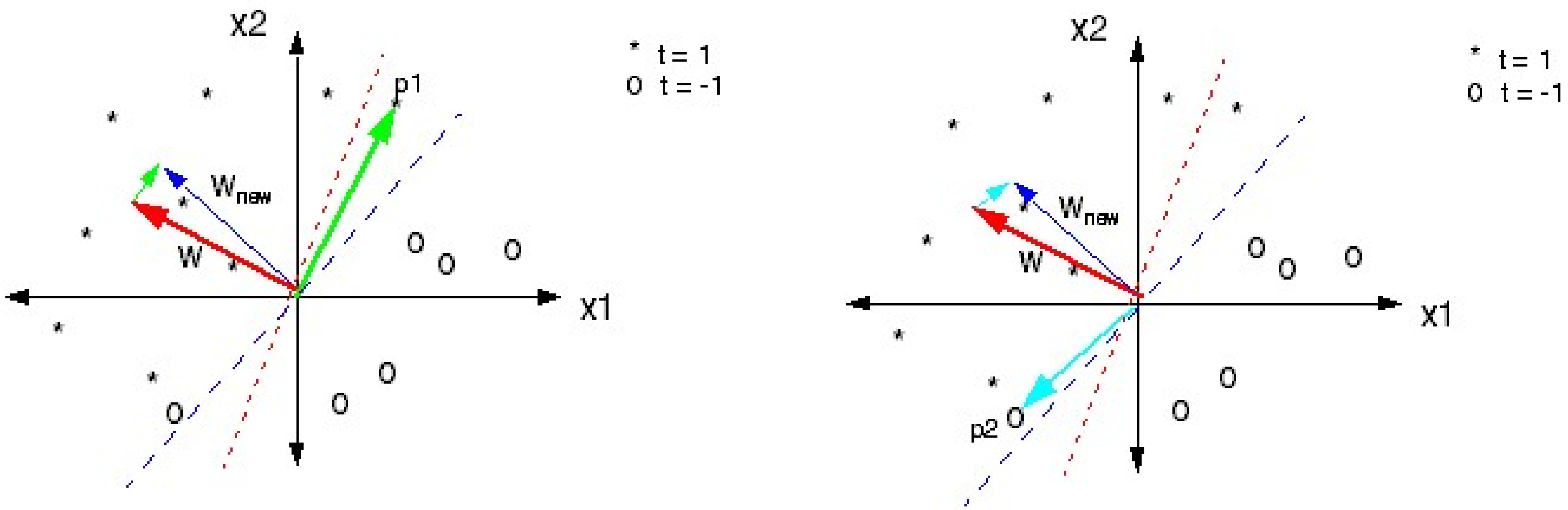
- for each training sample (x, t):
  - calculate the output activation $y = f(w \cdot x)$
  - If $y = t$, do not change the weights
  - If $y \neq t$, update the weights:
    - $w^{new} = w^{old} + \eta\,(t - y)\,x$

Decision boundary

- All points on the decision boundary have the same inner product with the weight vector.
- Therefore, they have the same projection on the weight vector, so they must lie on a line orthogonal to the weight vector.

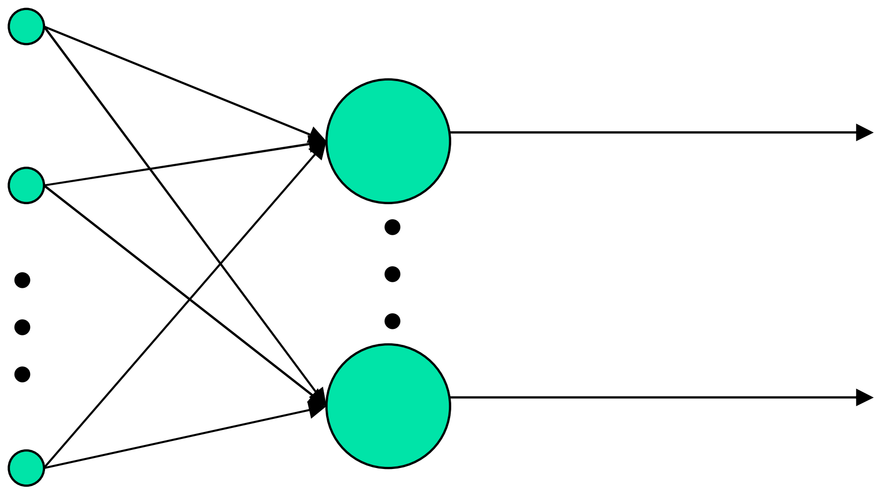


- The red dashed line is the decision boundary.
- We note that both p1 and p2 are classified incorrectly.
- Let us consider what happens when you choose the training sample p1 (or p2) to update the weights.



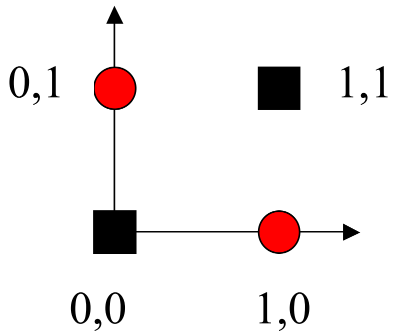
- Suppose we choose p1: p1 has target t=1, so the weight is moved by a small amount in the direction of p1.
- Suppose we choose p2: p2 has target t=-1, so the weight is moved by a small amount in the direction of -p2.
- In both cases, the new boundary (blue dashed line) is better than before.
- *The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps (epochs) if the problem is linearly separable.*

Multiple-neuron perceptron



- Each perceptron has its own decision boundary for classifying the input points into 2 classes.
- A multi-neuron perceptron can perform multi-class classification.

- The perceptron can only solve linear problems.
- The perceptron cannot solve the XOR problem:

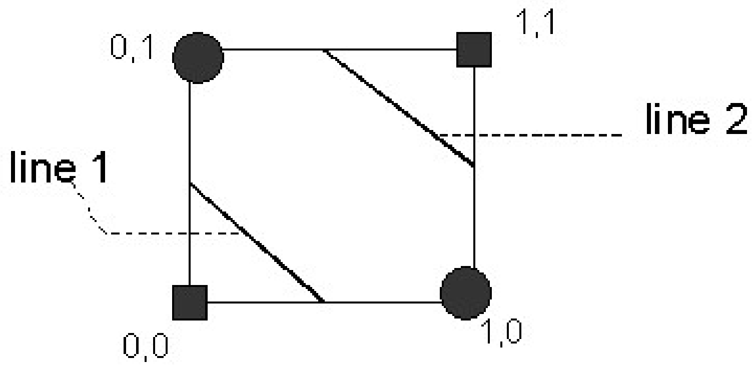


- Possible solutions:
  - use a neuron with a specially defined activation function, e.g.,

$$y = (x_1 - x_2)^2 = \begin{cases} 1 & \text{se } x_1 \neq x_2 \\ 0 & \text{se } x_1 = x_2 \end{cases}$$

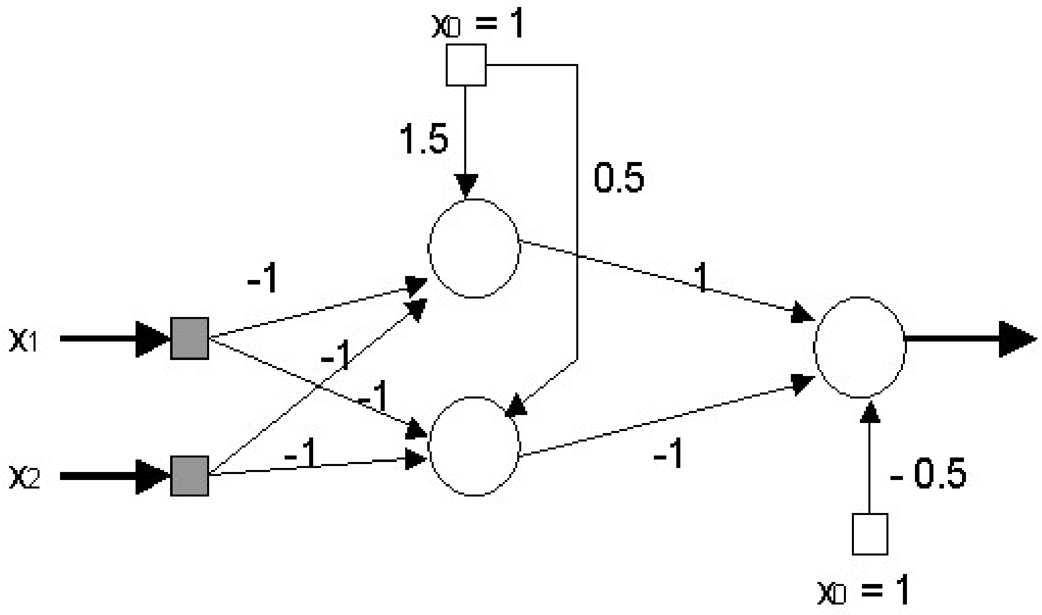
  - introduce *hidden layers*.

The XOR problem solved with two separating lines



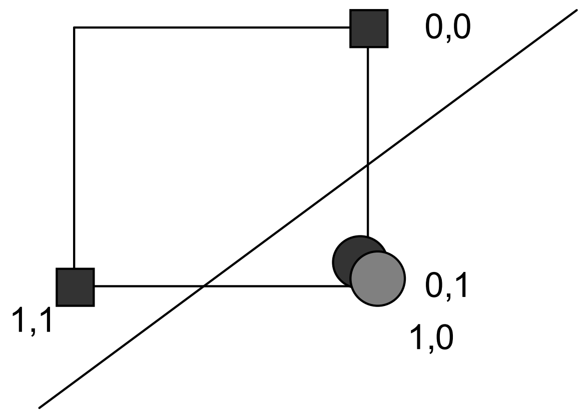
- The network needs two neurons, both fed with two inputs, to represent the two lines, and a third neuron to combine the information from these two lines.

- We use a threshold function.



Effect of the first layer of weights

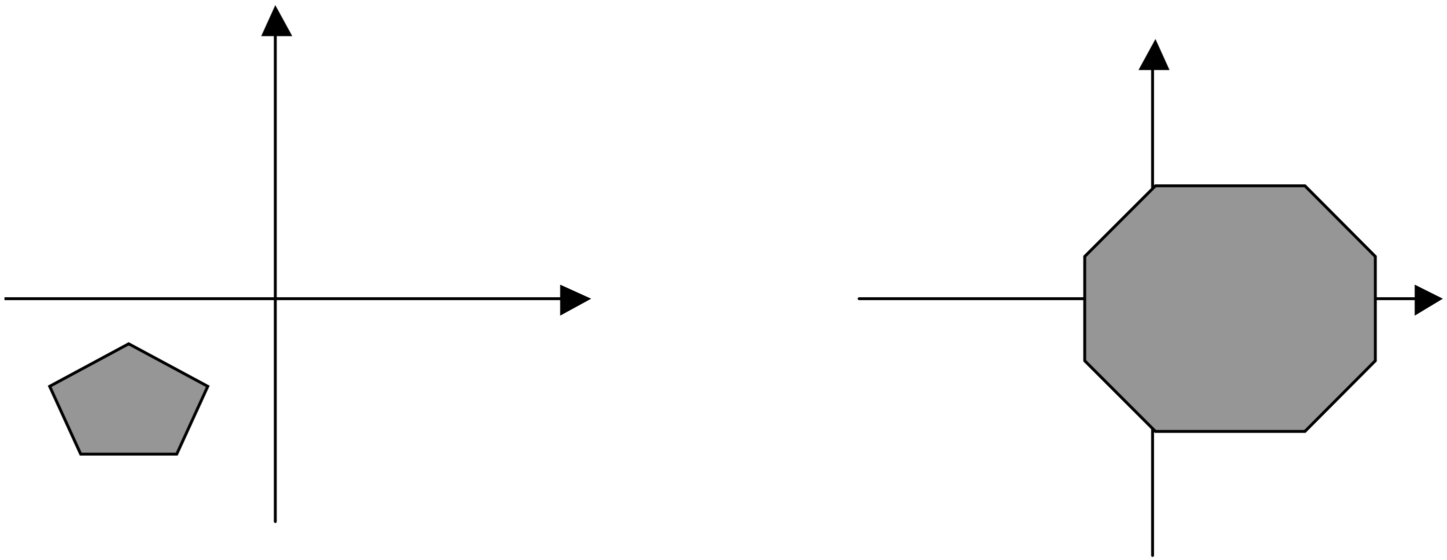
		Net input to hidden layer		Output from hidden layer	
x_1	x_2	unit 1	unit 2	unit 1	unit 2
1	1	-0.5	-1.5	0	0
1	0	0.5	-0.5	1	0
0	1	0.5	-0.5	1	0
0	0	1.5	0.5	1	1



- The inputs to the second layer are linearly separable.

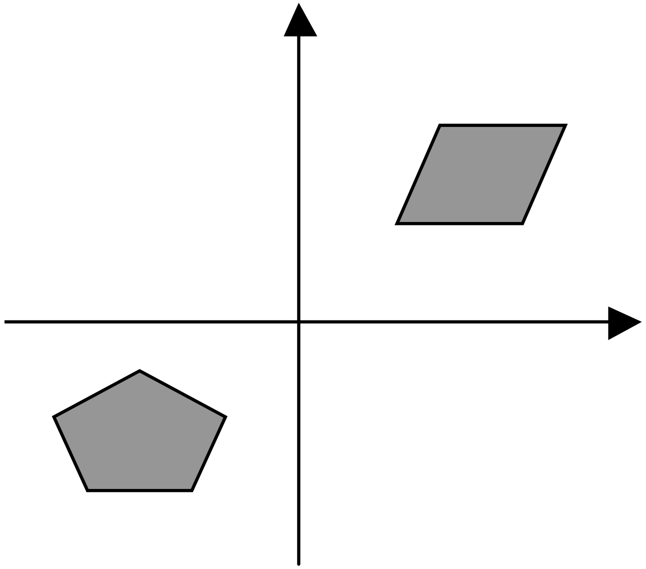
One hidden layer

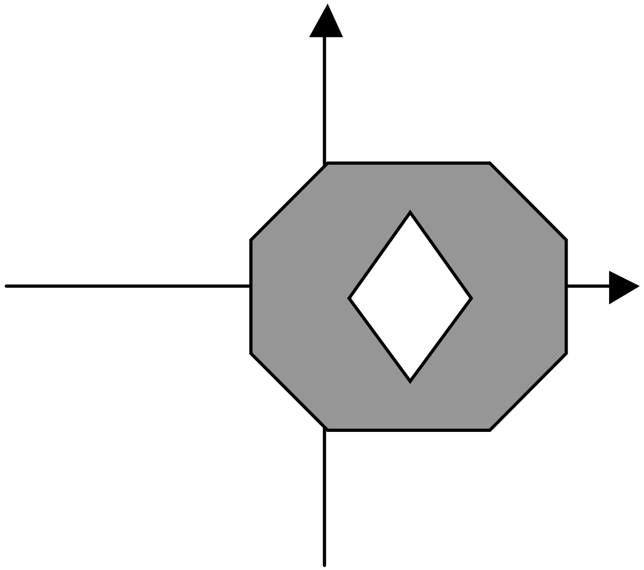
- A network with one hidden layer can model regions with a number of sides at most equal to ($\leq$) the number of hidden neurons.



Two hidden layers

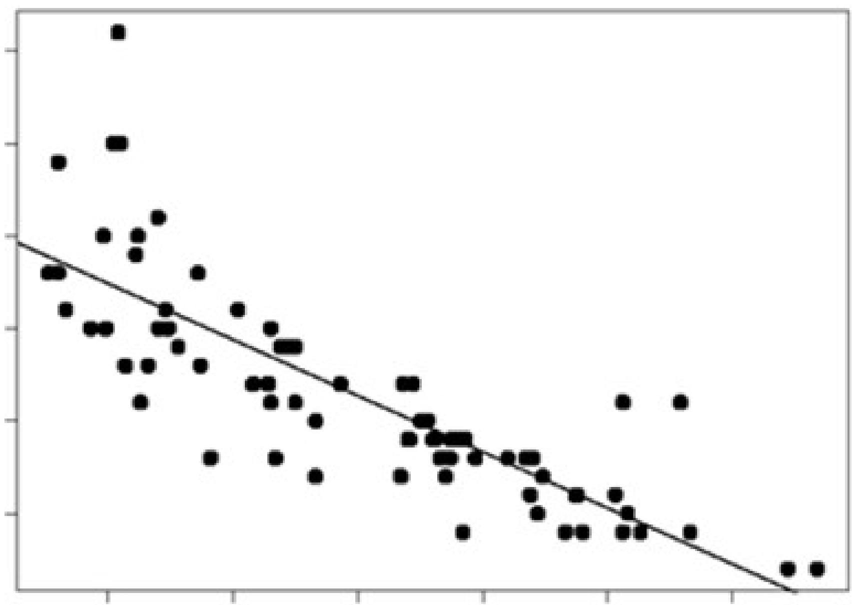
- A network with two hidden layers can model arbitrarily complex decision regions.

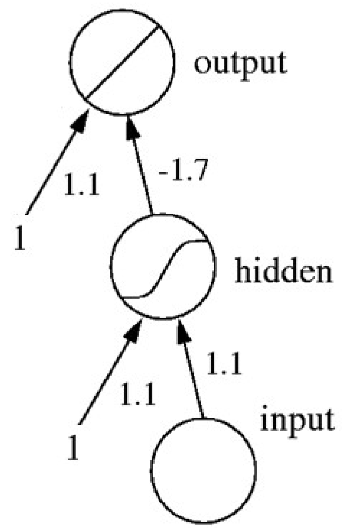




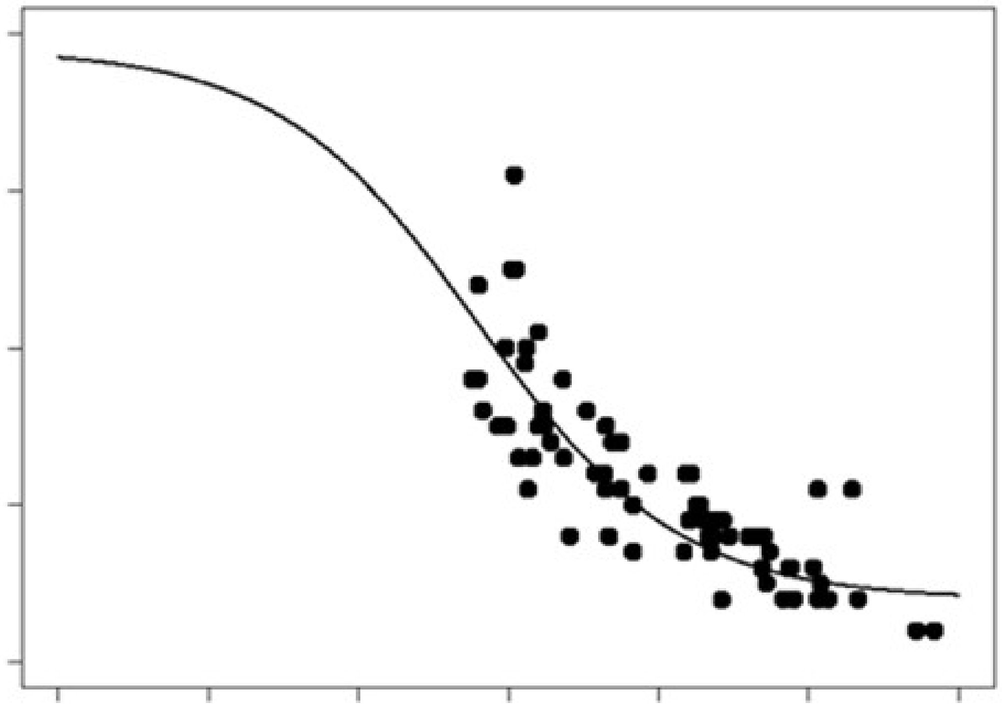
- In practice, most problems are solved with a single hidden layer, sometimes with two.

- Consider again the following scatter plot and the best linear model found by gradient descent. The data are not evenly distributed around the line.
- We can get a better approximation by using a hidden layer, consisting of a single neuron with a tanh function.



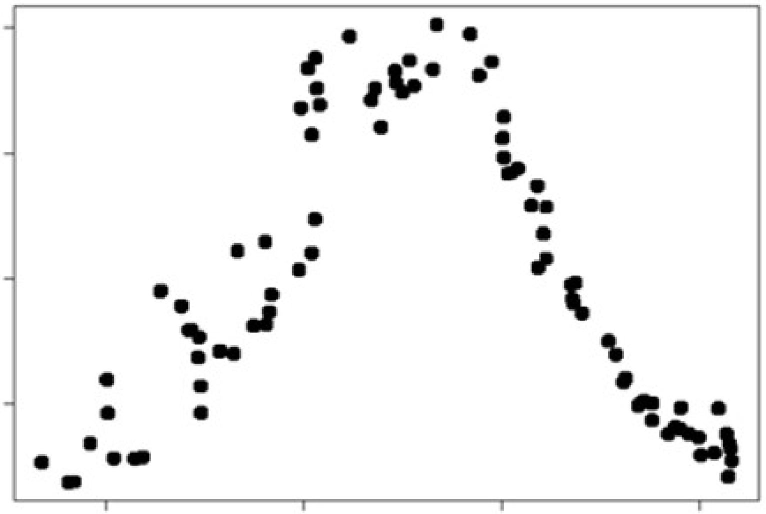


- Once trained, this network approximates the scatter plot as follows:



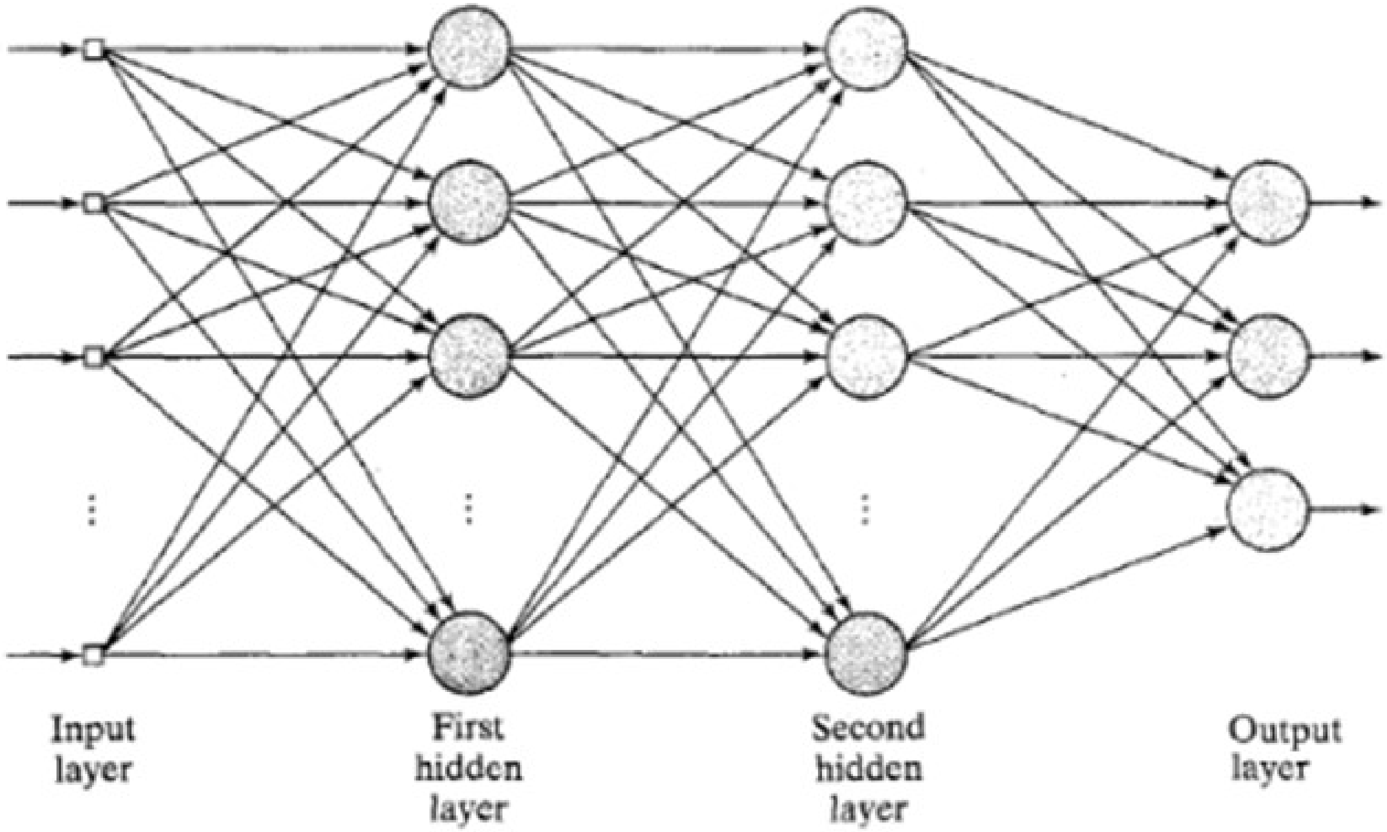
Hidden layers

- What can we do if the data look like this?



- Solution: *increase the number of hidden neurons*.
- Too many neurons in a single hidden layer or too many hidden layers can have a negative effect on the network performance. In general, the best choice is to start with a network that has a reasonable minimum number of hidden neurons, then train it and only if we do not get the desired result, we start increasing the number of hidden neurons or layers.
- A network with one hidden layer and enough hidden neurons can approximate any continuous function with any degree of accuracy.

Multilayer perceptron (MLP)



Multilayer perceptron with two hidden layers

Error Backpropagation

- How can we train a multilayer neural network?
- We do not have the desired values for hidden neurons.
- We use the *error backpropagation algorithm*.
  - Forward activation
  - Calculate the output error
  - Error backpropagation

Some theory

- Backpropagation uses a generalization of the delta rule.
- Error derivative $\dfrac{\partial E}{\partial w_{ij}} = \dfrac{\partial E}{\partial o_j} \dfrac{\partial o_j}{\partial net_j} \dfrac{\partial net_j}{\partial w_{ij}}$, where $net_j = \sum_{i=0}^{n} x_i w_{ij}$
- We define the error for neuron *j* $\delta_j = -\dfrac{\partial E}{\partial net_j}$
- This definition is consistent with the definition given by the original delta rule
$$\delta_j = -\frac{\partial E}{\partial o_j}$$
- In fact, the original delta rule considers linear neurons for which the output is the same as the input ($o_j = net_j$).

$$\delta_j = -\frac{\partial E}{\partial net_j}$$ can be rewritten as $$\delta_j = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

- Since $E_k = \frac{1}{2}\sum_j (t_j - o_j)^2$, we have $\frac{\partial E}{\partial o_j} = -(t_j - o_j)$

- For the activation function *f* the output is

$$o_j = f(net_j)$$

therefore

$$\frac{\partial o_j}{\partial net_j} = f'(net_j)$$

- So $\delta_j = (t_j - o_j) f'(net_j)$

- Since $net_j = \sum_i x_i w_{ij}$

we have $$\frac{\partial net_j}{\partial w_{ij}} = x_i$$

- Then, taking the product of each derivative and substituting it in the initial equation gives $$\frac{\partial E}{\partial w_{ij}} = -(t_j - o_j) f'(net_j) x_i = -\delta_j x_i$$

- Noting that the weight change should be in a direction opposite to the derivative of the error and applying the learning rate, the weight change for a neuron is

$$\Delta w_{ij} = \eta \delta_j x_i$$

- The error δ_j can be calculated for an output neuron, but the error for a hidden neuron is not directly related to the target output.
- However, a hidden neuron can be adapted in proportion to its contribution to the error in the next layer.

- More precisely, the error of a hidden neuron depends on the size of the error of each neuron in the next layer and the strength of the weight connecting both neurons.
- Therefore, for a generic hidden neuron the error is given by

$$\delta_j = f'(net_j) \sum_s \delta_s w_{js}$$

where *s* indexes the layer sending back the error.

- If we use the logistic function as activation function

$$f(net_j) = \frac{1}{1 + \mathrm{e}^{-net_j}}$$

we have

$$f'(net_j) = \frac{\mathrm{e}^{-net_j}}{\left(1 + \mathrm{e}^{-net_j}\right)^2}$$

$$= \frac{1}{1 + \mathrm{e}^{-net_j}} \left(1 - \frac{1}{1 + \mathrm{e}^{-net_j}}\right)$$

$$= f(net_j)[1 - f(net_j)]$$

Backpropagation algorithm (online)

- 1. Initialize the weights to small random values
- 2. For the first input sample calculate the output of all neurons

$$o_j = \frac{1}{1+e^{-net_j}}$$

- 3. For each output neuron calculate its error:

$$\delta_j = (t_j - o_j) f'(net_j) = (t_j - o_j) o_j (1 - o_j)$$

- 4. For all hidden layers (from output to input) calculate the error for each neuron:

$$\delta_j = f'(net_j)\sum_s \delta_s w_{js} = o_j(1-o_j)\sum_s \delta_s w_{js}$$

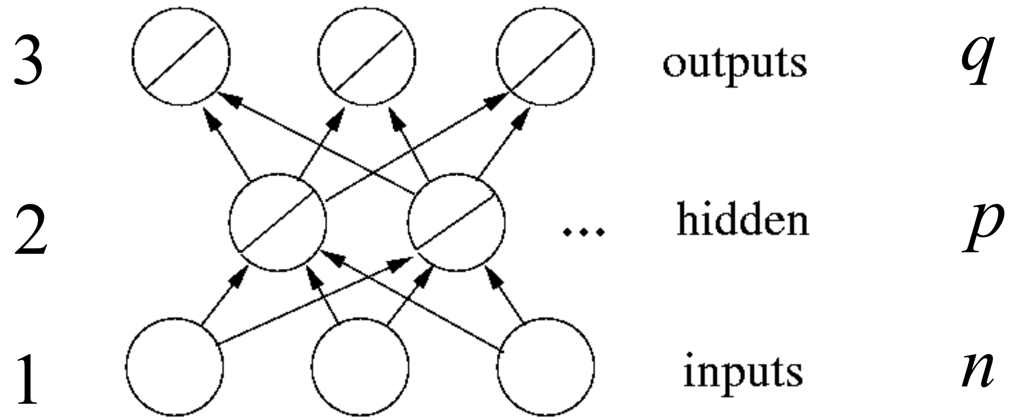
- 5. For all layers, update the weights for each neuron:

$$\Delta w_{ij}(n+1) = \eta \delta_j o_i + \alpha \Delta w_{ij}(n)$$

- 6. Repeat from step 2 for all training samples
- 7. Calculate the error on the training set: if the error falls within the desired tolerance, the algorithm is said to have *converged*; otherwise continue with the training

Limitation of a network with linear units

- A multilayer network with linear activation functions will only be able to solve a problem that a single-layer network can solve.



- n input neurons, p hidden neurons, q output neurons
- W_{21} $(p \times n)$, W_{32} $(q \times p)$ => $z = W_{32}Y = W_{32}W_{21}X = WX$
  with $W = W_{32}W_{21}$ $(q \times n)$

- For multilayer networks, therefore, a nonlinear activation function is required.

Creating a training set

- Creating the training set is a crucial step.
- This includes raw data collection, data analysis using statistical techniques, variable selection, and data preprocessing (for example, normalizing inputs and targets in order to make all variables comparable to each other).
- Another important action to take for each input variable is to identify and remove the *outliers*.
- Two other typical actions are the management of *missing values* of a given variable and the management of *non-numeric data*.
- Finally, the problem of *unbalanced data* must be managed.

How many training examples?

- The training set must be a representative sample of the data the network will work on.
- Large training sets reduce the risk of undersampling the underlying function. If the training set is too small, the network can learn it perfectly but fail in the final application.
- In practice, the number of cases required for training depends on several factors, such as the size of the network and the distribution of inputs and targets. In particular, a large network usually requires more training data than a small one. In fact, there is no specific recipe. Some heuristic guidelines say there should be 5 to 10 training samples for each weight.

Training stopping condition

- Training stops when
  - a certain number of epochs pass (an *epoch* is the presentation of all the training data),
  - the error reaches an acceptable level,
  - the error stops improving.

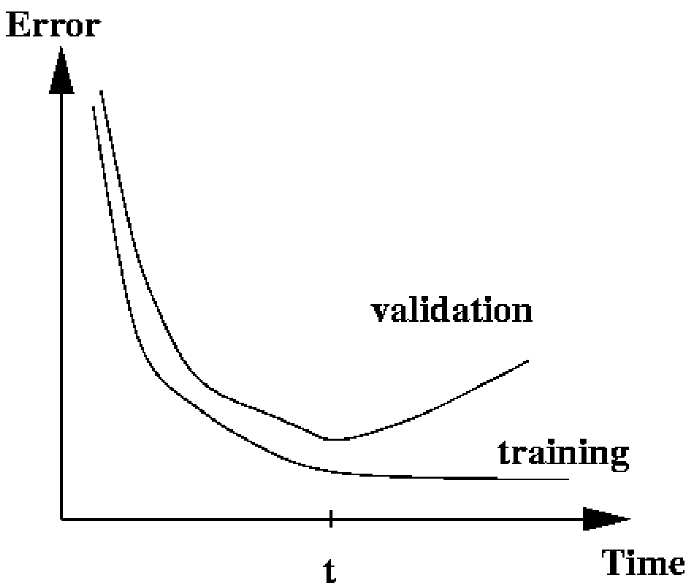
Training and testing

- During training, a neural network changes its weights repeatedly, epoch after epoch, to improve its performance.
- After training, we measure the *generalization* capacity of the network by testing it with an independent data set. This set, called the *test set*, consists of pairs (input, desired output) never seen by the network during training.
- During the test, the network passes the test samples forward through itself and calculates a performance index, such as the mean squared error, without changing the weights.
- Therefore, the available data are divided into two parts, for example, two thirds for training and one third for testing.

Early stopping (1)

In order to prevent inappropriate memorization of input data (also called *overtraining*, *overlearning* or *overfitting*), we need a third independent data set, called *validation set*, to be used during training to verify that the network is not 'overlearning'.

We can plot errors (e.g., MSE) on training and validation sets.



Both errors typically fall quickly at the start of training as the network moves its weights away from their original random positions. Over time, both curves become flatter. Typically, the training set error continues to decrease, but the validation set error eventually begins to increase.

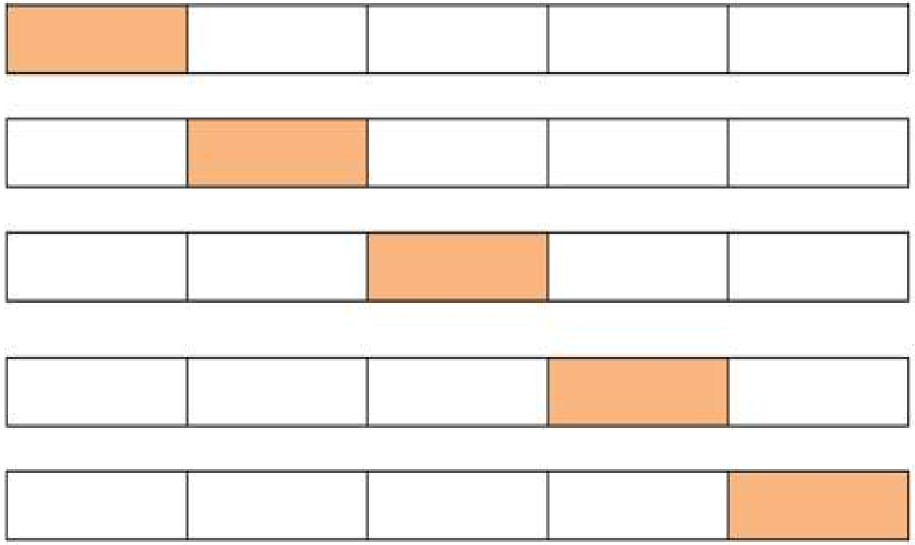
Early stopping (2)

- This increase shows that the network has stopped learning what training samples have in common with validation samples and has begun to learn meaningless differences. This overfitting of the training data harms the network's ability to generalize, as it merely memorizes the noise in the training data. In other words, as the training set error decreases, the test set error increases.
- For the best generalization, training stops when the validation-set error reaches its lowest point (*early stopping*).

Network’s size

- A large network has many weights that can be used to model a function. This can be an advantage for complex functions if we have sufficient training data. Otherwise, too many weights can be a disadvantage, because the neural network can use them to memorize training data. On the other hand, if the network is too small, it cannot learn the problem at all.
- The key is to find a network large enough to learn how to solve the specific problem but small enough to generalize well. The best choice is the network with the minimum number of weights needed to accurately process the test data. A good strategy is to start with a few hidden neurons and increase the number while monitoring the generalization by validating at each epoch.

K-fold cross-validation



- leave-one out
- stratified
- iterated (repeated)

Preventing overfitting (1)

- **Early stopping**
  Use a validation set.
- **Gathering more training data**
  Increase the size of the training data.
- **Performing dataset augmentation**
  Artificially increase the size of the dataset by introducing different types of transformations or distortions of the available data.
- **Reducing the capacity of the network**
  Reduce the size of the network.

Preventing overfitting (2)

- **Adding weight regularization**

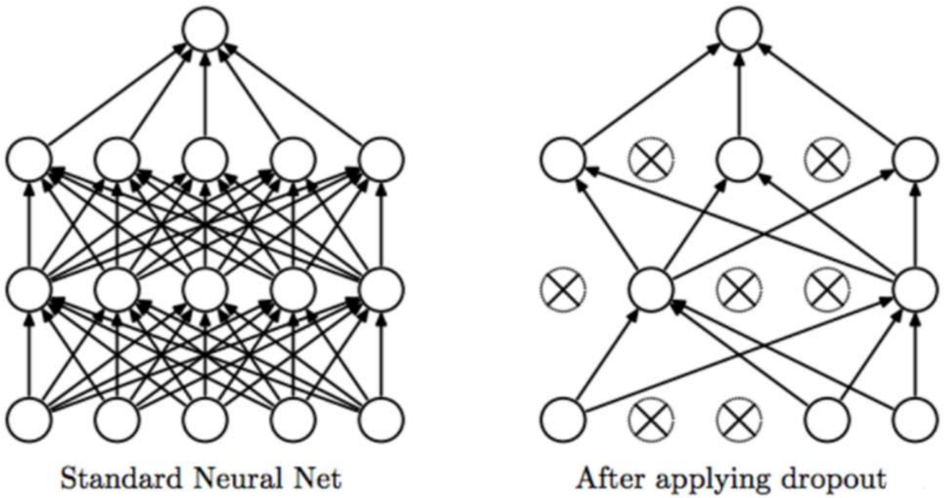
  Force the weights to take small values by adding a penalty that penalizes large weights to the loss function.

  L2 regularization: $J_{Reg} = \frac{1}{2}\lambda \sum_i w_i^2$

  L1 regularization: $J_{Reg} = \lambda \sum_i |w_i|$

  E.g., $J_{Tot} = J + J_{Reg}$ $\quad \frac{\partial J_{Reg}}{\partial w_k} = \frac{\partial}{\partial w_k}\left(\frac{1}{2}\lambda \sum_i w_i^2\right) = \lambda \cdot w_k$

- **Dropout**



Multi-class classification

Softmax function $y_j = \frac{e^{net_j}}{\sum_k e^{net_k}}$

Cross-entropy $H(p,q) = -\sum_x p(x) \log(q(x))$ p(x) = true distribution, q(x) = estimated distribution

When used as loss function

$$L = -\sum_{i=1}^{C} t_i \log(y_i) = -\mathrm{t} \cdot \log(y)$$

If *t* is a one-hot vector:

$$\mathrm{t} = [0\ 0\ \ldots\ \overset{k}{1}\ \ldots\ 0] \qquad L = -\log(y_k)$$

k = correct class, y_k = estimated probability for the correct class

minimum value of L: 0, maximum value: infinite