Perceptron, neural network, and the back-propagation algorithm

Pavlo Mozharovskyi[1]

[1]LTCI, Télécom Paris, Institut Polytechnique de Paris

Machine learning

Paris, March 12, 2022

Today

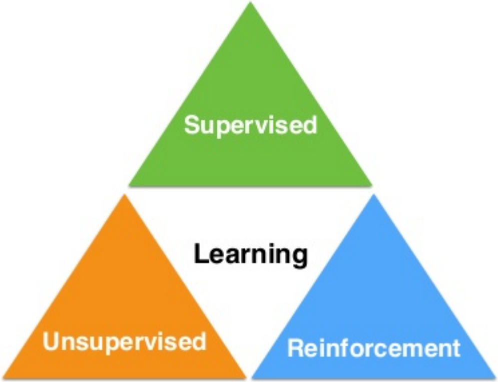
Literature

Supplementary **learning materials** include but are not limited to:

- Haykin, S. (2009).
  Neural Networks and Learning Machines (Third Edition).
  Pearson.
  - Introduction sections 3, 4, 6.
  - Sections 1.1–1.3.
  - Sections 3.3 (steepest descent), 3.5.
  - Sections 4.1–4.4.
- Vapnik, V. N. (1998).
  Statistical Learning Theory.
  John Wiley & Sons.
  - Section 9.1.
  - Section 9.6.
- Goodfellow, J., Bengio, Y., and Courville, A. (2016).
  Deep Learning.
  MIT Press.
- Bertsekas, D. P. (2016).
  Nonlinear programming (Third Edition).
  Athena Scientific.

Types of machine leaning

- Labeled data
- Direct feedback
- Predict outcome/future



- No labels
- No feedback
- “Find hidden structure”

- Decision process
- Reward system
- Learn series of actions

Binary supervised classification (reminder)

Notation:

- **Given:** for the random pair (X, Y) in $\mathbb{R}^d \times \{0, 1\}$ consisting of a random observation X and its random binary label Y (class), a sample of n i.i.d.: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)$.
- **Goal:** predict the label of the new (unseen before) observation $\boldsymbol{x}$.
- **Method:** construct a classification rule:

$$g \,:\, \mathbb{R}^d \to \{0, 1\}\,,\ \boldsymbol{x} \mapsto g(\boldsymbol{x})\,,$$

  so $g(\boldsymbol{x})$ is the prediction of the label for observation $\boldsymbol{x}$.
- **Criterion:** of the performance of g is the **error probability**:

$$R(g) = \mathbb{P}[g(X) \neq Y] = \mathbb{E}[\mathbb{1}\big(g(X) \neq Y\big)]\,.$$

- **The best solution:** is to know the distribution of (X,Y):

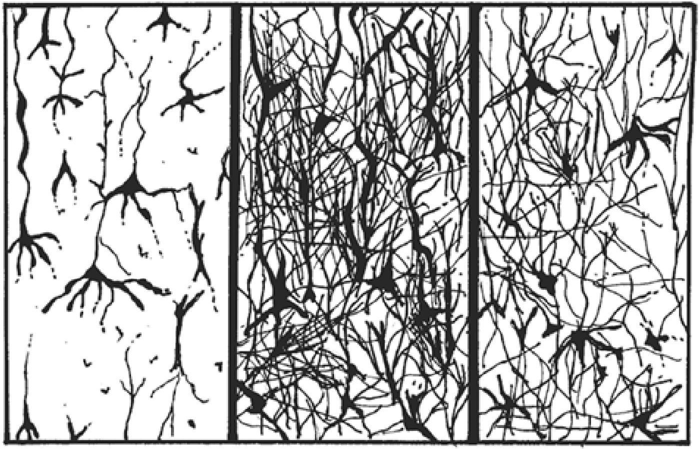
$$g(\boldsymbol{x}) = \mathbb{1}\big(\mathbb{E}[Y|X = \boldsymbol{x}] > 0.5\big)\,.$$

Contents

Contents

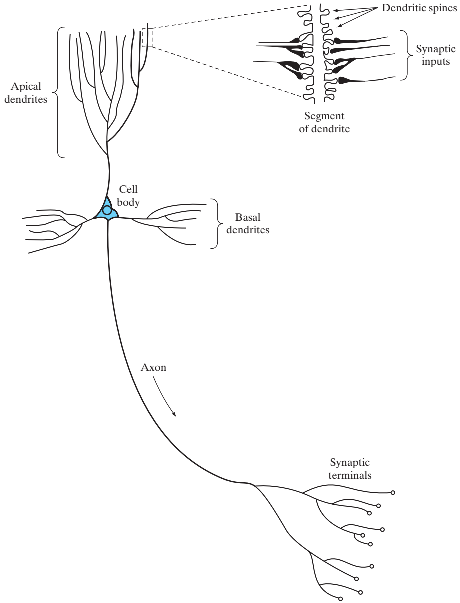
Neurons in the human body

Density of neurons in the human brain at different ages



at a child's birth   at 7 years of age   at 15 years of age

Pyramidal neuron



Contents

The three waves of machine learning

Wave 1 **1952–1974: The birth and the golden years**

- Biological analogy – McCulloch and Pitts model of neuron
- First try – Rosenblatt's perceptron
- Statistical foundations – Vapnik-Chervonenkis theory

Wave 2 **1980–1987: The boom**

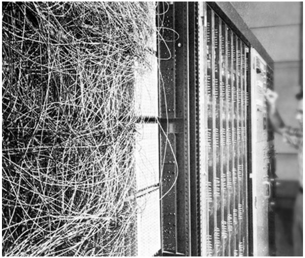
- Visual cortex model – neocognitron by Fukushima
- Expert systems
- Knowledge engineering
- Recurrent architectures – Hopfield net
- Learning algorithm – back-propagation by Hinton and Rumelhart

Wave 3 **1993–.......: Contemporary architectures**

- Convolutional networks (CNNs) – LeNet by LeCun
- CNNs with ReLU, drop-out, GPUs – AlexNet by Krizhevsky et al.
- Generative adversarial networks (GANs) – Goodfellow
- Big data deep learning (DL)
- Artificial general intelligence – full AI
- ...

Rosenblatt's perceptron

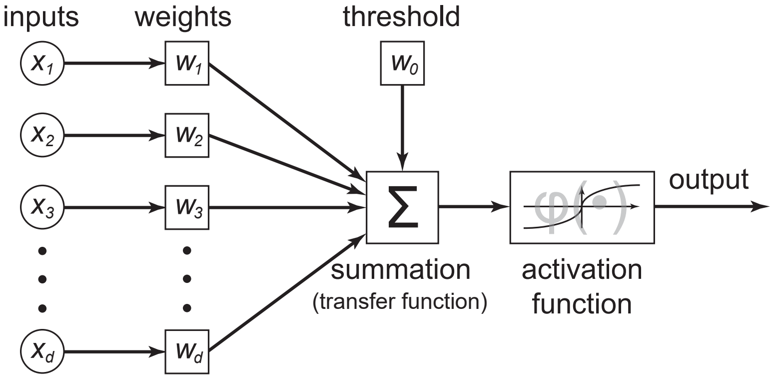
The **perceptron algorithm** was invented in 1957 at the Cornell Aeronautical Laboratory by **Frank Rosenblatt**.



(Photo downloaded from http://www.usbdata.co/rosenblatt-perceptron.html)

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm. The machine was connected to a **Camera** that used **20x20** cadmium sulfide **photocells** to produce a 400-pixel image. The main visible feature is a **Patchboard** that **allowed experimentation with** different combinations of **input features**. To the right of that are arrays of **Potentiometers** that **implemented the adaptive weights**.

Rosenblatt's perceptron



Let $\boldsymbol{w} = (w_1, w_2, ..., w_d)^T$ be the weight vector, then a new observation $\boldsymbol{x} = (x_1, x_2, ..., x_d)^T$ is classified as

$$g(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \phi\left(\sum_{k=1}^{d} w_k x_k + w_0\right) > 0\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

Rosenblatt's perceptron (training algorithm)

Initialize w_0 and $\boldsymbol{w}$ randomly or set $w_0 = 0$ and $\boldsymbol{w} = \boldsymbol{0}$.
Choose constant $\gamma \in (0, 1]$ controlling the learning speed.

Feed training pairs $(\boldsymbol{x}, y)$, and for each of them update current threshold and weights $w_0^{(i)}$ and $\boldsymbol{w}^{(i)}$ to $w_0^{(i+1)}$ and $\boldsymbol{w}^{(i+1)}$ as follows:

1. Classify current observation $\boldsymbol{x}$:

$$o^{(i)} = \begin{cases} 1 & \text{if } \sum_{k=1}^{d} w_k x_k + w_0 > 0\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

2. Calculate correction:

$$\delta^{(i)} = \begin{cases} 0 & \text{if } o^{(i)} = y\,, \\ 1 & \text{if } o^{(i)} = 0 \text{ but } y = 1\,, \\ -1 & \textit{if } o^{(i)} = 1 \text{ but } y = 0\,. \end{cases}$$

3. Update threshold and weights:

$$\begin{aligned} \boldsymbol{w}^{(i+1)} &= \boldsymbol{w}^{(i)} + \gamma \delta^{(i)} \boldsymbol{x}\,, \\ w_0^{(i+1)} &= w_0^{(i)} + \gamma \delta^{(i)}\,. \end{aligned}$$

Iris data

Fisher's iris data: is this the same flower?

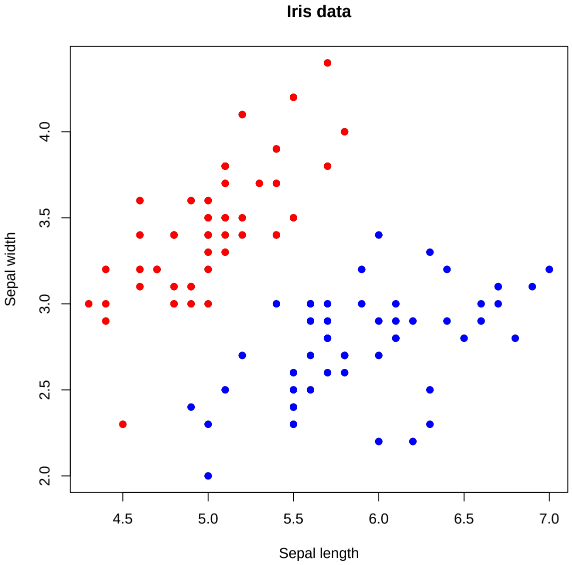
Iris **setosa**

Iris **versicolor**

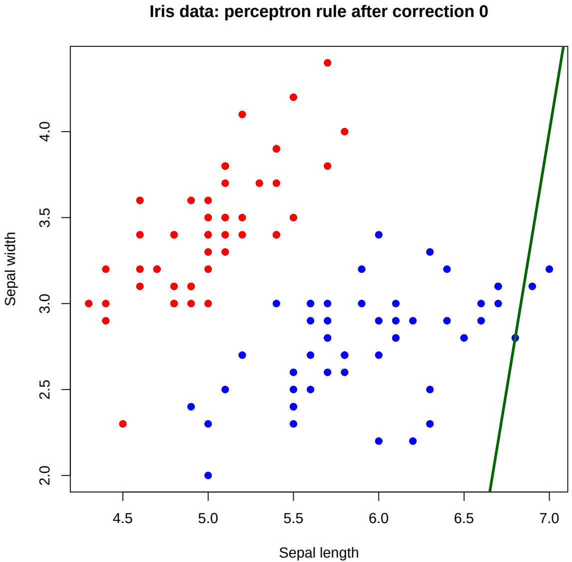
Iris data

Iris **setosa**		Iris **versicolor**	
Sepal length (cm)	Sepal width (cm)	Sepal length (cm)	Sepal width (cm)
5.1	3.5	7	3.2
4.9	3	6.4	3.2
4.7	3.2	6.9	3.1
4.6	3.1	5.5	2.3
5	3.6	6.5	2.8
5.4	3.9	5.7	2.8
4.6	3.4	6.3	3.3
5	3.4	4.9	2.4
4.4	2.9	6.6	2.9
...	...	...	...
...	...	...	...
...	...	...	...
4.6	3.2	6.2	2.9
5.3	3.7	5.1	2.5
5	3.3	5.7	2.8

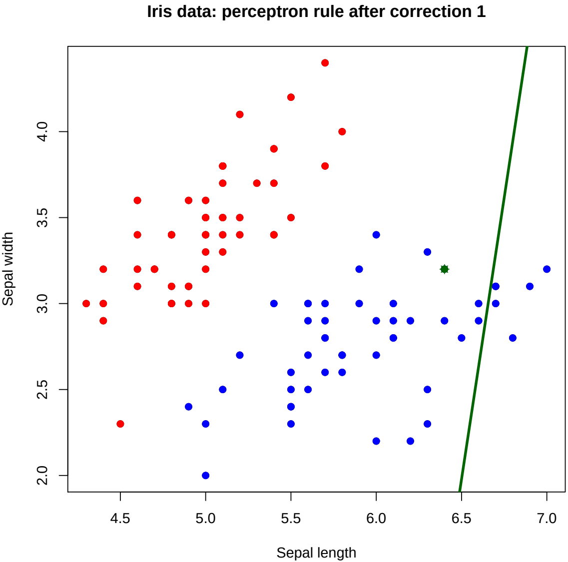
Rosenblatt's perceptron (iris data)



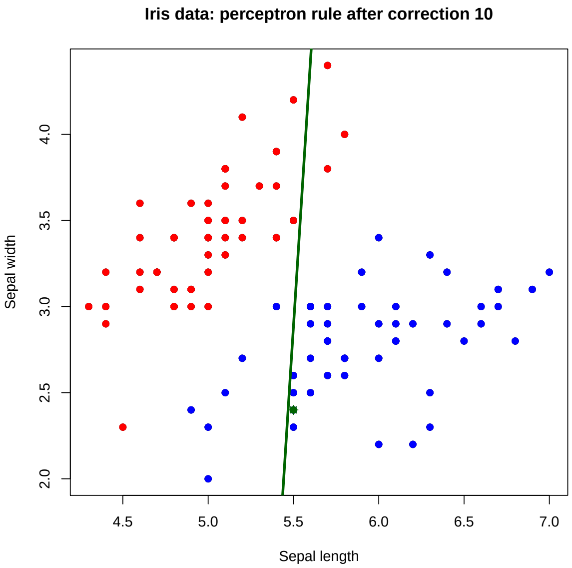
Rosenblatt's perceptron (iris data)



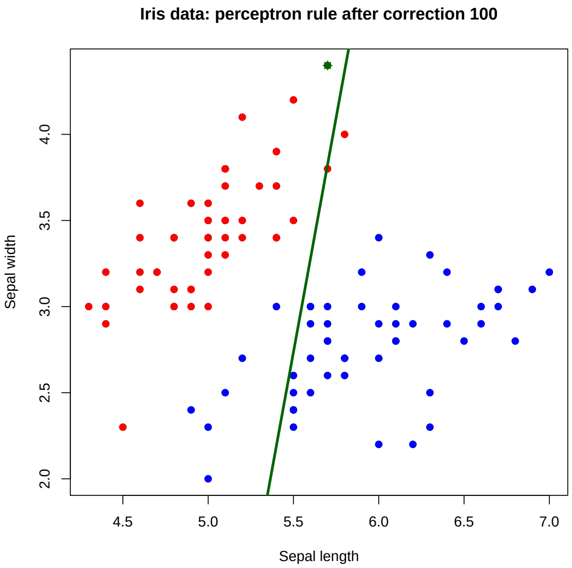
Rosenblatt's perceptron (iris data)



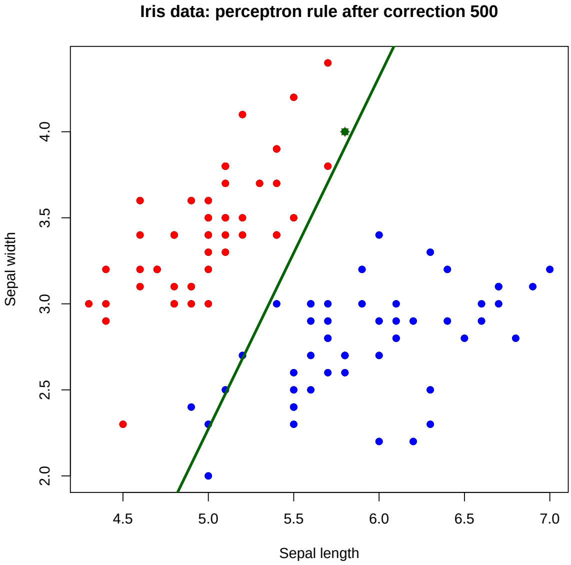
Rosenblatt's perceptron (iris data)



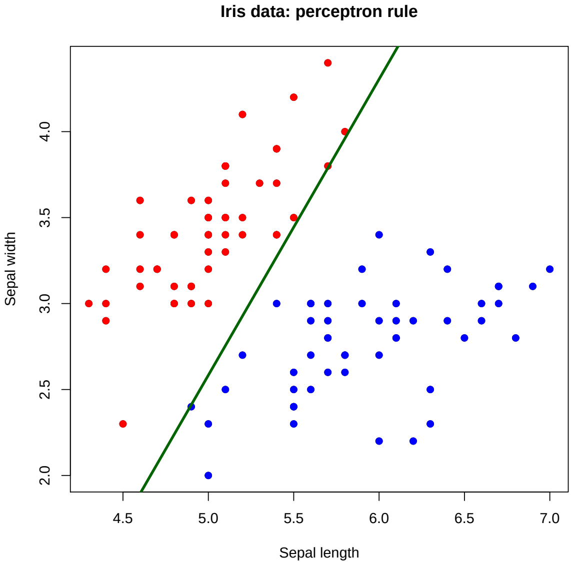
Rosenblatt's perceptron (iris data)



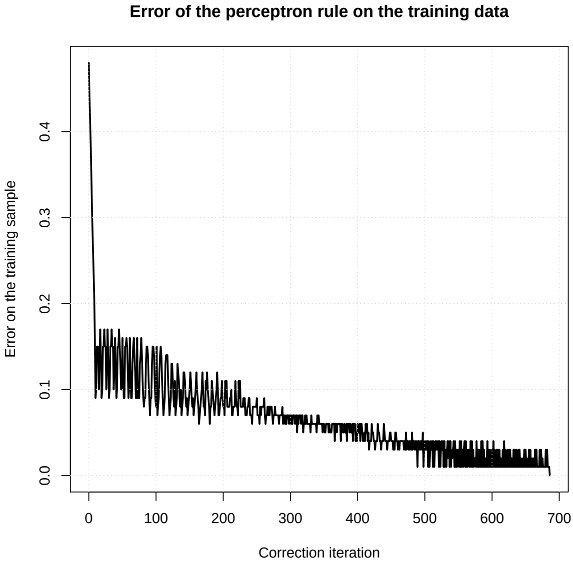
Rosenblatt's perceptron (iris data)



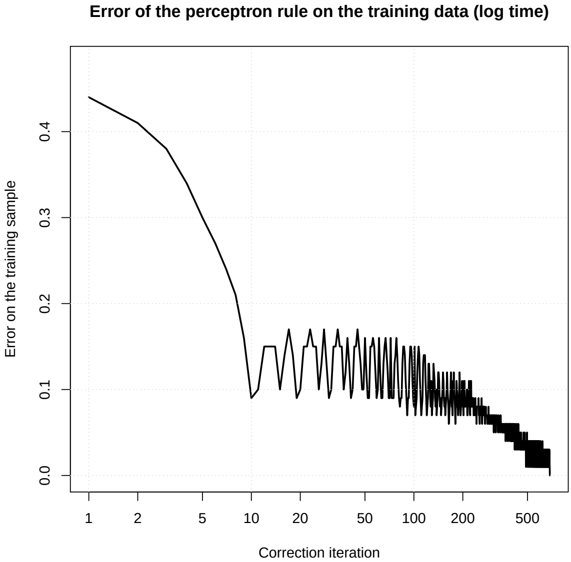
Rosenblatt's perceptron (iris data)



Rosenblatt's perceptron (iris data)



Rosenblatt's perceptron (iris data)



Contents

Novikoff's convergence theorem

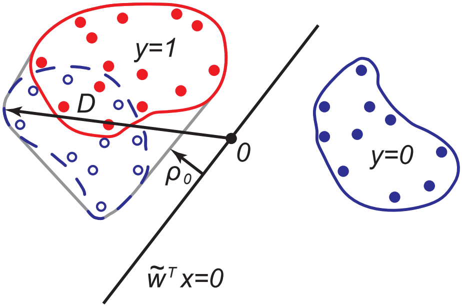
- Let $w_0 = 0$ and set $\gamma = 1$.
- Let $(\mathcal{X}, \mathcal{Y}) = (\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_i, y_i), \ldots$ be an infinite training sequence.
- In addition, let (construct) $\tilde{\mathcal{X}} = \{\boldsymbol{x} \mid (\boldsymbol{x}, y) \in (\mathcal{X}, \mathcal{Y}),\ y = 1\} \cup \{-\boldsymbol{x} \mid (\boldsymbol{x}, y) \in (\mathcal{X}, \mathcal{Y}),\ y = 0\}$.
- Let $\tilde{\boldsymbol{w}}$ exist such that for some $\rho_0 > 0$ it holds

  $$\min_{\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}} \frac{\tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}}}{\|\tilde{\boldsymbol{w}}\|} \geq \rho_0 \,.$$

  i.e. the classes are **linearly separable via the origin** with margin ρ_0.
- Let $0 < D < \infty$ exist such that it holds

  $$\max_{\boldsymbol{x} \in \mathcal{X}} \|\boldsymbol{x}\| < D \,.$$

Novikoff's convergence theorem



Theorem (Novikoff, 1962)

The perceptron constructs a hyperplane that correctly separates all pairs $(\boldsymbol{x}, y) \in (\mathcal{X}, \mathcal{Y})$ *with the number of corrections at most*

$$\left\lfloor \frac{D^2}{\rho_0^2} \right\rfloor .$$

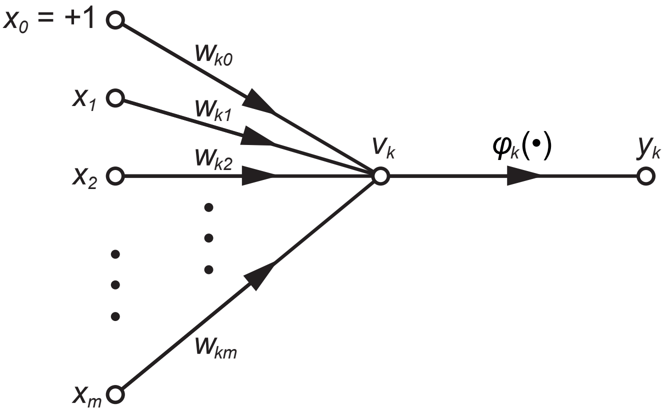
Contents

A signal-flow graph

A **signal-flaw graph** is a network of directed **links (branches)** that are interconnected at certain points called **nodes**.

1. A signal flows along a link only in the direction defined by the arrow on the link.
   There are two types of links:
   - **Synaptic links**: behavior is defined by a **linear** input-output relation. Specifically, the node signal x_j is multiplied with the synaptic weight w_{kj} to produce the node signal y_k.
   - **Activation links**: behavior is defined by a **nonlinear** input-output relation. The change of the signal is performed due to the activation function $\phi(\cdot)$.
2. A node signal equals the algebraic sum of all signals entering the pertinent node via the incoming links: **synaptic convergence** or **fan-in**.
3. The signal at node is transmitted to each outgoing link originating from that node, with the transmission entirely independent of the transfer functions of the outgoing links: **synaptic divergence** or **fan-out**.

A signal-flow graph of a neuron



Definition of a neural network

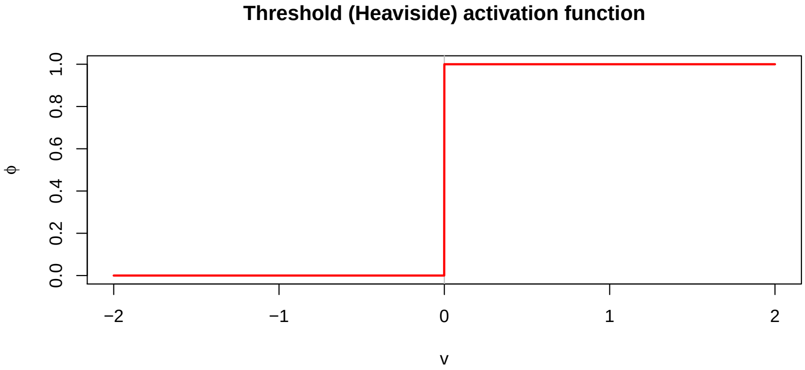
A **neural network is a directed graph** consisting of nodes with interconnecting synaptic and activation links and is characterized by four properties:

1. Each neuron is represented by a set of linear synaptic links, an externally applied bias, and a possibly nonlinear activation link. The bias is represented by a synaptic link connected to an input fixed at $+1$.
2. The **synaptic links** of a neuron weight their respective input signals.
3. The weighted sum of the input signals defines the **induced local field** of the neuron in question.
4. The **activation link** squashes the induced local field of the neuron to produce **output**.

Activation function $\phi(v)$

- **Threshold (Heaviside) function:**

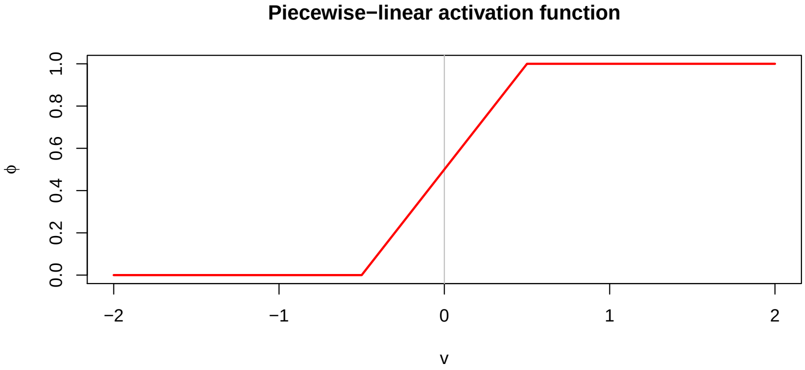
$$\phi(v) = \begin{cases} 1 & \text{if } v \geq 0\,, \\ 0 & \text{if } v < 0\,. \end{cases}$$



Activation function $\phi(v)$

- **Piecewise-linear function:**

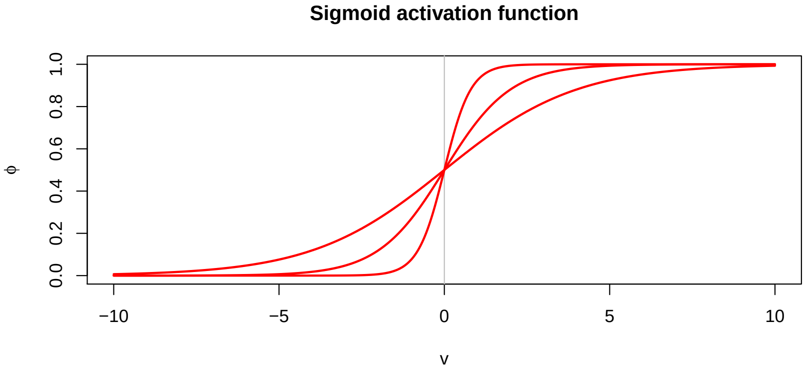
$$\phi(v) = \begin{cases} 1 & \text{if } v \geq \frac{1}{2}\,, \\ v + \frac{1}{2} & \text{if } -\frac{1}{2} < v < \frac{1}{2}\,, \\ 0 & \text{if } v \leq -\frac{1}{2}\,. \end{cases}$$



Activation function $\phi(v)$

- **Sigmoid function:**

$$\phi(v) = \frac{1}{1 + exp(-av)} .$$



Contents

Contents

Minimization of the empirical risk: a reminder

- For:
  - a random pair (X, Y),
  - a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

  one seeks a classifier close to:

$$g^* = \arg\min_g \mathbb{E}[\ell(g(X), Y)] \, .$$

- **Strategy:** Given a *training sample* $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)$ of (X, Y), one minimizes the empirical version of $\mathbb{E}[\ell(g(X), Y)]$:

$$\frac{1}{n} \sum_{i=1}^{n} \ell\big(g(\boldsymbol{x}_i), y_i\big) \, .$$

- **Method:** Numerical optimization, *e.g.*, **gradient descent**.

- **Stochastic gradient descent:** Use a single (randomly drawn) observation to iteratively approximate g^*.

Contents

Method of gradient descent

- Consider a cost function $\mathcal{E}$ that is **continuously differentiable** function of some unknown weight (parameter) vector $\boldsymbol{w}$.
- In the **method of gradient descent**, the successive adjustments are applied to $\boldsymbol{w}$ in the direction of gradient descent, *i.e.* in the direction opposite to the gradient vector $\nabla\mathcal{E}$:

$$\boldsymbol{g} = \nabla\mathcal{E}(\boldsymbol{w}) = \left(\frac{\partial\mathcal{E}}{\partial w_1}\boldsymbol{e}_{w_1}, \frac{\partial\mathcal{E}}{\partial w_2}\boldsymbol{e}_{w_2}, ..., \frac{\partial\mathcal{E}}{\partial w_m}\boldsymbol{e}_{w_m}\right)^T .$$

- The step of the algorithm is then defined as

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \gamma\boldsymbol{g}(n)\,,$$

  where γ is the **learning rate**.
- When going from iteration n to iteration $n+1$, the **correction** is applied to the weights:

$$\begin{aligned}\Delta\boldsymbol{w}(n) &= \boldsymbol{w}(n+1) - \boldsymbol{w}(n)\\ &= -\gamma\boldsymbol{g}(n)\,.\end{aligned}$$

Method of gradient descent

- Let us show that the constructed algorithm fulfills the idea of the **iterative descent**, *i.e.* that it satisfies

$$\mathcal{E}\big(\boldsymbol{w}(n+1)\big) < \mathcal{E}\big(\boldsymbol{w}(n)\big)\,.$$

- Using the first-order Taylor series expansion around $\boldsymbol{w}(n)$ to approximate $\mathcal{E}\big(\boldsymbol{w}(n+1)\big)$ as

$$\mathcal{E}\big(\boldsymbol{w}(n+1)\big) \approx \mathcal{E}\big(\boldsymbol{w}(n)\big) + \boldsymbol{g}^T(n)\Delta\boldsymbol{w}(n)$$

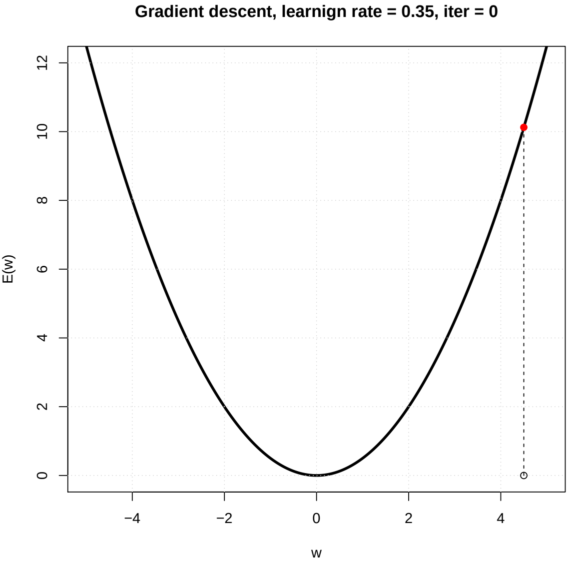
is justified for γ small enough.

- Substituting $\Delta\boldsymbol{w}(n)$ gives:

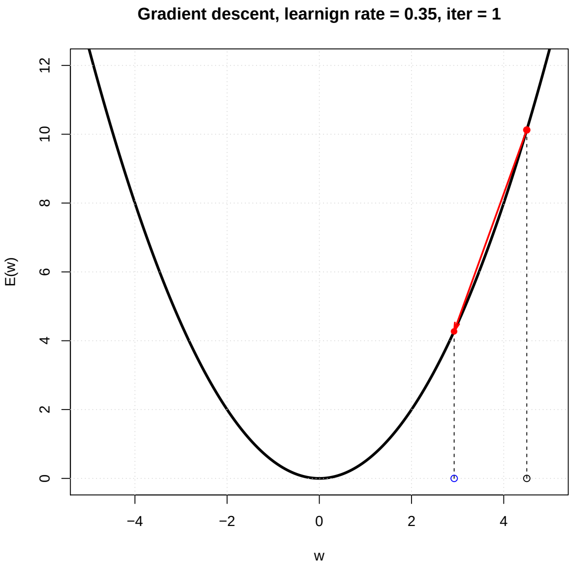
$$\begin{aligned}\mathcal{E}\big(\boldsymbol{w}(n+1)\big) &\approx \mathcal{E}\big(\boldsymbol{w}(n)\big) - \gamma\boldsymbol{g}^T(n)\boldsymbol{g}(n)\\ &= \mathcal{E}\big(\boldsymbol{w}(n)\big) - \gamma\|\boldsymbol{g}(n)\|^2\,,\end{aligned}$$

and thus for a positive learning rate the cost function decreases on each iteration.

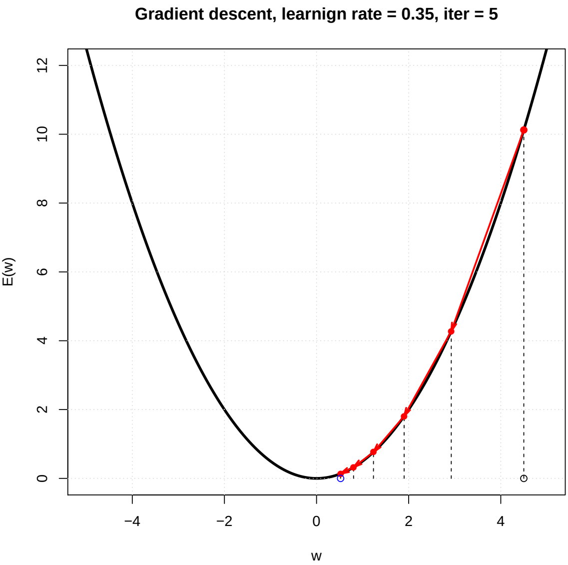
Method of gradient descent: example (slow learning rate)



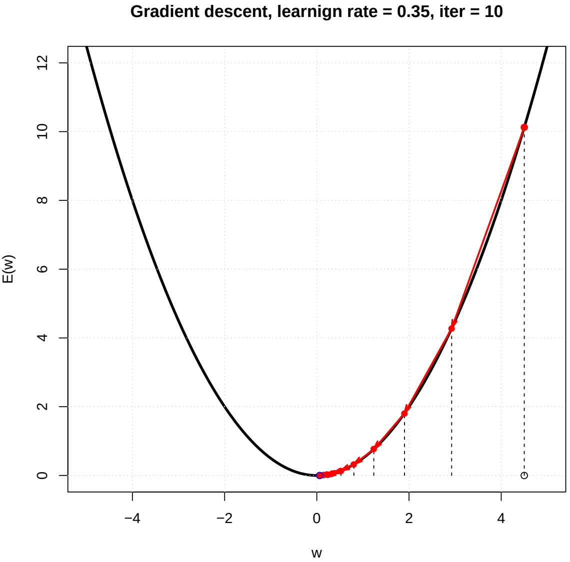
Method of gradient descent: example (slow learning rate)



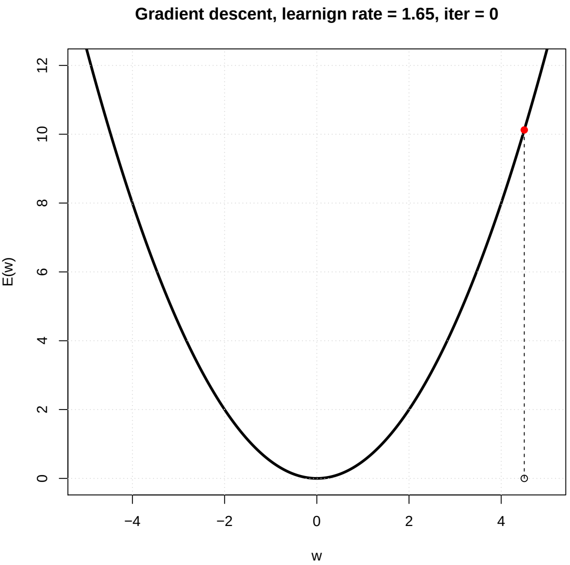
Method of gradient descent: example (slow learning rate)



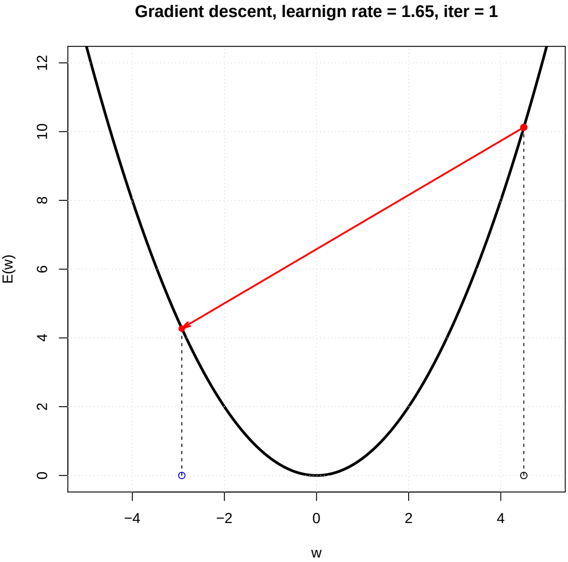
Method of gradient descent: example (slow learning rate)



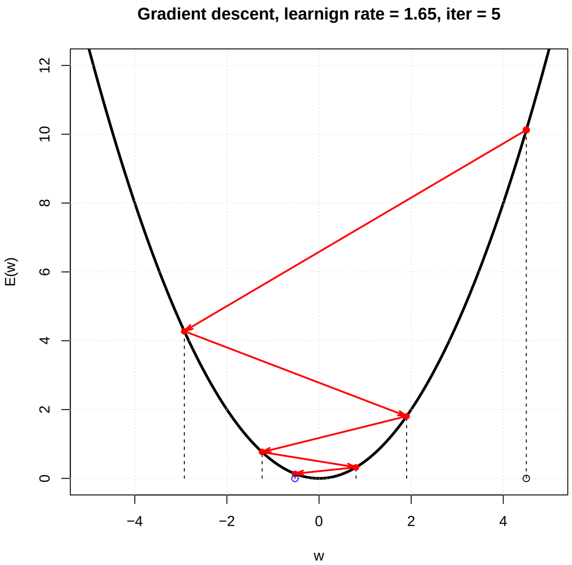
Method of gradient descent: example (fast learning rate)



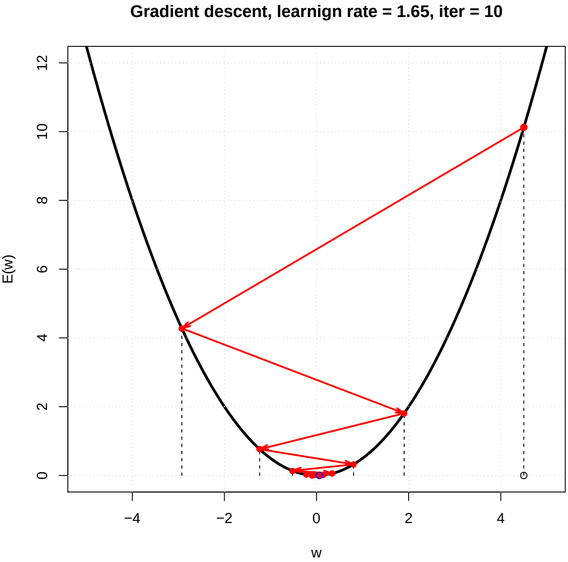
Method of gradient descent: example (fast learning rate)



Method of gradient descent: example (fast learning rate)



Method of gradient descent: example (fast learning rate)

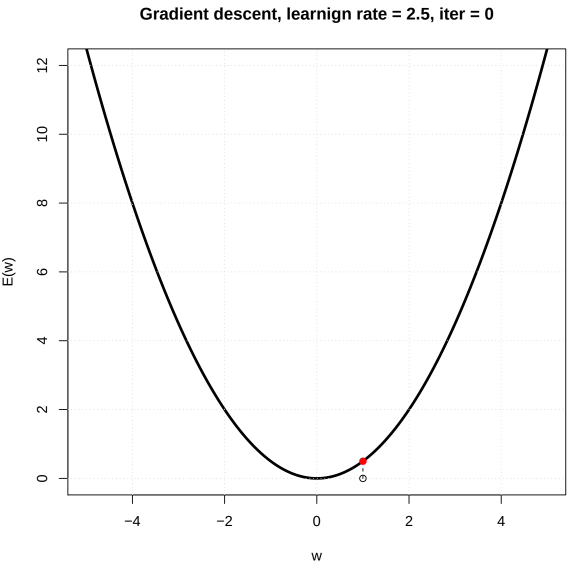


Method of gradient descent, choice of learning rate

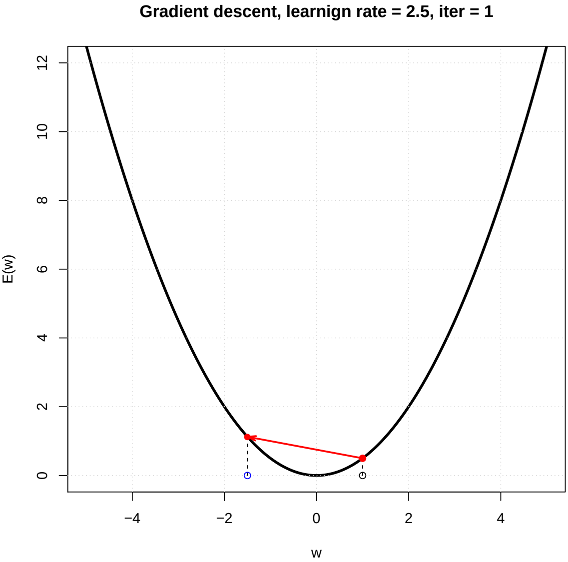
The method **converges** to the optimal solution **slowly**.
Moreover, the **learning rate** γ has a profound influence on its convergence behavior:

- When γ is small, the transient response of the algorithm is **overdamped**, and the trajectory of $\boldsymbol{w}(n)$ follows a smooth path in the parameter space.
- When γ is large, the transient response is **underdamped**, and the trajectory of $\boldsymbol{w}(n)$ follows a zigzagging (oscillatory) path.
- When γ exceeds a certain critical value, the algorithm becomes unstable and may diverge.

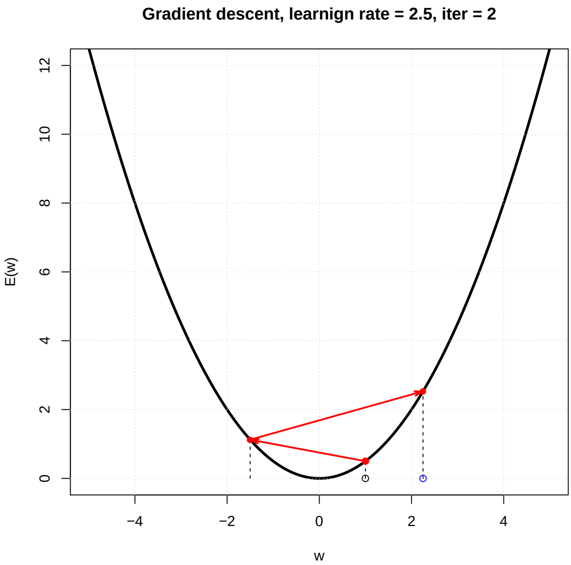
Method of gradient descent: example (divergent case)



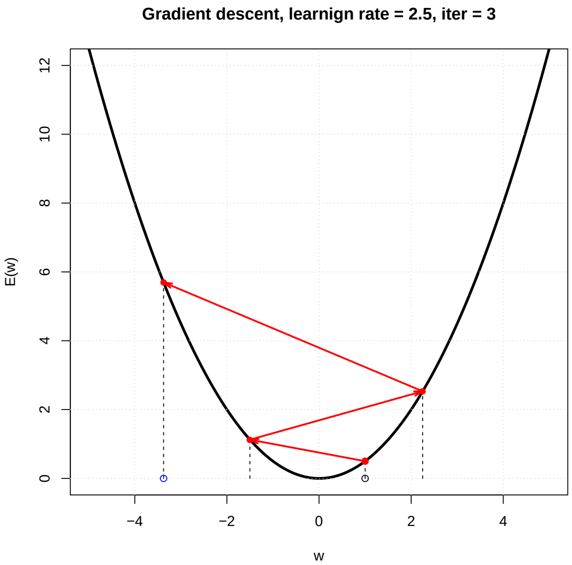
Method of gradient descent: example (divergent case)



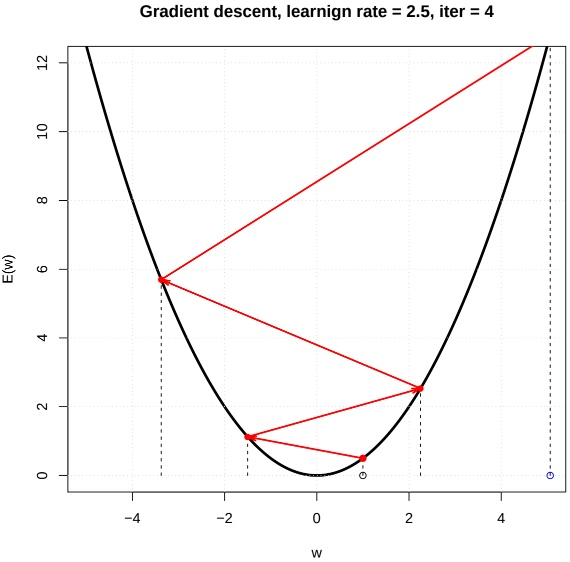
Method of gradient descent: example (divergent case)



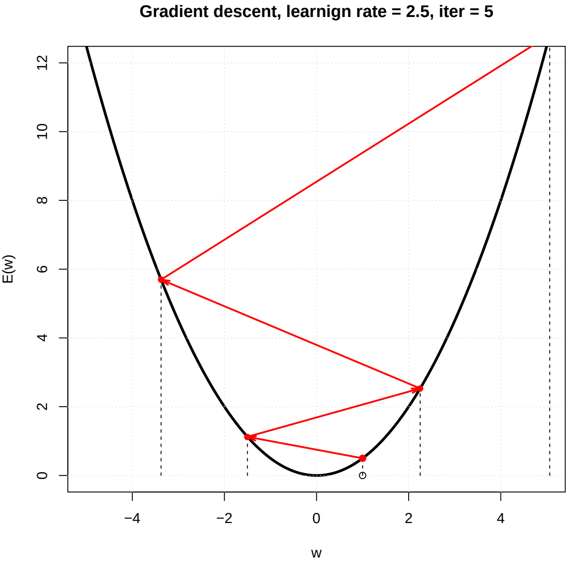
Method of gradient descent: example (divergent case)



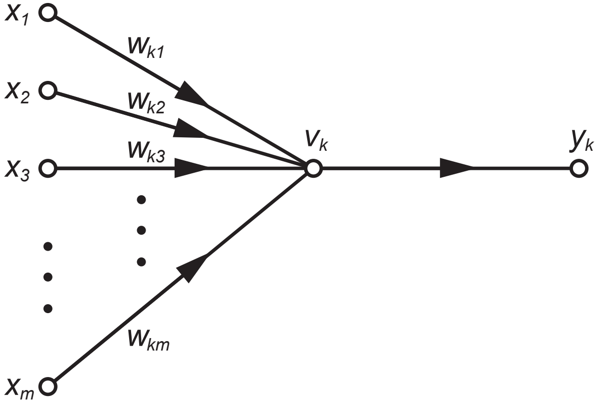
Method of gradient descent: example (divergent case)



Method of gradient descent: example (divergent case)



A signal-flow graph of a simplified neuron



Method of gradient descent: iris data

- A simplified neuron has the following prediction function:

$$p_{\boldsymbol{w}}(\boldsymbol{x}) = \sum_{q=1}^{m} w_q x_q \,.$$

- For a data sample $(\boldsymbol{x}_1, y_1)$, $(\boldsymbol{x}_2, y_2)$ ..., $(\boldsymbol{x}_n, y_n)$, let us measure its empirical error by a convex function, *e.g.* using the quadratic loss:

$$\mathcal{E}(\boldsymbol{w}) = \frac{1}{2} \times \frac{1}{n} \sum_{j=1}^{n} e_j(\boldsymbol{w})^2 = \frac{1}{2} \times \frac{1}{n} \sum_{j=1}^{n} \big(y_j - p_{\boldsymbol{w}}(\boldsymbol{x}_j)\big)^2 \,.$$

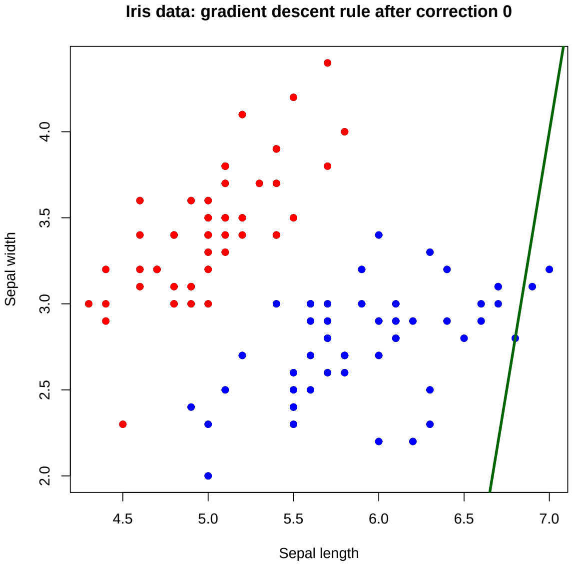
- The gradient equals:

$$\boldsymbol{g} = \nabla \mathcal{E}(\boldsymbol{w}) = -\frac{1}{n} \sum_{j=1}^{n} \big(y_j - p_{\boldsymbol{w}}(\boldsymbol{x}_j)\big) \boldsymbol{x}_j \,.$$

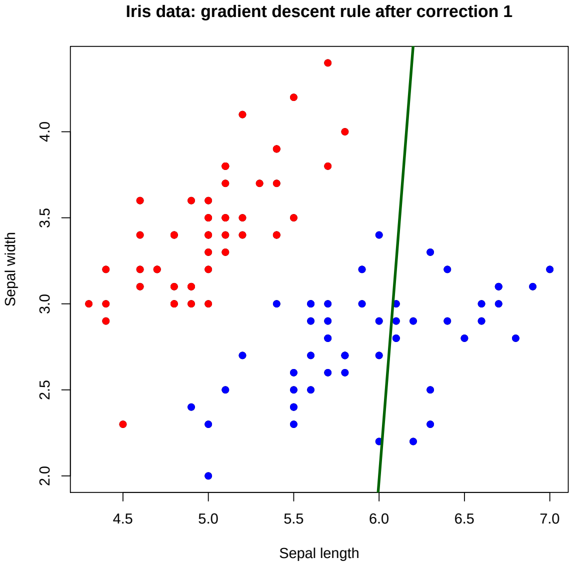
- The step of the algorithm is then:

$$\boldsymbol{w}(i+1) = \boldsymbol{w}(i) - \gamma \boldsymbol{g}(i) = \boldsymbol{w}(i) + \frac{\gamma}{n} \sum_{j=1}^{n} \big(y_j - \boldsymbol{w}(i)^T \boldsymbol{x}_j\big) \boldsymbol{x}_j \,.$$

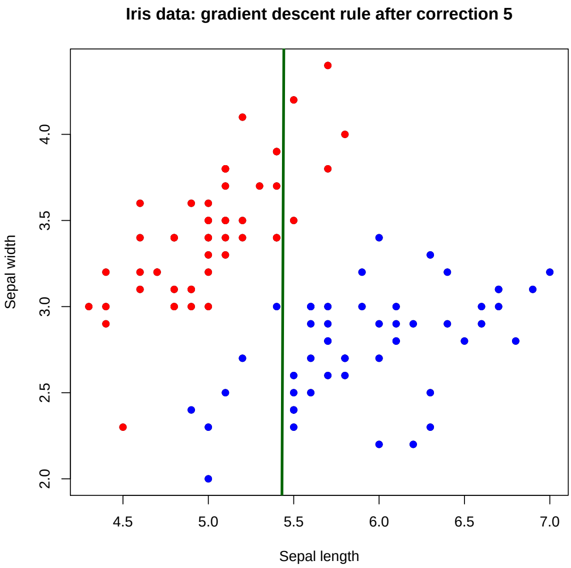
Method of gradient descent: iris data



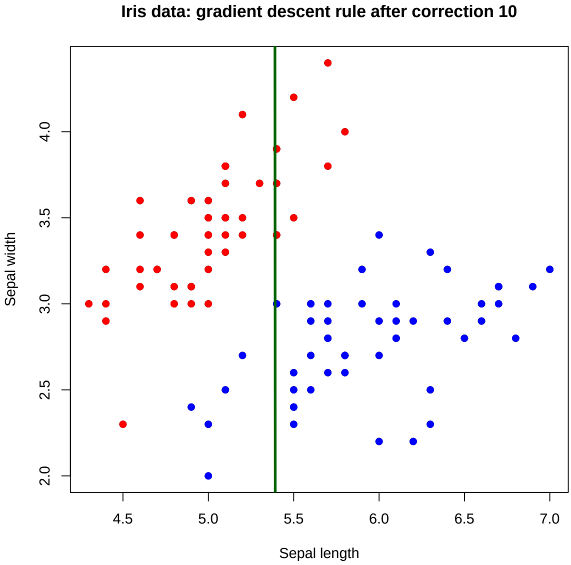
Method of gradient descent: iris data



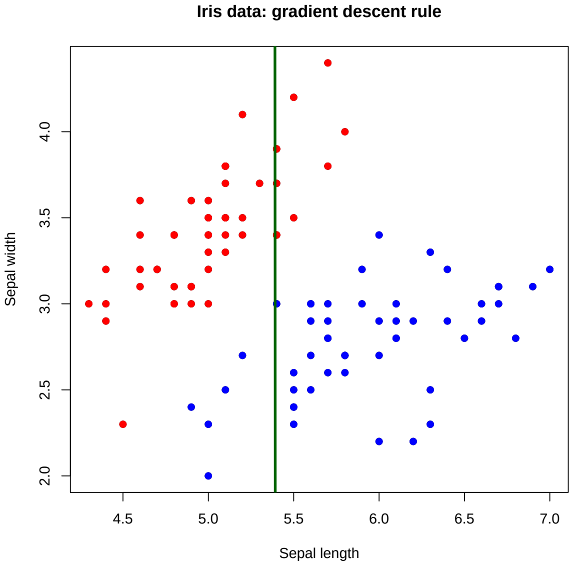
Method of gradient descent: iris data



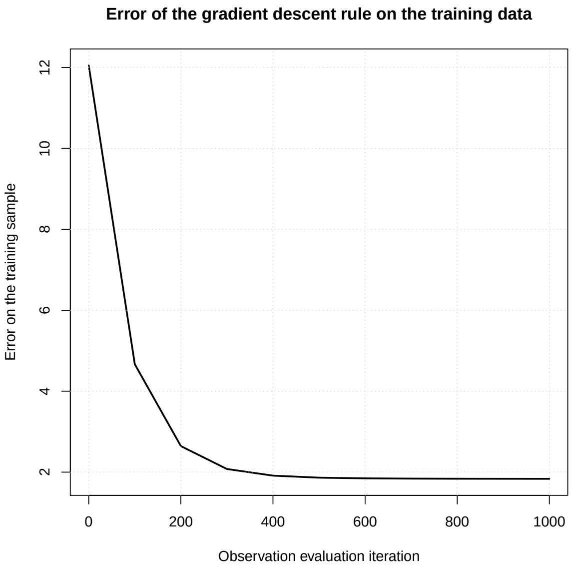
Method of gradient descent: iris data



Method of gradient descent: iris data



Method of gradient descent: iris data



Contents

The least-mean-square algorithm

- The **least-mean-square (LMS) algorithm** attempts to minimize the **instantaneous value** of the cost function

$$\mathcal{E}(\hat{\boldsymbol{w}}) = \frac{1}{2}e^2(n)\,,$$

where $e(n)$ is the error measured at time n.
- Differentiating $\mathcal{E}(\hat{\boldsymbol{w}})$ w.r.t. $\hat{\boldsymbol{w}}$ gives:

$$\frac{\partial \mathcal{E}(\hat{\boldsymbol{w}})}{\partial \hat{\boldsymbol{w}}} = e(n)\frac{\partial e(n)}{\partial \hat{\boldsymbol{w}}}\,.$$

- When operating on the linear neuron, the error can be expressed as:

$$e(n) = d(n) - \boldsymbol{x}^T(n)\hat{\boldsymbol{w}}(n)\,.$$

- Thus

$$\frac{\partial e(n)}{\partial \hat{\boldsymbol{w}}} = -\boldsymbol{x}(n) \quad \text{and} \quad \frac{\partial \mathcal{E}(\hat{\boldsymbol{w}})}{\partial \hat{\boldsymbol{w}}(n)} = -\boldsymbol{x}(n)e(n) = \hat{\boldsymbol{g}}(n)\,.$$

- Now, LMS can be formulated as follows:

$$\hat{\boldsymbol{w}}(n+1) = \hat{\boldsymbol{w}}(n) + \gamma\boldsymbol{x}(n)e(n)\,.$$

The least-mean-square algorithm

Input:

- Input signals $\boldsymbol{x}(n)$ with correct outputs $d(n)$ for n=1,2,...
- Learning rate γ.

Initialization:

- Set $\hat{\boldsymbol{w}}(1) = \mathbf{0}$.

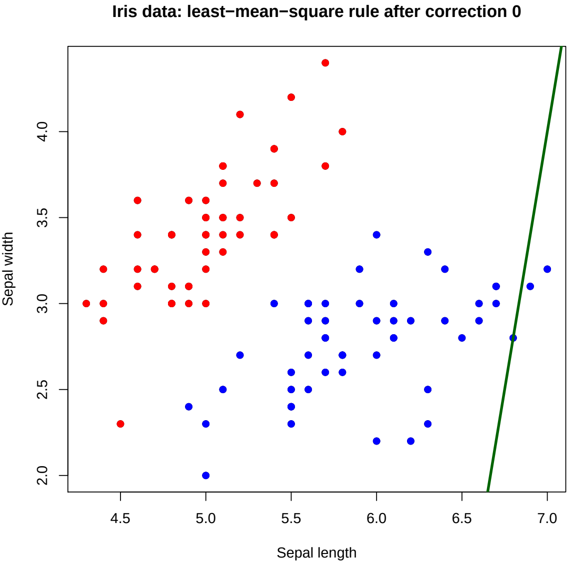
Iterations:

- For $n = 1, 2, \dots$, compute
  - $e(n) = d(n) - \hat{\boldsymbol{w}}^T(n)\boldsymbol{x}(n)$,
  - $\hat{\boldsymbol{w}}(n+1) = \hat{\boldsymbol{w}}(n) + \gamma \boldsymbol{x}(n) e(n)$.

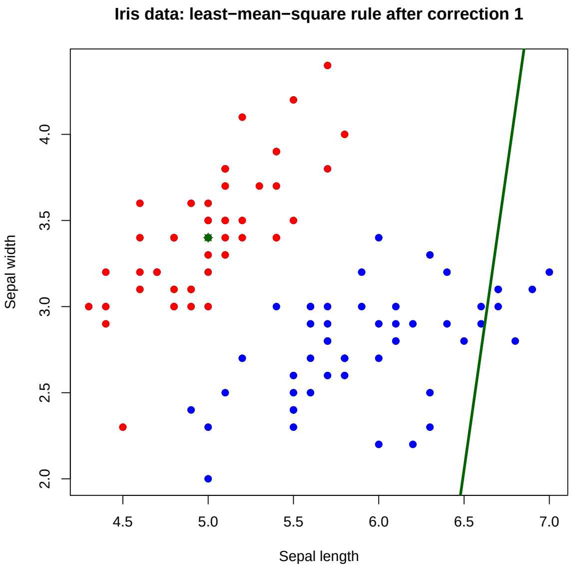
Remarks:

- The inverse of the learning rate acts as a **measure of the memory** of the LMS algorithm: the smaller γ is set, the longer the memory span over which the LMS remembers the past data will be.
- Having $\hat{\boldsymbol{w}}(n)$ in place of $\boldsymbol{w}(n)$ emphasizes that the LMS algorithm produces the **instantaneous estimate** of the weights, which would result from the gradient descent.
- For this last reason, the LMS-updated weights $\hat{\boldsymbol{w}}(n)$ trace a random trajectory in the parameter space: **stochastic gradient descent**.

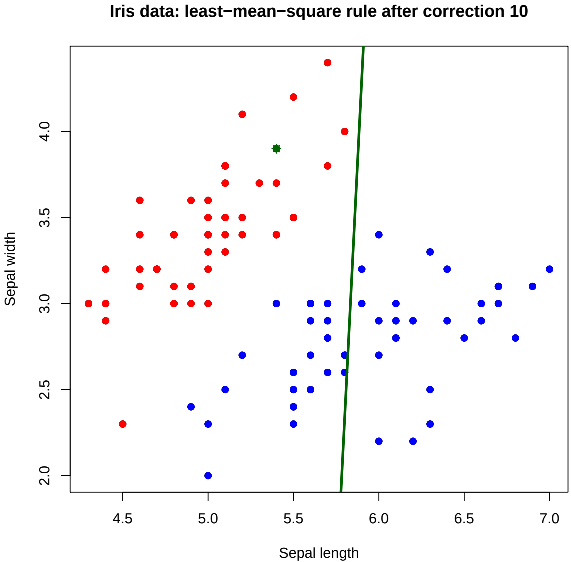
The least-mean-square algorithm: iris data



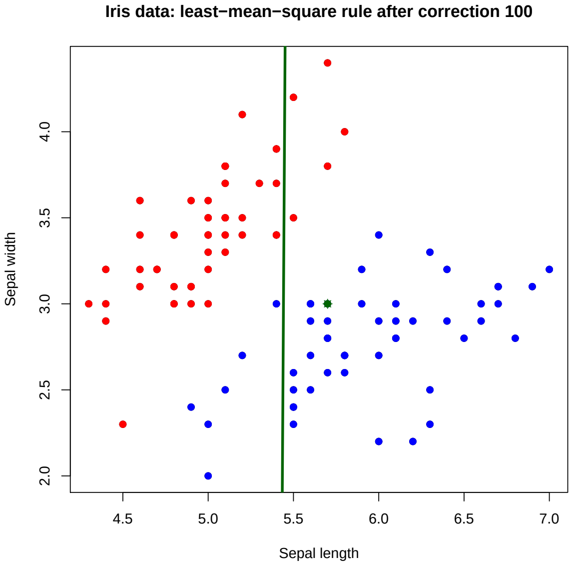
The least-mean-square algorithm: iris data



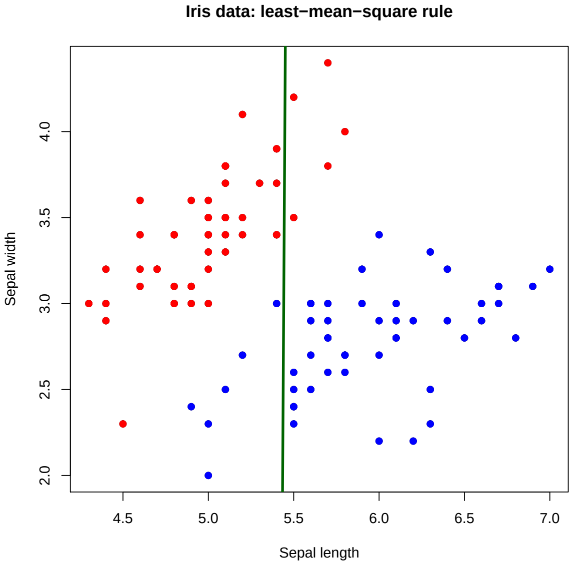
The least-mean-square algorithm: iris data



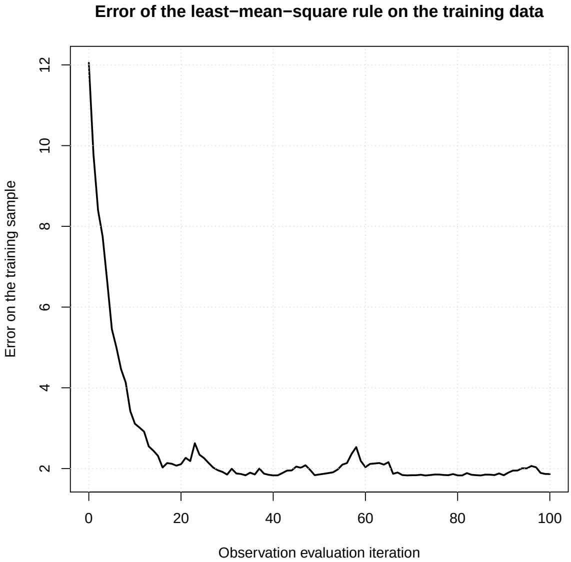
The least-mean-square algorithm: iris data



The least-mean-square algorithm: iris data



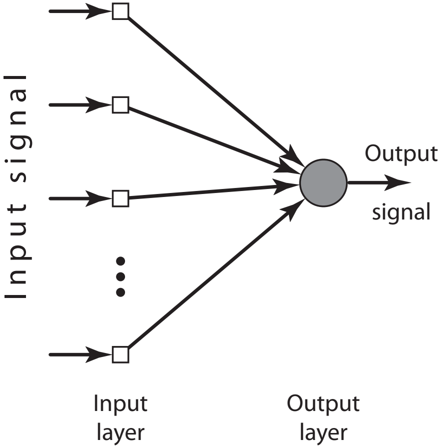
The least-mean-square algorithm: iris data



Contents

Contents

Single layer perceptron



Preliminaries

- **Error signal** produced at the output of neuron j is defined by

$$e_j(n) = d_j(n) - y_j(n)\,,$$

where $d_j(n)$ is the correct output.

- As before, we define the **cost function** penalizing neuron j as

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n)\,.$$

- The **induced local field** $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

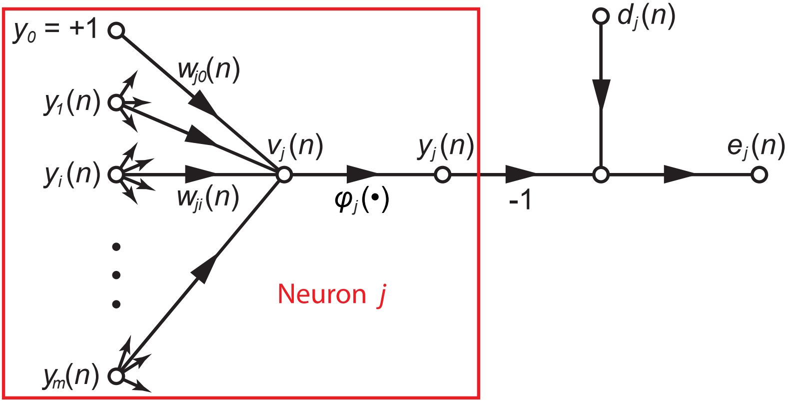
$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)\,,$$

where m is the number of inputs excluding bias applied to neuron j.

- The **output signal** $y_j(n)$ appearing at the output of neuron j at iteration n equals:

$$y_j(n) = \phi\big(v_j(n)\big)\,.$$

A signal-flow graph of a neuron and its output



Derivation for a single (output-layer) neuron

- In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight w_{ji}, which is proportional to the partial derivative $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$. Applying of the **chain rule** of the calculus, this gradient can be expressed as:

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \times \frac{\partial e_j(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial v_j(n)} \times \frac{\partial v_j(n)}{\partial w_{ji}(n)} \,.$$

- Partial derivatives equal:

$$\begin{aligned}
\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} &= e_j(n)\,, \\
\frac{\partial e_j(n)}{\partial y_j(n)} &= -1\,, \\
\frac{\partial y_j(n)}{\partial v_j(n)} &= \phi_j'(v_j(n))\,, \\
\frac{\partial v_j(n)}{\partial w_{ji}(n)} &= y_i(n)\,.
\end{aligned}$$

Derivation for a single (output-layer) neuron

- From these, the partial derivative can be composed as:

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\phi_j'\big(v_j(n)\big)y_i(n)\,.$$

- This partial derivative represents a **sensitivity factor**, which determines the direction of search in the parameter space for synaptic weight w_{ji}.
- The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined as:

$$\Delta w_{ji}(n) = -\gamma \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}\,,$$

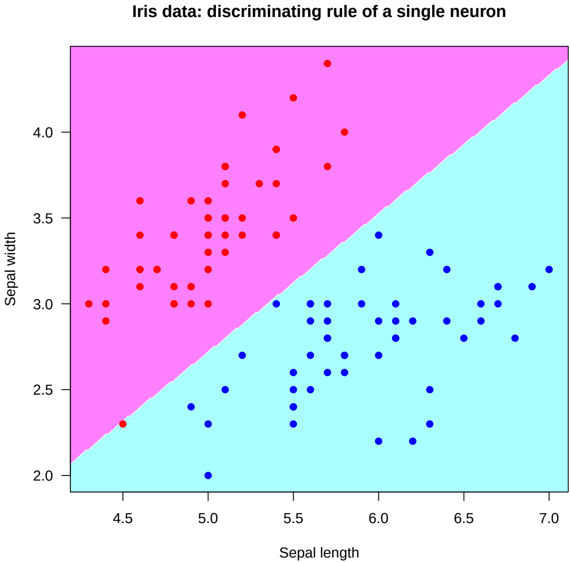
  with γ being the **learning parameter** of the back-propagation algorithm.
- Accordingly, one can generalize

$$\Delta w_{ji}(n) = \gamma \delta_j(n) y_i(n)$$

  where the **local gradient** $\delta_j(n)$ is defined by

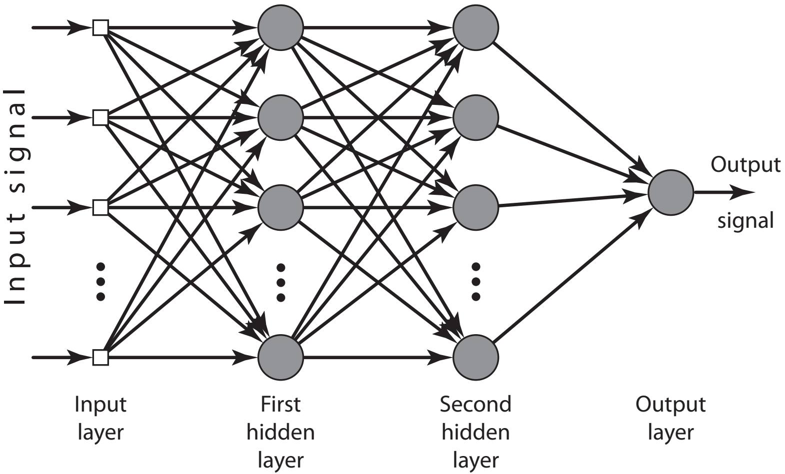
$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n)\phi_j'\big(v_j(n)\big)\,.$$

Single neuron: iris data



Contents

Multilayer neural network (example)



Derivation for a hidden neuron

- **Problem:** When neuron j is located in a hidden layer, there is **no specified desired response** for it.
- For a **hidden neuron** j, one may define the local gradient $\delta_j(n)$ as:

$$\begin{aligned} \delta_j(n) &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \phi_j'(v_j(n)) \,, \quad \text{neuron } j \text{ is hidden} \,. \end{aligned}$$

- To **calculate the partial derivative** $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$, one may proceed as follows (denoting C the set of all output neurons for generality):

$$\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n) \,, \quad \text{neuron } k \text{ is an output node} \,.$$

- Differentiating this w.r.t. the output of neuron j $y_j(n)$, one gets

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k \frac{\partial e_k(n)}{\partial y_j(n)} \,.$$

Derivation for a hidden neuron

- Application of the chain rule to the partial derivative $\frac{\partial e_k(n)}{\partial y_j(n)}$ gives:

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \, .$$

- Using

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \phi_k\big(v_k(n)\big) \, , \quad \text{neuron } k \text{ is an output node} \, , \end{aligned}$$

  one gets

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\phi_k'\big(v_k(n)\big) \, .$$

- Taking into account that the induced local field for neuron k is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \, ,$$

  (with m being the number of inputs) one obtains

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \, .$$

Derivation for a hidden neuron

- The desired partial derivative equals

$$\begin{aligned}\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= -\sum_{k \in C} e_k(n)\phi_k'\big(v_k(n)\big)w_{kj}(n) \\ &= -\sum_{k \in C} \delta_k(n) w_{kj}(n)\,,\end{aligned}$$

  with $\delta_k(n) = e_k(n)\phi_k'\big(v_k(n)\big)$ as before.
- Finally, the **back-propagation formula** for the local gradient $\delta_j(n)$ can be written as:

$$\delta_j(n) = \phi_j'\big(v_j(n)\big)\sum_{k \in C} \delta_k(n) w_{kj}(n)\,, \quad \text{neuron } j \text{ is hidden}\,.$$

Summarizing:

1. If neuron j is an **output neuron**, the local gradient equals

$$\delta_j(n) = \phi_j'(n) e_j(n)\,.$$

2. If neuron j is a **hidden neuron**, the local gradient equals

$$\delta_j(n) = \phi_j'(n)\sum_{k} \delta_k(n) w_{kj}(n)\,, \quad k \text{ indexes next layer neurons}\,.$$

Summary of the back-propagation algorithm

- The back-propagation algorithm applies correction $\Delta w_{ji}(n)$ to the synaptic weight connecting neuron i to neuron j, defined by the delta rule:

$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning} \\ \textit{rate} \\ \gamma \end{pmatrix} \times \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j\,, \\ y_i(n) \end{pmatrix}$$

- To increase the rate of learning while avoiding the danger of instability one may include a momentum term:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \gamma \delta_j(n) y_j(n)\,,$$

  where α is usually a positive number called **momentum constant**. This rule is also called the generalized delta rule (delta rule is its special case with $\alpha = 0$).
- **Criterion of convergence**:
  The back-propagation algorithm is considered to have converged when the absolute rate of change in the average square error per epoch is sufficiently small.

Summary of the back-propagation algorithm

1. **Initialization:** Pick the synaptic weights and thresholds from a uniform distribution with mean 0 and variance chosen due to the shape of the sigmoid function.
2. **Presentation of training examples:** Present the network an epoch of training examples, repeating for each example steps 3. and 4.
3. **Forward computation (classification):** Passing layers $l = 1, ..., L$, compute outputs and errors:

$$y^{(0)}(n) = x_j(n);\ y_j^{(l)}(n) = \phi_j\Big(\sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)\Big);\ e_j(n) = d_j(n) - y_j^{(L)}(n)$$

4. **Backward computation:** Passing layers $l = 1, ..., L$, compute local gradients:

$$\delta_j^{(l)}(n) = \begin{cases} \phi_j'\big(\sum_i w_{ji}^{(L)}(n) y_i^{(L-1)}(n)\big)\, e_j^{(L)}(n) & \text{if output}\,, \\ \phi_j'\big(\sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)\big)\, \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{if hidden}\,. \end{cases}$$

   Adjust synaptic weights:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha w_{ji}^{(l)}(n-1) + \gamma \delta_j^{(l)}(n) y_i^{(l-1)}(n)\,.$$

5. **Iteration:** Feed randomly permuted epochs till convergence.

Contents

An example of $\phi(\cdot)$: logistic function

- In its general form logistic function is defined by:

$$\phi_j\big(v_j(n)\big) = \frac{1}{1+\exp\big(-av_j(n)\big)}\,, \quad a>0\,,$$

  where $v_j(n)$ is the induced local field of neuron j.

- Differentiating w.r.t. $v_j(n)$ one gets:

$$\phi_j'\big(v_j(n)\big) = \frac{a\exp\big(-av_j(n)\big)}{\Big(1+\exp\big(-av_j(n)\big)\Big)^2} = ay_j(n)\big(1-y_j(n)\big)\,.$$

- Denoting network output $o_j(n) = y_j(n)$, the local gradient equals

$$\begin{aligned}\delta_j(n) &= \phi_j'\big(v_j(n)\big)\,e_j(n)\\ &= a\big(d_j(n)-o_j(n)\big)o_j(n)\big(1-o_j(n)\big)\,, \quad j \text{ is an output neuron}\,,\end{aligned}$$

- and respectively

$$\begin{aligned}\delta_j(n) &= \phi_j'\big(v_j(n)\big)\,\sum_k \delta_k(n)w_{kj}(n)\\ &= ay_j(n)\big(1-y_j(n)\big)\sum_k \delta_k(n)w_{kj}(n)\,, \quad j \text{ is a hidden neuron}\,.\end{aligned}$$

Contents

Contents

Complexity regularization

- In designing a multilayer perceptron by whatever method, we are in effect building **a non-linear model** of the physical phenomenon responsible for the generation of the input-output examples used to train the network.
- Insofar as the network design is statistical in nature, we need an appropriate **tradeoff** between **reliability** of the training data and **goodness** of the model.
- In the context of back-propagation learning, or any other supervised learning procedure for that matter, we may realize this tradeoff by minimizing the **total risk**, expressed as a function of the parameter vector $\boldsymbol{w}$, as follows:

$$R(\boldsymbol{w}) = \mathcal{E}_{av}(\boldsymbol{w}) + \lambda \mathcal{E}_c(\boldsymbol{w}) \,.$$

- $\mathcal{E}_{av}(\boldsymbol{w})$ is the standard **performance metric**, which depends on both the network (model) and the input data, and in back-propagation learning is typically defined as a mean-square error.
- $\mathcal{E}_c(\boldsymbol{w})$ is the **complexity penalty**, where the notion of complexity is measured in terms of the network (weights) alone.
- λ is a **regularization parameter**.

L^2 parameter regularization

- A simple and most common parameter norm penalty is the L^2 parameter norm penalty commonly known as **weight decay**:

$$\mathcal{E}_c(\boldsymbol{w}) \; = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\sum_{k\in\mathcal{C}_{total}} w_k^2 \,.$$

- It is also called *ridge regression* of *Tikhonov regularization*.
- Such a model has the following **total risk**:

$$R(\boldsymbol{w}) = \frac{\lambda}{2}\boldsymbol{w}^\top\boldsymbol{w} + \mathcal{E}_{av}(\boldsymbol{w})\,,$$

- and the corresponding parameter **gradient**:

$$\boldsymbol{g} = \lambda\boldsymbol{w} + \nabla\mathcal{E}_{av}(\boldsymbol{w})\,.$$

- The **weight update** is:

$$\boldsymbol{w}(n+1) = (1-\gamma\lambda)\boldsymbol{w}(n) - \gamma\nabla\mathcal{E}_{av}\big(\boldsymbol{w}(n)\big)\,.$$

- The weight decay term now **multiplicatively shrinks the weight vector** by a constant factor, just **before** performing the usual **gradient update**.

L^1 parameter regularization

- While L^2 weight decay is the most common form of weight decay, another option is to use L^1 regularization.
- L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as:

$$\mathcal{E}_c(\boldsymbol{w}) \;= \|\boldsymbol{w}\|_1 = \sum_{k \in \mathcal{C}_{total}} |w_k| \,.$$

- The **total risk** is given by:

$$R(\boldsymbol{w}) = \lambda \|\boldsymbol{w}\|_1 + \mathcal{E}_{av}(\boldsymbol{w}) \,,$$

- and the corresponding **gradient**:

$$\boldsymbol{g} = \lambda \text{sign}(\boldsymbol{w}) + \nabla \mathcal{E}_{av}(\boldsymbol{w}) \,.$$

- The **weight update** is then:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \gamma\lambda \text{sign}(\boldsymbol{w}) - \gamma \nabla \mathcal{E}_{av}\big(\boldsymbol{w}(n)\big) \,.$$

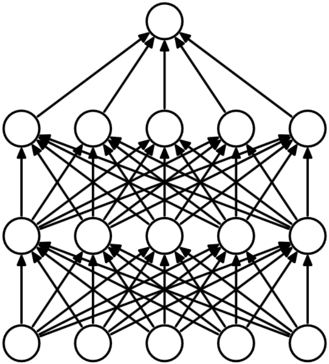
- The regularization **contribution** to the gradient is a **constant factor** with a sign equal (component-wise) to $\text{sign}(w_k)$.
- L^1 regularization introduces **sparsity** (some w_k have optimal value $= 0$). This is used for *feature selection*, see also *LASSO*.

The dropout strategy

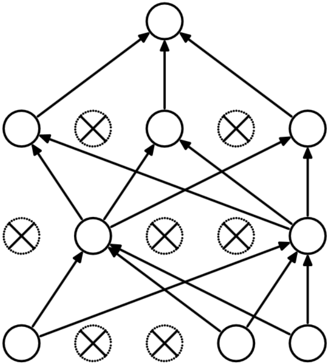
- **Dropout** provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- To a first approximation, dropout can be thought of as a method of making **bagging** practical **for ensembles of** very many large **neural networks**.
- Bagging involves **training multiple models**, and evaluating multiple models on each test example. This **seems impractical** when each model is a large neural network, since training and evaluating such networks is costly in terms of computation time and memory.
- It is common to use ensembles of five to ten neural networks, but more than this rapidly becomes unwieldy.
- Dropout provides an **inexpensive approximation to** training and evaluating a **bagged** ensemble of exponentially many **neural networks**.
- Specifically, **dropout trains the ensemble consisting of all sub-networks** that can be formed by removing non-output neurons from an underlying base network.

The dropout strategy

- An example of applying the dropout strategy:



Standard neural network



After applying dropout

The dropout strategy

- In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively **remove a neuron** from a network **by multiplying its output value by zero**.
- To **train with dropout**, we **use** a **minibatch-based learning** algorithm that makes small steps, such as *stochastic gradient descent*.
- Each time we **randomly sample a** different **binary mask** to apply to all of the input and hidden neurons in the network.
- Typically, an input neuron is included with probability 0.8 and a hidden neuron is included with probability 0.5. These probabilities constitute a **hyperparameter** to be fixed in advance.
- The forward propagation, back-propagation, and the learning update are run as usual.
- To **predict** an unknown observation, we **average the output from many masks**. Even 10–20 masks are often sufficient to obtain good performance.

Contents

Stopping criteria

- In general, back-propagation algorithm *cannot be shown to converge*, and there are no well-defined criteria for stopping its operation.
- A necessary condition for $\boldsymbol{w}^*$ to be a (*local*!) minimum is that the gradient vector $\boldsymbol{g}(\boldsymbol{w})$ of the error surface w.r.t. the weight vector $\boldsymbol{w}$ must be zero at $\boldsymbol{w} = \boldsymbol{w}^*$.
- **The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.**
- But with this criterion learning time may be long.
- **The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.**
- The rate of convergence in the average squared error is typically considered to be small enough if it lies in the range of 0.1 t 1 percent per epoch (0.01 percent per epoch is used as well).

Remarks on the back-propagation algorithm

Heuristics for making the back-propagation algorithm perform better:

- **Stochastic learning** is preferable to the *large batch/entire training sample*, updates especially when the training data sample is large and highly redundant.
- **Maximizing information content**, *e.g. shuffling* observations to ensure that successive examples rarely belong to the same class.
- Correct choice of the **activation function**, *e.g.* a sigmoid one which is *odd in its argument*.
- Choosing **target values** properly w.r.t. the activation function.
- **Normalizing the inputs**: *mean removal*, *decorrelation*, *covariance equalization*.
- **Initialization**: not large (*saturation*!) and not small (*slowing down*!) initial values; *e.g. uniform distribution* with zero mean and variance equal to the reciprocal of the number of synaptic connections of a neuron.
- **Learning from hints**: include *prior information* about your task, *e.g.* sparsity (weight sharing) in the convolutional neural networks.
- **Learning rates**: learning rate can be *smaller in the last layers* than in the front layers.

Thank you for your attention!

Thank you for your attention!

And some references

- Hastie, T., Tibshirani, R., and Friedman, J. (2009).
  The Elements of Statistics Learning: Data Mining, Inference, and Prediction (Second Edition).
  Springer.
- Devroye, L., Györfi, L., Lugosi, G. (1996).
  A Probabilistic Theory of Pattern Recognition.
  Springer.
- Vapnik, V. N. (1998).
  Statistical Learning Theory.
  John Wiley & Sons.
- Haykin, S. (2009).
  Neural Networks and Learning Machines (Third Edition).
  Pearson.
- Goodfellow, J., Bengio, Y., and Courville, A. (2016).
  Deep Learning.
  MIT Press.
- Bishop, C. M. (2006).
  Pattern Recognition and Machine Learning.
  Springer.