WEIGHTS IN NEURAL NETWORKS

There are three approaches to the computation of weights in neural networks:

* Weights are directly computed from the problem description without going through a training phase. This method is used in e.g. Hopfield memories.
* Weights are adjusted during a training phase in order to reproduce as closely as possible the behavior given by input-output pairs. This is called *supervised learning.* It is e.g. used for feedforward networks.
* Weights are adjusted during a training phase in order to cluster data given in a training set. No desired output is known; the network tries to find similarities in the input patterns. This is called *unsupervised learning.* It is e.g. used in Kohonen networks.

NEURON MODEL WITH BIAS

* Neuron model with N inputs as considered until now:

$$y = g\left(\sum_{j=1}^{N} w_j x_j\right)$$

* In many cases an extra *bias* input with value 1 is used; it is multiplied by weight w_0:

$$y = g\left(w_0 + \sum_{j=1}^{N} w_j x_j\right)$$

McCULLOCH-PITTS NEURON MODEL

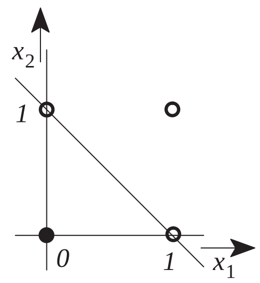
* Proposed in 1943.
* A special case of a neuron with bias where all weight values are ± 1 and the activation function g is the step function S ($S(v) = 1$ if $v \geq 0$ and $S(v) = 0$ otherwise).
* Single neurons can be used to build Boolean functions. Examples:
  + NOT: $y = S(-x_1)$.
  + 2-input OR: $y = S(-1 + x_1 + x_2)$.
  + 2-input AND: $y = S(-2 + x_1 + x_2)$.
  + 3-input AND: $y = S(-3 + x_1 + x_2 + x_3)$.
* Any logic function can be constructed by combining elementary functions built from single neurons (because any function can be constructed from ORs, ANDs and NOTs).
* Not a serious alternative for implementation with logic gates.

GEOMETRIC INTERPRETATION (1)

* Consider the condition for which the induced local field of the 2-input OR becomes 0:

$$-1 + x_1 + x_2 = 0.$$

* The equation separates the points where the function is 1 from the points where the function is 0.
* If such a separation is possible, the points are called *linearly separable.* The points of the two-input XOR function are e.g. *not* linearly separable.
* Note that the bias input allows for lines that do not pass through the origin.



GEOMETRIC INTERPRETATION (2)

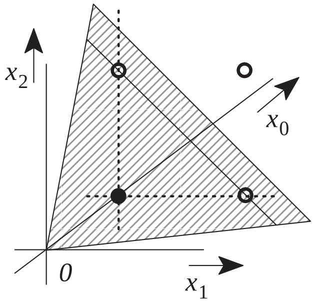
* It is convenient to see the bias as part of the input vector and to consider from now on the *extended input vector:*

$$\mathbf{x} = (1, x_1, \ldots, x_N)^T$$

* Similarly, it is convenient to define an *extended weight vector:*

$$\mathbf{w} = (w_0, w_1, \ldots, w_N)^T$$

* The points are now separated by a *hyperplane* described by $\mathbf{x} \cdot \mathbf{w} = 0$. Note that the hyperplane always passes through the origin.



ROSENBLATT PERCEPTRON

* Proposed in 1958.
* It differs from the McCulloch-Pitts neuron in the fact that real-valued inputs and weights are allowed. The activation function is still the step function. The neuron's behavior is described by ($\mathrm{sgn}(v) = 1$ if $v \geq 0$ and $\mathrm{sgn}(v) = -1$ otherwise):

$$y = \mathrm{sgn}(\mathbf{w} \cdot \mathbf{x})$$

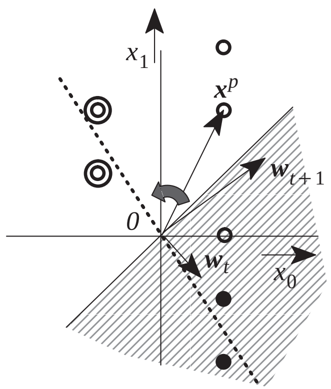
* It can implement any function $R^N \to \{-1, 1\}$ provided that the input vectors are linearly separable.
* The desired behavior is given by pairs $(\mathbf{x}^p, d^p)$, $p = 1, \ldots, Q$, $d^p = \pm 1$.
* The issue is to find the specification of the hyperplane or, equivalently, the correct extended weight vector $\mathbf{w}$ such that $d^p(\mathbf{w} \cdot \mathbf{x}^p) \geq 0$ for all $p = 1, \ldots, Q$.

PERCEPTRON LEARNING RULE

* Start with any weight vector $\mathbf{w}_0$.
* Process the desired input-output pairs:

$$\text{if } d^p \mathbf{w}_t \cdot \mathbf{x}^p \leq 0,$$
$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta d^p \mathbf{x}^p.$$

* It can be proven that the repetitive application of the learning rule will lead to a weight vector $\mathbf{w}^*$ that correctly specifies the separating hyperplane.



CONVERGENCE PROOF (1)

* Assumptions: the vectors $\mathbf{x}^p$ are normalized, i.e. $\|\mathbf{x}^p\| = 1$; the vector $\mathbf{w}^*$ is normalized as well. The choice for the learning rate: $\eta = 1$.
* Suppose that the weight vector is corrected for some $\mathbf{x}^p$:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + d^p \mathbf{x}^p$$

* The angle ρ between $\mathbf{w}_{t+1}$ and $\mathbf{w}^*$ is found from:

$$\cos\rho = \frac{\mathbf{w}^* \cdot \mathbf{w}_{t+1}}{\|\mathbf{w}^{t+1}\|}$$

* Consider the numerator:

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} = \mathbf{w}^* \cdot (\mathbf{w}_t + d^p \mathbf{x}^p)$$
$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} = \mathbf{w}^* \cdot \mathbf{w}_t + d^p \mathbf{w}^* \cdot \mathbf{x}^p$$
$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_t + \delta$$

where $\delta = \min_{p=1}^{Q} d^p \mathbf{w}^* \cdot \mathbf{x}^p$.

* Note that $\delta > 0$ because $\mathbf{w}^*$ correctly separates the points $\mathbf{x}^p$.
* By induction:

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_0 + (t+1)\delta$$

CONVERGENCE PROOF (2)

* Consider the denominator:

$$\|\boldsymbol{w}^{t+1}\|^2 = (\boldsymbol{w}_t + d^p\boldsymbol{x}^p)\cdot(\boldsymbol{w}_t + d^p\boldsymbol{x}^p)$$
$$\|\boldsymbol{w}^{t+1}\|^2 = \|\boldsymbol{w}_t\|^2 + 2d^p\boldsymbol{w}_t\cdot\boldsymbol{x}^p + \|\boldsymbol{x}^p\|^2$$

* Because $d^p\boldsymbol{w}_t\cdot\boldsymbol{x}^p \le 0$ (otherwise no correction would be necessary) and $\|\boldsymbol{x}^p\|^2 = 1$:

$$\|\boldsymbol{w}^{t+1}\|^2 \le \|\boldsymbol{w}_t\|^2 + 1$$

* And by induction:

$$\|\boldsymbol{w}^{t+1}\|^2 \le \|\boldsymbol{w}_0\|^2 + t + 1$$

CONVERGENCE PROOF (3)

* Combining the results for the numerator and denominator:

$$\cos\rho \ge \frac{\boldsymbol{w}^*\cdot\boldsymbol{w}_0 + (t+1)\delta}{\sqrt{\|\boldsymbol{w}_0\|^2 + t + 1}}$$

* This expression grows to infinity as $t \to \infty$. However, $\cos\rho$ cannot grow larger than 1. The conclusion is that the algorithm converges to a solution after a finite number of time steps.

SPEEDING UP CONVERGENCE

* The learning rule was:

$$\text{if } d^p\boldsymbol{w}_t\cdot\boldsymbol{x}^p \le 0,\ \boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \eta d^p\boldsymbol{x}^p.$$

* Modify the update of the weights to (ϵ is a small positive constant):

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \frac{-d^p\boldsymbol{w}_t\cdot\boldsymbol{x}^p + \epsilon}{\|\boldsymbol{x}^p\|^2} d^p\boldsymbol{x}^p.$$

* The rule guarantees that the error due to $\boldsymbol{x}^p$ is corrected in one step.

$$d^p\boldsymbol{w}_{t+1}\cdot\boldsymbol{x}^p = d^p\left(\boldsymbol{w}_t + \frac{-d^p\boldsymbol{w}_t\cdot\boldsymbol{x}^p + \epsilon}{\|\boldsymbol{x}^p\|^2} d^p\boldsymbol{x}^p\right)\cdot\boldsymbol{x}^p = \epsilon > 0.$$

* So, no weight correction is necessary due to $\boldsymbol{x}^p$ at time $t+1$.

SOLUTION BY LINEAR PROGRAMMING (1)

Remember the linear-programming formulation of optimization problems.

* Given: matrix A vectors $\boldsymbol{b}, \boldsymbol{c}$ (constants) and the vector $\boldsymbol{x}$ (unknowns).

Standard form:

* Minimize or maximize: $\boldsymbol{c}^T\boldsymbol{x}$.
* Subject to: $A\boldsymbol{x} \le \boldsymbol{b}$, $\boldsymbol{x} \ge \boldsymbol{0}$.

* Solvable in polynomial time by *ellipsoid* algorithm; in practice better performance with *simplex* algorithm (exponential time complexity).

SOLUTION BY LINEAR PROGRAMMING (2)

* The training set of the Rosenblatt perceptron contains the pairs $(\mathbf{x}^p, d^p)$, $p = 1, \dots, Q$, $d^p = \pm 1$. Each pair can be translated into a *linear constraint:*

$$w_0 + x_1^p w_1 + \dots + x_N^P w_N > 0 \text{ if } d^p = 1.$$

$$w_0 + x_1^p w_1 + \dots + x_N^P w_N < 0 \text{ if } d^p = -1.$$

* Multiplying the equations by -1 where appropriate and applying the substitution $w_i = z_{2i-1} - z_{2i}$, $i = 1, \dots, N$ and $z_{2i-1}, z_{2i} > 0$, one can translate all constraints to a form suitable for linear programming:

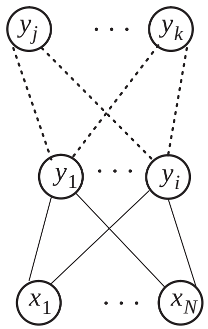
$$A\mathbf{z} \leq \mathbf{0},\ \mathbf{z} \geq \mathbf{0}.$$

* The cost function is not relevant. The constraints, and therefore also perceptron learning, can be solved in polynomial time.

MULTILAYER PERCEPTRONS

* A Rosenblatt perceptron can discriminate between two halves of the input space (the weight vector specifies a separating hyperplane).
* Consider the following feedforward multilayer neural network:
  + The first layer consists of Rosenblatt perceptrons and compute different bisections of the input space.
  + The next layers compute Boolean combinations of the outputs of the first layer, e.g. using McCulloch-Pitts neurons.
* Such a feedforward multilayer network can be configured such that any function $R^N \rightarrow \{-1, 1\}$ can be computed provided that the network contains a sufficient number of neurons in the appropriate layers.

MULTILAYER PERCEPTRON TERMINOLOGY



Output layer

One or more *hidden* layers

Input layer

* Normally there will be fewer neurons in the hidden layer(s) than inputs in order to achieve some kind of "data compression".

TRAINING MULTILAYER PERCEPTRONS

* The training problem for a multilayer network is much more complex than than the training of a single neuron. Some important issues:
  + What complexity should the network have? How many layers, how many neurons per layer?
  + Too few neurons makes the network unable to learn the desired behavior. Too many neurons increases the complexity of the learning algorithm.
  + A desired property of a neural network is its ability to generalize from the training set. If there are too many neurons, there is the danger of overfitting: the network gives the desired output for inputs in the training set, but shows "wild" behavior for inputs not in the training set.
  + How should the training be performed? Does there exist an effective algorithm?

BACKPROPAGATION ALGORITHM (1)

* One of the most successful training algorithms for multilayer feedforward networks is the *backpropagation algorithm*.
* It is a *gradient-descent* method, which implies that the functions describing the neural network should be differentiable. This especially means that the activation functions should be differentiable.
* Activation functions that are often used are the *sigmoid* function and the *logistic* function. They vary between -1 and 1 and, respectively 0 and 1.
* *Sigmoid function:*

$$f(v) = \frac{1 - e^{-\alpha v}}{1 + e^{-\alpha v}}$$

* *Logistic function:*

$$g(v) = \frac{1}{1 + e^{-\alpha v}}$$

BACKPROPAGATION ALGORITHM (2)

* A neuron output will be indicated by y_i, $i = 1, \ldots, M$.
* The neurons belonging to the output layer have indices $i \in C$.
* The training set is given by pairs $(\boldsymbol{x}^p, \boldsymbol{d}^p)$ where $\boldsymbol{x}^p$ has dimension N and $\boldsymbol{d}^p$ has dimension $|C|$.
* The output of a neuron with index i for an input pattern $\boldsymbol{x}^p$ is y_i^p.
* The error that the network makes for an input pattern $\boldsymbol{x}^p$ is:

$$\mathcal{E}^p = \frac{1}{2} \sum_{k \in C} \left[e_j^p \right]^2;\ e_j^p = d_j^p - y_j^p.$$

* The algorithm to be presented performs the training on a pattern-by-pattern basis. So, the error measure given above is sufficient. More sophisticated algorithms may use an error measure based on the entire training set before updating the weight.

BACKPROPAGATION ALGORITHM (3)

* The main idea of the algorithm is:
  + Apply some input pattern $\boldsymbol{x}^p$ to the network's input and propagate its effects forward to the output neurons.
  + Calculate the error backward from outputs to inputs to determine the error at each neuron.
  + Update the weights of each neuron based on the errors.
* Consider the sensitivity of an output neuron for its weights:

$$\frac{\partial \mathcal{E}^p}{\partial w_{ji}} = \frac{\partial \mathcal{E}^p}{\partial e_j^p} \frac{\partial e_j^p}{\partial y_j^p} \frac{\partial y_j^p}{\partial v_j^p} \frac{\partial v_j^p}{\partial w_{ji}}$$

* v_j^p is the induced local field of neuron j for input $\boldsymbol{x}^p$.

BACKPROPAGATION ALGORITHM (4)

* The different partial derivatives are straightforward to compute:

$$\frac{\partial \mathcal{E}^p}{\partial e_j^p} = e_j^p;\ \frac{\partial e_j^p}{\partial y_j^p} = -1;\ \frac{\partial y_j^p}{\partial v_j^p} = g'(v_j^p);\ \frac{\partial v_j^p}{\partial w_{ji}} = y_i^p.$$

* The weights can be updated as:

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial \mathcal{E}^p}{\partial w_{ij}},\ \text{or: } \Delta w_{ji} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ji}} = \eta e_j^p g'(v_j^p) y_i^p = \eta \delta_j^p y_i^p.$$

* δ_j^p is called the *local gradient.* So, the value for weight update depends on the local gradient and the input values of the neuron on which the weight is applied.

BACKPROPAGATION ALGORITHM (5)

* For neurons in hidden layers $\Delta w_{ji} = \eta \delta_j^p y_i^p$ still holds. Only the local gradient is more difficult to derive. Consider first the layer just before the output layer.

$$\delta_j^p = -\frac{\partial \mathcal{E}^p}{\partial y_j^p}\frac{\partial y_j^p}{\partial v_j^p} = -\frac{\partial \mathcal{E}^p}{\partial y_j^p} g'(v_j^p).$$

$$\frac{\partial \mathcal{E}^p}{\partial y_j^p} = \sum_{k \in C} e_k^p \frac{\partial e_k^p}{\partial y_j^p} = \sum_{k \in C} e_k^p \frac{\partial e_k^p}{\partial v_j^p}\frac{\partial v_k^p}{\partial y_j^p} = -\sum_{k \in C} e_k^p g'(v_j^p) w_{kj} = -\sum_{k \in C} \delta_k^p w_{kj}.$$

* The above rule can be used in general to calculate Δw_{ji} for any weight in the network.

BACKPROPAGATION ALGORITHM (6)

Issues that matter:

* Does the algorithm get stuck in local minima? Answer: yes.
* What is an appropriate stopping criterion? Convergence cannot be proved in general.
* Can the learning speed be improved?

FURTHER READING

* The presentation of the McCullogh-Pitts and Rosenblatt perceptrons was mainly based on:

[1] Rojas, R., "Neural Networks, A Systematic Introduction", Springer, Berlin, (1996).

* The presentation of the backpropagation algorithm was mainly based on:

[2] Haykin, S., Neural Networks, A Comprehensive Foundation, Prentice Hall International, Upper Saddle River, New Jersey, Second Edition, (1999).