Neural Networks

Statistical Methods for Machine Learning

Christian Igel
igel@diku.dk

Department of Computer Science
University of Copenhagen

Warm-up: Gradient

- *Rate of change* of $f : \mathbb{R}^d \to \mathbb{R}$ at a point $\boldsymbol{x} \in \mathbb{R}^d$ when moving in the direction $\boldsymbol{u} \in \mathbb{R}^d$, $\|\boldsymbol{u}\| = 1$, is defined as:

$$\nabla_{\boldsymbol{u}} f(\boldsymbol{x}) = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{u}) - f((x))}{h}$$

- The *gradient*

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \dots, \frac{\partial f(\boldsymbol{x})}{\partial x_d} \right)^{\mathsf{T}}$$

points in the direction $\nabla f(\boldsymbol{x}) / \|\nabla f(\boldsymbol{x})\|$ giving maximum rate $\|\nabla f(\boldsymbol{x})\|$ of change.

Warm-up: Chain rule

The *chain rule* for computing the derivative of a composition of two functions,

$$\frac{\partial f(g(x))}{\partial x} = f'(g(x))g'(x)$$

with $f'(x) = \frac{\partial f(x)}{\partial x}$ and $g'(x) = \frac{\partial g(x)}{\partial x}$, can be extended to:

$$\frac{\partial f(g_1(x), g_2(x), \ldots, g_n(x))}{\partial x} = \sum_{i=1}^{n} \frac{\partial f(g_1(x), \ldots, g_n(x))}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$

Outline

1. Neural Networks
2. Neurons
3. Feed-forward Artificial Neural Networks (NNs)
4. Loss functions and encoding
5. Backpropagation & Gradient-based learning
6. Regularization

Outline

What are Artificial Neural Networks?

> *"There is no universally accepted definition of an NN.But perhaps most people in the field would agree that an NN is a network of many simple processors ("units"), each possibly having a small amount of local memory.The units are connected by communication channels ("connections") which usually carry numeric (as opposed to symbolic) data, encoded by any of various means.The units operate only on their local data and on the inputs they receive via the connections. The restriction to local operations is often relaxed during training. "*

(Artificial) Neural Networks FAQ

Computational neuroscience vs. machine learning

Two applications of neural networks:

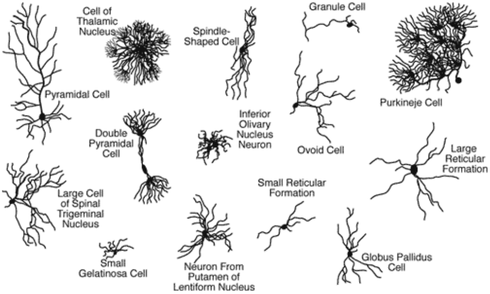
Computational neuroscience: Modelling biological information processing to gain insights about biological information processing

Machine learning: Deriving learning algorithms (loosely) inspired by neural information processing to solve technical problems better than other methods

Outline

Neurons: The shape of things

- Neurons are extremely complex biophysical and biochemical entities coming in a large variety of spatial structures
- To model neurons and especially networks of neurons we must resort to simplifications



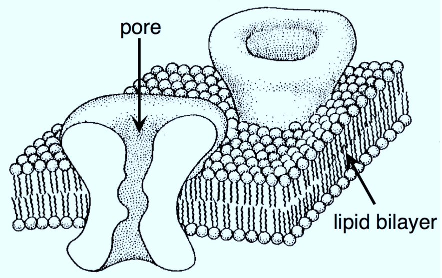
(based on drawings by S. Ramón y Cajal)

Membrane potential

- Membrane potential is essential for neural information processing
- Differences in ion concentrations in- and outside the cell arise from
  - impermeable cell membrane
  - selective (partly voltage dependent) ion channels
  - ion pumps

extracellular
potential defined to be 0 mV

⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕⊕



⊖ ⊖ ⊖ ⊖ ⊖ ⊖ ⊖ ⊖ ⊖⊖

intracellular
resting potential $V_{\text{rest}} < 0\,\text{mV}$

(figure adopted from B. Hille)

Integration and firing

- Integration of incoming signals and generating action potentials ("firing") are basic elements of neuronal information processing
  - Integration: aggregating changes in membrane potential
  - Firing: if membrane potential reaches depolarization level an action potential is triggered
- Neuronal information processing is a spatio-temporal process

Outline

Feed-forward Artificial Neural Networks

Different classes of NNs exist:

- feed-forward NNs $\longleftrightarrow$ recurrent networks
- supervised $\longleftrightarrow$ unsupervised learning

We

- concentrate on feed-forward NNs,
- consider regression and classification,
- just consider supervised learning.

That is, we

- use data to adapt (train) the parameters (weights) of a mathematical model,
- ignore space and time.

Simple neuron models

- Let the input be $x_1, \dots, x_d$ collected in the vector $\boldsymbol{x} \in \mathbb{R}^d$.
- Let the output of our neuron i be denoted by $z_i(\boldsymbol{x})$. Ofter we omit writing the dependency on $\boldsymbol{x}$ to keept the notation uncluttered.
- Integration reduces to computing a weighted sum

$$a_i = \sum_{j=1}^{d} w_{ij} x_j + b_i$$

  with bias (threshold, offset) parameter $b_i \in \mathbb{R}$.
- Firing is simulated by a transfer function (activation function) σ:

$$z_i = \sigma(a_i) = \sigma\left(\sum_{j=1}^{d} w_{ij} x_j + b_i\right)$$

Activation functions

Step / threshold:

$$\sigma(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

Fermi / logistic:

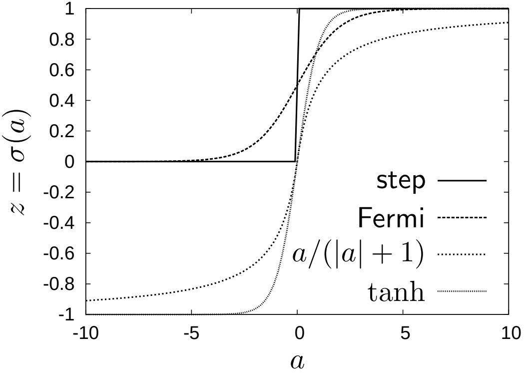
$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

Hyperbolic tangens:

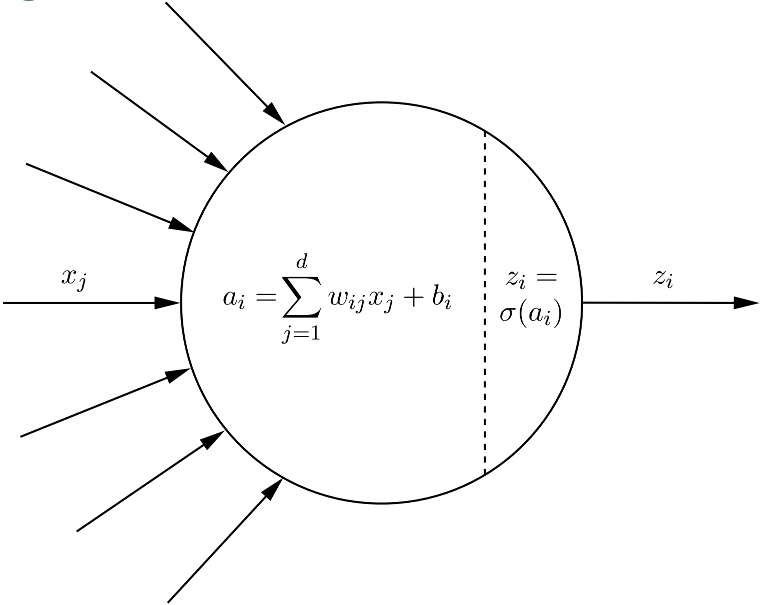
$$\sigma(a) = \tanh(a)$$

Alternative sigmoid:

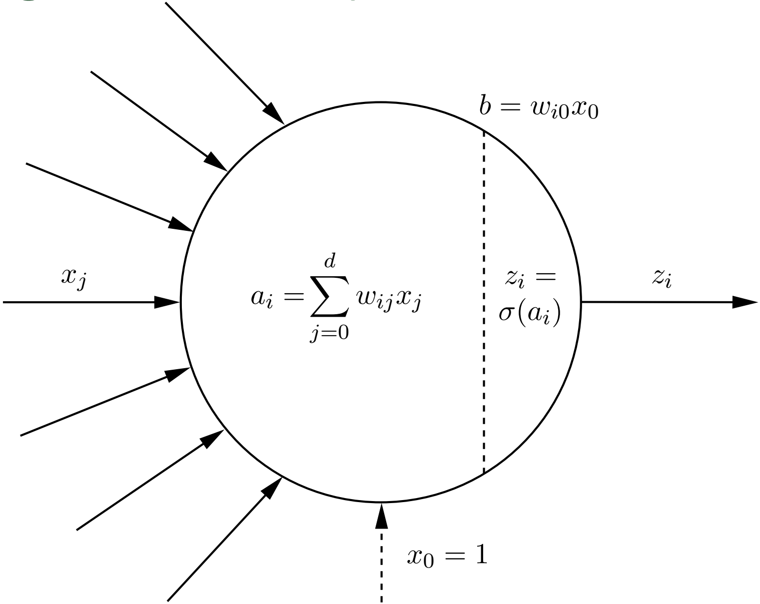
$$\sigma(a) = \frac{a}{1 + |a|}$$



Single neuron with bias



Single neuron with implicit bias



Simple neural network models

- Neural network (NN): set of connected neurons
- NN can be described by a weighted directed graph
  - Neurons are the nodes
  - Connections between neurons are the edges
  - Strength of connection from neuron j to neuron i is described by weight w_{ij}
  - All weights are collected in weight vector $\boldsymbol{w}$
- Neurons are numbered by integers
- Restriction to feed-forward NNs: we do not allow cycles in the connectivity graph
- NN represents mapping

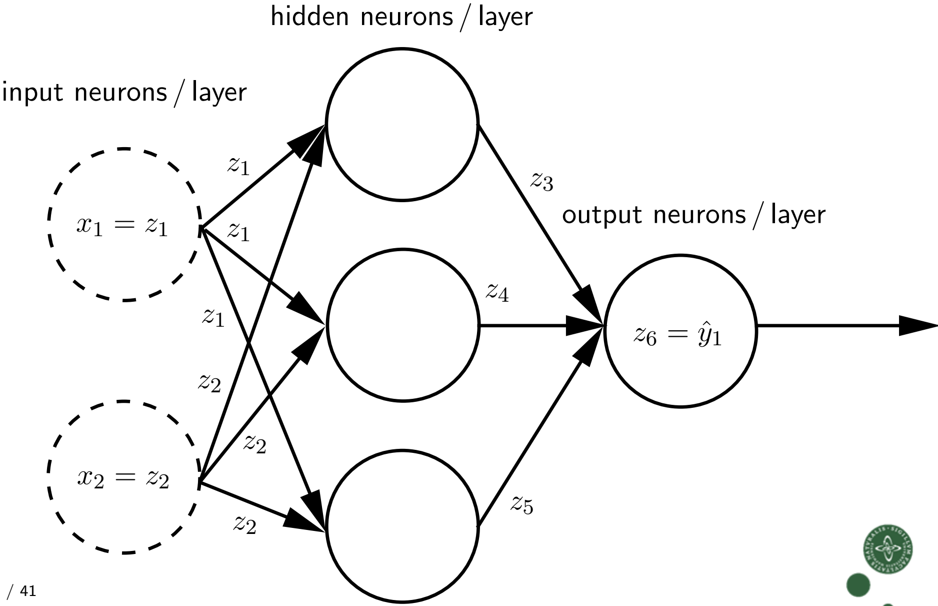
$$f : \mathbb{R}^d \to \mathbb{R}^m$$

parameterized by $\boldsymbol{w}$

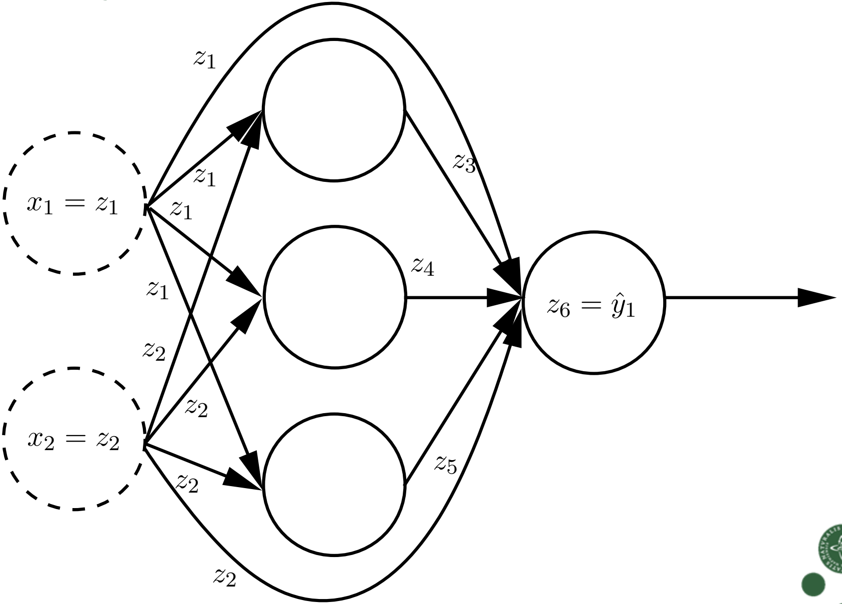
Notation

- Neuron i can get only input from neuron j if $j < i$, this ensures that the graph is acyclic
- Output of neuron i is denoted by z_i
- $z_0(\boldsymbol{x}) = 1$ ($w_{i0}z_0$ is the bias parameter of neuron i)
- $z_1(\boldsymbol{x}) = x_1, \dots, z_d(\boldsymbol{x}) = x_d$ (input neurons)
- $z_i(\boldsymbol{x}) = \sigma_{\text{hidden}}\left(\sum_{0 \le j < i} w_{ij} z_j\right)$ for $d < i \le M - m$
- $z_i(\boldsymbol{x}) = \sigma_{\text{output}}\left(\sum_{0 \le j < i} w_{ij} z_j\right)$ for $i > M - m$ (output neurons)
- $\hat{y}_1 = z_{M-m+1}(\boldsymbol{x}), \dots, \hat{y}_m = z_M(\boldsymbol{x})$
- M neurons in total, d input neurons, m output neurons, $M - d - m$ *hidden* neurons

Multi-layer perceptron network



Multi-layer perceptron network with shortcuts



Outline

Regression

- NN shall learn function

$$f : \mathbb{R}^d \to \mathbb{R}^m$$

$\Rightarrow d$ input neurons, m output neurons

- Training data $S = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \dots, (\boldsymbol{x}_\ell, \boldsymbol{y}_\ell)\}$, $\boldsymbol{x}_i \in \mathbb{R}^d$, $\boldsymbol{y}_i \in \mathbb{R}^m$, $1 \le i \le \ell$
- Sum-of-squares error

$$E = \frac{1}{2} \sum_{n=1}^{\ell} \| f(\boldsymbol{x}_n; \boldsymbol{w}) - \boldsymbol{y}_n \|^2 = \frac{1}{2} \sum_{n=1}^{\ell} \sum_{i=1}^{m} ([f(\boldsymbol{x}_n; \boldsymbol{w})]_i - [\boldsymbol{y}_n]_i)^2$$

- Usually linear output neurons $\sigma_{\text{output}}(a) = a$

Classification

- For binary classification, we use $\mathcal{Y} = \{-1, 1\}$ or $\mathcal{Y} = \{0, 1\}$
- For m-class classification, we use one-hot encoding (1 out of m encoding):
  - $\mathcal{Y} = \mathbb{R}^m$
  - the jth component of y_i is one, if x_i belongs to the jth class, and zero otherwise
  - example: if $m = 4$ and x_i belongs to third class, then $\boldsymbol{y}_i = (0, 0, 1, 0)^\mathsf{T}$
- Use sigmoid σ_{output} with the same range as $\mathcal{Y}$
- Well-working heuristic: combine one-hot encoding and squared error
- Theoretically sound way: minimizing proper negative logarithmic likelihood ($\rightarrow$ "cross-entropy error function")

Outline

Gradient descent

- Consider learning by iteratively changing the weights

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \Delta \boldsymbol{w}^{(t)}$$

- Simplest choice is (steepest) gradient descent

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}}$$

  with learning rate $\eta > 0$

- Often a *momentum term* is added to improve the performance

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}} + \mu \Delta \boldsymbol{w}^{(t-1)}$$

  with momentum parameter $\mu \geq 0$

Backpropagation I

Let g be differentiable. From

$$z_i = \sigma(a_i) \qquad a_i = \sum_{j<i} w_{ij} z_j$$

$$E = \sum_{n=1}^{\ell} E^n \qquad \text{e.g.} \qquad \sum_{n=1}^{\ell} \underbrace{\frac{1}{2} \| \boldsymbol{y}_n - f(\boldsymbol{x}_n \,|\, \boldsymbol{w}) \|^2}_{E^n}$$

we get the with partial derivatives:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{n=1}^{\ell} \frac{\partial E^n}{\partial w_{ji}}$$

In the following, we derive $\frac{\partial E^n}{\partial w_{ij}}$; the index n is omitted to keep the notation uncluttered (i.e., we write E for E^n, $\boldsymbol{x}$ for $\boldsymbol{x}_n$, etc.).

Backpropagation II

We want

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

and define:

$$\delta_i := \frac{\partial E}{\partial a_i}$$

With

$$\frac{\partial a_i}{\partial w_{ij}} = z_j$$

we get:

$$\frac{\partial E}{\partial w_{ij}} = \delta_i z_j$$

Backpropagation III

For an output unit $i \in \{M-m+1,\ldots,M\}$ we have:

$$\delta_i = \frac{\partial E}{\partial a_i} = \frac{\partial z_i}{\partial a_i}\frac{\partial E}{\partial z_i} = \sigma'_{\text{output}}(a_i)\frac{\partial E}{\partial z_i} = \sigma'_{\text{output}}(a_i)\frac{\partial E}{\partial \hat{y}_{i-M+m}}$$

If $\sigma_{\text{output}}(a) = a$, i.e., the output is linear and $\sigma'_{\text{output}}(a) = 1$, and $E = \frac{1}{2}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|^2$, we get:

$$\delta_i = \hat{y}_{i-M+m} - y_{i-M+m}$$

To get the δs for a hidden unit $i \in \{d+1,\ldots,M-m\}$, we need the chain rule again

$$\delta_i = \frac{\partial E}{\partial a_i} = \sum_{k=i+1}^{M}\frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_i} = \sum_{k=i+1}^{M}\frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_i}\frac{\partial z_i}{\partial a_i}$$

and obtain:

$$\delta_i = \sigma'_{\text{hidden}}(a_i)\sum_{k=i+1}^{M} w_{ki}\delta_k$$

Backpropagation IV

For each training pattern $(\boldsymbol{x}, \boldsymbol{y})$:

- *Forward pass (determines output of network given $\boldsymbol{x}$):*
  1. Compute $z_i, \ldots, z_M$ in sequential order
  2. $z_{M-m+1}, \ldots, z_M$ define $\hat{\boldsymbol{y}} = f(\boldsymbol{x} \,|\, \boldsymbol{w})$
- *Backward pass (determines partial derivatives):*
  1. After a forward pass, compute $\delta_i, \ldots, \delta_m$ in reverse order
  2. Compute the partial derivatives according to $\partial E / \partial w_{ij} = \delta_i z_j$

Online vs. batch learning

Consider training set with ℓ patterns and error function

$$E = \sum_{n=1}^{\ell} E^n \qquad \text{e.g.} \qquad \sum_{n=1}^{\ell} \underbrace{\frac{1}{2}(y_n - f(\boldsymbol{x}_n \,|\, \boldsymbol{w}))^2}_{E^n}$$

Batch learning: Compute the gradients over all training samples and do update

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}}$$

Online learning: Choose a pattern $(\boldsymbol{x}_n, \boldsymbol{y}_n)$, $1 \leq n \leq \ell$, (e.g., randomly) and do update

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E_n|_{\boldsymbol{w}^{(t)}}$$

with a smaller learning rate η; using momentum is advisable

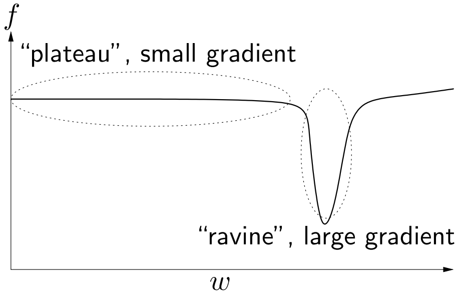
Efficient gradient-based optimization

- Vanilla steepest-descent is usuallynot the best choice for (batch) gradient-based learning
- Many powerful gradient-based search techniques exist
- Simple & robust & fast method: Resilient Backpropagation (RProp)

Riedmiller: Advanced supervised learning in multi-layer perceptrons – From backpropagation to adaptive learning algorithms. *Computer Standards and Interfaces*, 16(5):265–278, 1994

Igel, Hüsken: Empirical evaluation of the improved Rprop learning algorithm, *Neurocomputing*, 50(C):105–123, 2003

Resilient Backpropagation: Basic ideas



Resilient Backpropagation algorithm

Algorithm 1: Rprop algorithm

initialize $\boldsymbol{w}^{(0)}$; $\forall i,j : \Delta_{ij} > 0,\ g_{ij}^{(0)} = 0,\ \eta^{+} = 1.2;\ \eta^{-} = 0.5,\ t \leftarrow 1$
while *stopping criterion not met* **do**
    $g_{ij}^{(t)} = \partial f(\boldsymbol{w}^{(t)})/\partial w_{ij}^{(t)}$
    foreach w_{ij} **do**
        if $g_{ij}^{(t-1)} \cdot g_{ij}^{(t)} > 0$ **then** $\Delta_{ij}^{(t)} \leftarrow \min\left(\Delta_{ij}^{(t-1)} \cdot \eta^{+}, \Delta_{\mathsf{max}}\right)$
        else if $g_{ij}^{(t-1)} \cdot g_{ij}^{(t)} < 0$ **then**
            $\Delta_{ij}^{(t)} \leftarrow \max\left(\Delta_{ij}^{(t-1)} \cdot \eta^{-}, \Delta_{\mathsf{min}}\right)$
        $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \operatorname{sign}\left(g_{ij}^{(t)}\right) \cdot \Delta_{ij}^{(t)}$
    $t \leftarrow t + 1$

RProp features

- ⊕ Robust w.r.t. hyperparameters
- ⊕ Easy to implement
- ⊕ Fast
- ⊕ Independent of magnitude of partal derivatives
  - well-suited for deep architectures
  - well-suited for “noisy” gradients
- ⊖ Cannot detect correlations
- ⊖ Does not work well for online learning

Outline

Weight-decay

- The smaller the weights, the "more linear" is the neural network function.
- Thus, small $\|\boldsymbol{w}\|$ corresponds to smooth functions.
- Therefore, one can penalize large weights by optimizing

$$E + \gamma \frac{1}{2} \|\boldsymbol{w}\|^2$$

  with regularization hyperparameter $\gamma \geq 0$.
- Note: the weights of linear output neurons should not be considered when computing the norm of the weight vector.

Early stopping

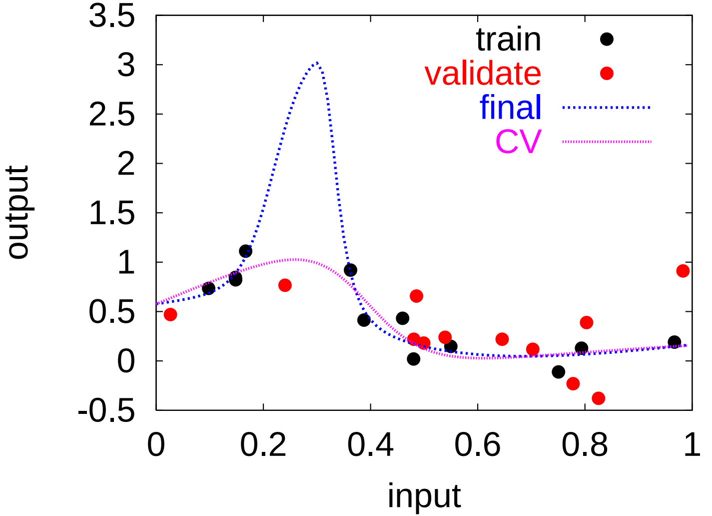
Early-stopping: the learning algorithm

- partitions sample S into training S_{train} and validation S_{val} data
- produces iteratively a sequence of hypotheses

$$h_1, h_2, h_3, \dots$$

  based on S_{train}, ideally corresponding to a nested sequence of hypothesis spaces $\mathcal{H}_1 \subseteq \mathcal{H}_2 \dots$ with $h_i \in \mathcal{H}_i$ and
    - non-decreasing complexity and
    - decreasing empirical risk $\mathcal{R}_{S_{\text{train}}}(h_i) > \mathcal{R}_{S_{\text{train}}}(h_{i+1})$ on S_{train}
- monitors empirical risk $\mathcal{R}_{S_{\text{val}}}(h_i)$ on the validation data
- outputs the hypothesis h_i minimizing $\mathcal{R}_{S_{\text{val}}}(h_i)$.

Early stopping example



Neural network architecture

- Magnitude of the weights is more important for the complexity of the model than number of neurons.
- Depth of network in general increases complexity.
- Training "deep" NNs implementing hierarchical processing is currently an active research field,

The secrets of successful shallow network training

- Normalize the data component-wise to zero-mean and unit variance
- Use a single layer with "enough neurons"
- Start with small weights
- Employ early stopping
- Try shortcuts
- Optimization techniques relying on line search are not recommended, Rprop and steepest-descent may be preferable