Università degli Studi di Pavia

Deep Learning

02-Artificial Neural Networks
Basic Ideas, Notation and all that

Marco Piastra & Andrea Pedrini(*)

(*) Dipartimento di Matematica F. Casorati

This presentation can be downloaded at:
http://vision.unipv.it/DL

Function approximation: Linear Combination

Function Approximation: linear combination

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

a.k.a. "*single layer perceptron*"

A first approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

i.e. this is a vector of dimension d

Note that, when the input is scalar, the approximator becomes

$$\tilde{y} = wx + b$$

i.e. a straight line

Function Approximation: linear combination

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

A first approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

dataset

A set of actual inputs and outputs is all we know about the target function

$$D := \{(\boldsymbol{x}^{(i)},\, y^{(i)})\}_{i=1}^{N}, \qquad y^{(i)} = f^*(\boldsymbol{x}^{(i)}), \forall i$$

Function Approximation: linear combination

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

A first approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

dataset

A set of actual inputs and outputs is all we know about the target function

$$D := \{(\boldsymbol{x}^{(i)},\, y^{(i)})\}_{i=1}^{N}, \qquad y^{(i)} = f^*(\boldsymbol{x}^{(i)}), \forall i$$

Three other fundamental aspects to be considered:

- ***representation***: *which* *<u>parametric approximator</u>* *for a given target function?*
- ***evaluation***: *how do you tell that some parameter values are* *<u>better</u>* *than others?*
- ***optimization***: *how can we* *<u>learn</u>* *optimal values for the parameters?*

Function Approximation: linear combination

- Example: XOR

$$y = \mathrm{XOR}(\boldsymbol{x}), \quad \boldsymbol{x} \in \{0,1\}^2$$

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Dataset:

$$D := \{(\boldsymbol{x}^{(i)},\, y^{(i)})\}_{i=1}^{N}$$

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

this is our *dataset* ($N = 4$)

Function Approximation: linear combination

- Example: XOR

$$y = \mathrm{XOR}(\boldsymbol{x}), \quad \boldsymbol{x} \in \{0,1\}^2$$

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

this is our *dataset* ($N = 4$)

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Dataset:

$$D := \{(\boldsymbol{x}^{(i)},\, y^{(i)})\}_{i=1}^{N}$$

Loss function (evaluation):

$$L(\boldsymbol{x}^{(i)}, y^{(i)}) := (\tilde{y}(\boldsymbol{x}^{(i)}) - y^{(i)})^2$$

Squared Error

$$L(D) := \frac{1}{N} \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in D} L(\boldsymbol{x}^{(i)}, y^{(i)})$$

Mean Squared Error (MSE)

Function Approximation: linear combination

- Example: XOR

$$y = \mathrm{XOR}(\boldsymbol{x}), \quad \boldsymbol{x} \in \{0,1\}^2$$

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

this is our *dataset* ($N = 4$)

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Dataset:

$$D := \{(\boldsymbol{x}^{(i)},\, y^{(i)})\}_{i=1}^{N}$$

Optimization problem:

We need to find

i.e. the set of parameter values that minimizes loss w.r.t. to the dataset

$$(\boldsymbol{w}, b)^* = \underset{(\boldsymbol{w},b)}{\mathrm{argmin}}\ L(D)$$

Function Approximation: linear combination

- Loss minimization

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Loss function:

$$\begin{aligned} L(D) &:= \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{x}^{(i)}, y^{(i)}) \\ &= \frac{1}{N} \sum_{i=1}^{N} (\tilde{y}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{N} \sum_{i=1}^{N} ((\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)})^2 \end{aligned}$$

Can we express this summation by using linear algebra?

*As we will see later on, matrix representation may lead to a better **parallelization** of computations*

Function Approximation: linear combination

- Loss minimization

  Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

  Loss function:

$$L(D) = \frac{1}{N} \sum_{i=1}^{N} ((\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)})^2$$

  define:

$$\boldsymbol{X} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix}$$

input data in matrix form (item index first)

Function Approximation: linear combination

- Loss minimization

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Loss function:

$$L(D) = \frac{1}{N} \sum_{i=1}^{N} ((\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)})^2$$

define:

$$\hat{\boldsymbol{X}} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} & 1 \end{bmatrix} \quad \boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix} \quad \boldsymbol{y} := \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

The loss function becomes:

$$L(D) = \frac{1}{N} (\hat{\boldsymbol{X}} \boldsymbol{\vartheta} - \boldsymbol{y})^2$$

loss function in matrix form

This is a positive-definite quadratic form

Function Approximation: linear combination

- Loss minimization

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Loss function:

$$L(D) = \frac{1}{N} \sum_{i=1}^{N} ((\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)})^2$$

define:

$$\hat{\boldsymbol{X}} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} & 1 \end{bmatrix} \quad \boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix} \quad \boldsymbol{y} := \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

The loss function becomes:

$$L(D) = \frac{1}{N} (\hat{\boldsymbol{X}} \boldsymbol{\vartheta} - \boldsymbol{y})^2$$

loss function in matrix form

This is a positive-definite quadratic form

Function Approximation: linear combination

- Loss minimization

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

Loss function:

$$L(D) = \frac{1}{N}(\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y})^2$$

XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

this is our *dataset* ($N = 4$)

For XOR:

$$\hat{\boldsymbol{X}} := \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} \qquad \boldsymbol{y} := \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Function Approximation: linear combination

- Loss minimization

Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

representation

Loss function:

$$L(D) = \frac{1}{N}(\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y})^2$$

evaluation

Optimization:

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) = 0$$

optimization

the loss function is convex:
by solving this equation we can find $\boldsymbol{\vartheta}^*$
i.e. the optimal parameter values

Function Approximation: linear combination

- Loss minimization

  Approximator: *linear combination*

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

  Optimization:

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) &= \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\vartheta}} (\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y})^2 \\
&= \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\vartheta}} (\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y})^T (\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y}) = \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\vartheta}} (\boldsymbol{\vartheta}^T \hat{\boldsymbol{X}}^T - \boldsymbol{y}^T)(\hat{\boldsymbol{X}}\boldsymbol{\vartheta} - \boldsymbol{y}) \\
&= \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\vartheta}} (\boldsymbol{\vartheta}^T \hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} - \boldsymbol{\vartheta}^T \hat{\boldsymbol{X}}^T \boldsymbol{y} - \boldsymbol{y}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} + \boldsymbol{y}^T \boldsymbol{y}) \\
&= \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\vartheta}} (\boldsymbol{\vartheta}^T \hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} - 2\boldsymbol{\vartheta}^T \hat{\boldsymbol{X}}^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y}) \\
&= \frac{1}{N} (2\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} - 2\hat{\boldsymbol{X}}^T \boldsymbol{y})
\end{aligned}$$

all these terms are scalars

Function Approximation: linear combination

- Loss minimization

  Approximator: *linear combination*

  $$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

  Optimization:

  $$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) = \frac{1}{N}(2\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} - 2\hat{\boldsymbol{X}}^T \boldsymbol{y})$$

  $$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) = 0 \quad \Longrightarrow \quad 2\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} - 2\hat{\boldsymbol{X}}^T \boldsymbol{y} = 0$$

  $$\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} \boldsymbol{\vartheta} = \hat{\boldsymbol{X}}^T \boldsymbol{y}$$

  $$\boldsymbol{\vartheta} = (\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}})^{-1} \hat{\boldsymbol{X}}^T \boldsymbol{y}$$

  this is what we need

  this matrix is SQUARE
  and, typically, with actual datasets,
  is invertible (i.e. full rank)

Function Approximation: linear combination

- Loss minimization

  Approximator: *linear combination*

  $$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

  For XOR:

  $$\boldsymbol{\vartheta} = (\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}})^{-1} \hat{\boldsymbol{X}}^T \boldsymbol{y}$$

XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

$$\hat{\boldsymbol{X}} := \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} \quad \boldsymbol{y} := \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}} = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 4 \end{bmatrix} \quad (\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}})^{-1} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & 0.5 \\ 0.5 & 0.5 & 0.75 \end{bmatrix}$$

$$(\hat{\boldsymbol{X}}^T \hat{\boldsymbol{X}})^{-1} \hat{\boldsymbol{X}}^T \boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 0.5 \end{bmatrix}$$

Function Approximation: linear combination

- Loss minimization

  Approximator: *linear combination*

  $$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$$

  For XOR:

  $$\boldsymbol{\vartheta} := \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.5 \end{bmatrix}$$

  hence the XOR linear approximator becomes:

  $$\tilde{y} = 0.5$$

  What ???

XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Function approximation: Feed-Forward Neural Network

Feed-Forward Neural Network

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

i.e. this is a matrix of dimensions $h \times d$

this is a non-linear scalar function, applied elementwise

Feed-Forward Neural Network

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

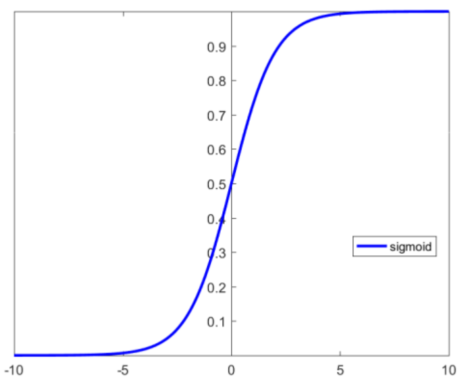
$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

Popular choices for the non-linear function:

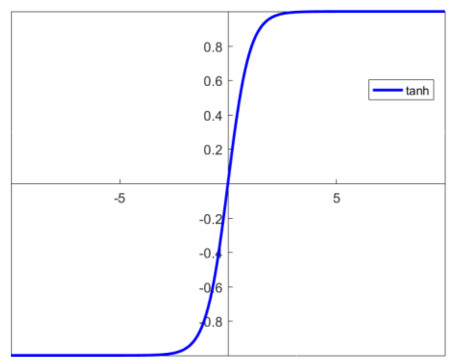
$$g(x) = \sigma(x) = \frac{1}{e^{-x} + 1}$$

$$g(x) = \tanh(x)$$

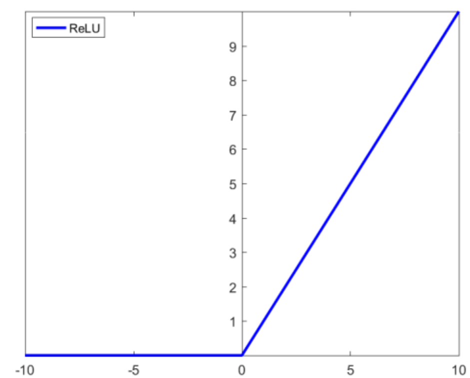
$$g(x) = \max(0, x)$$



Sigmoid



Hyperbolic Tangent



ReLU

Feed-Forward Neural Network

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

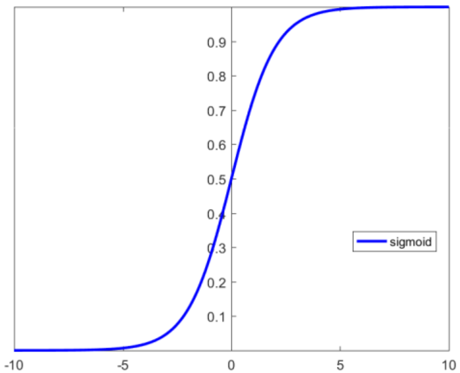
Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

Popular choices for the non-linear function:

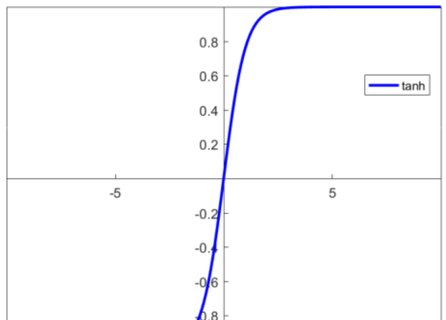
this is somewhat special...

$$g(x) = \sigma(x) = \frac{1}{e^{-x} + 1}$$



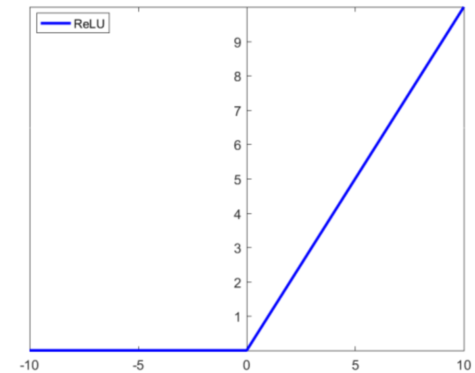
Sigmoid

$$g(x) = \tanh(x)$$



Hyperbolic Tangent

$$g(x) = \max(0, x)$$



ReLU

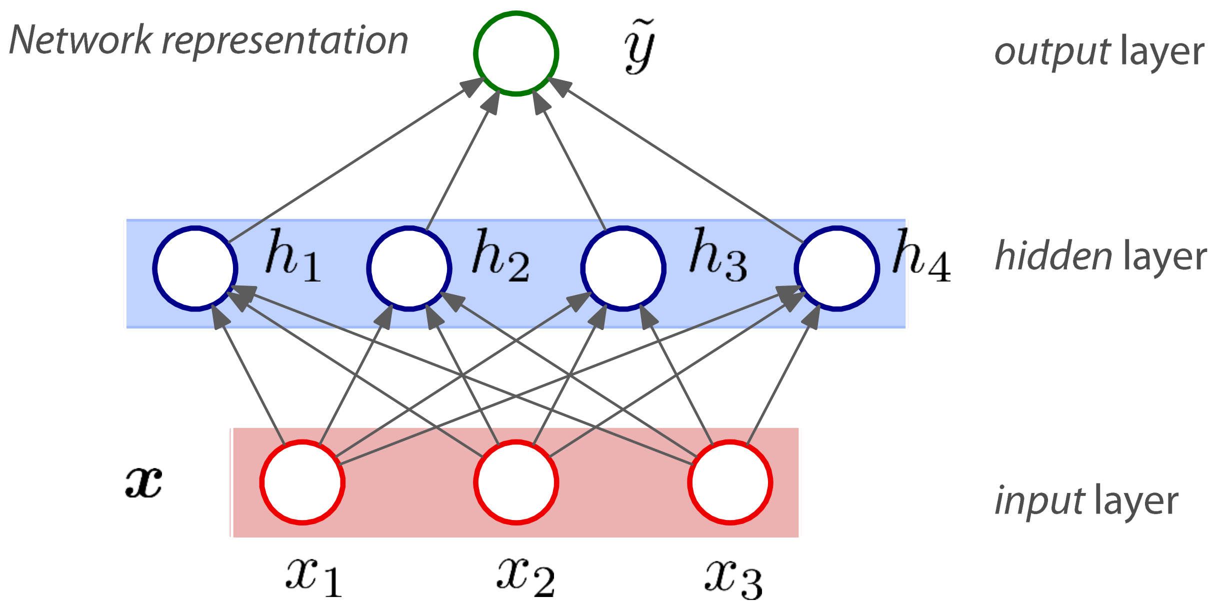
Feed-Forward Neural Network

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \; \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$



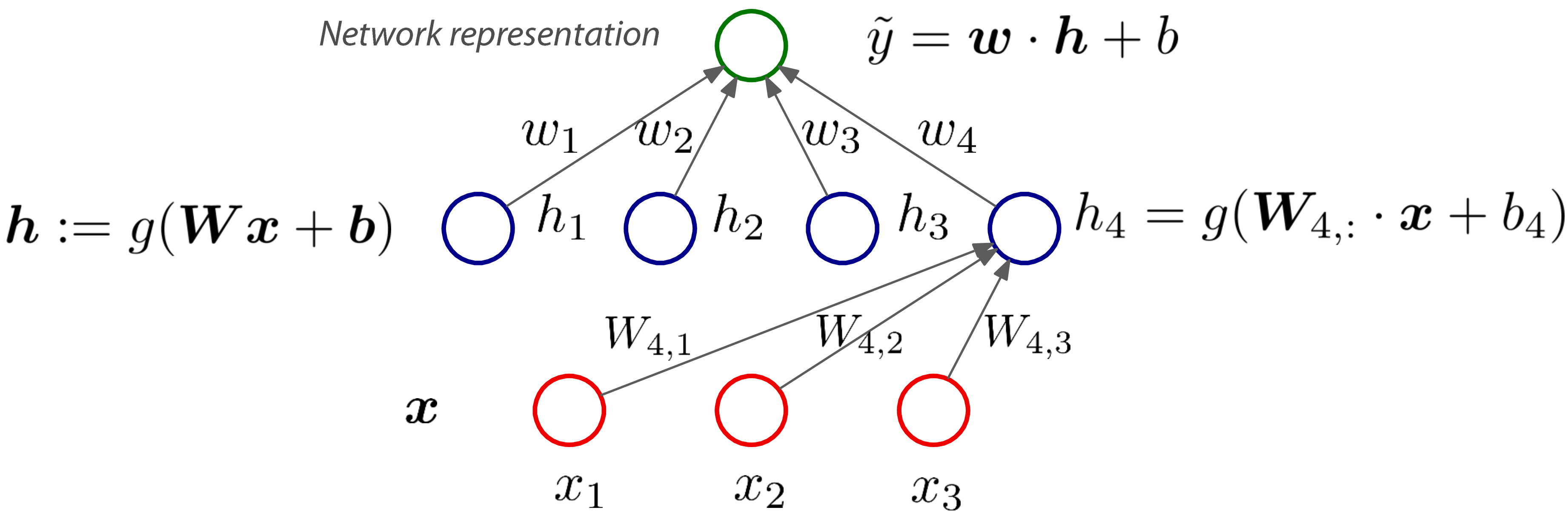
Feed-Forward Neural Network

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d},\ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$



NOTE: *biases* $\boldsymbol{b}$ and b are NOT represented in the graph

Universality of FF Neural Networks

- **Universal approximation theorem** (Cybenko, 1989; Hornik, 1991; Leshno et al. 1991)

  For any target function

  $$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

  (which is continuous and Borel measurable)

  and any $\varepsilon > 0$ there exists parameters

  $$h \in \mathbb{Z}^+, \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

  this is the dimension of the hidden layer: it is a parameter in the theorem

  such that the ***(shallow) feed-forward neural network***

  $$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b$$

  approximates the target function by less than ε

  $$\sup_{\boldsymbol{x}} | f^*(\boldsymbol{x}) - (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b) | < \varepsilon$$

  (on any compact subset of $\mathbb{R}^d$)

  This theorem holds with any of the non-linear functions seen before

Universality of FF Neural Networks

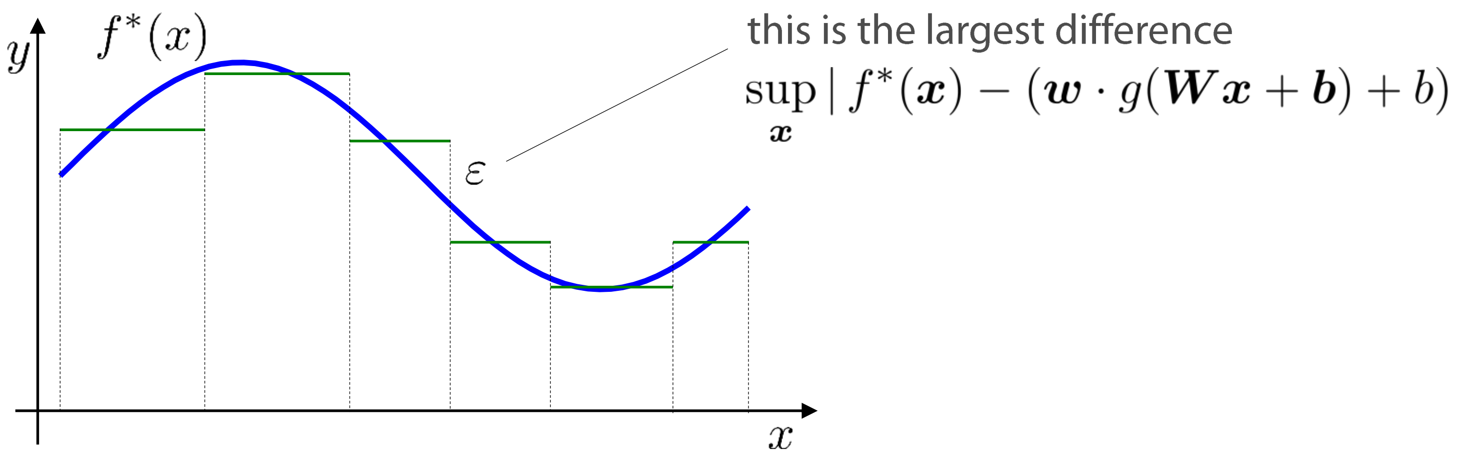
- **Universal approximation theorem** (Cybenko, 1989; Hornik, 1991; Leshno et al. 1991)

  <u>*Intuitive rationale*</u>

  Any continuous target function

  $$y = f^*(x), \quad x \in \mathbb{R}$$

  can be approximated arbitrarily well by a stepwise function



  for simplicity, assume x is *scalar* (hence $\boldsymbol{W}$ is *vector*)

  $$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}x + \boldsymbol{b}) + b$$

Universality of FF Neural Networks

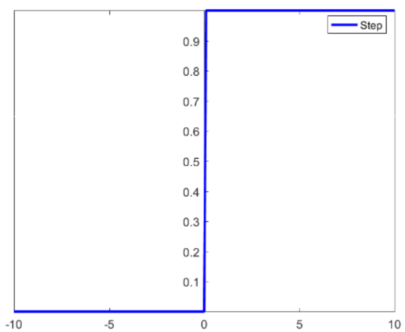
- **Universal approximation theorem** (Cybenko, 1989; Hornik, 1991; Leshno et al. 1991)

<u>*Intuitive rationale*</u>

Consider the *step function* as the non-linearity

$$\tilde{y} = \boldsymbol{w} \cdot \text{step}(\boldsymbol{W}x + \boldsymbol{b}) + b$$

$g(x) = \text{step}(x)$



then, by expanding the approximator

$$\tilde{y} = w_1 \,\text{step}(W_1 x + b_1) + \cdots + w_h \,\text{step}(W_h x + b_h) + b$$

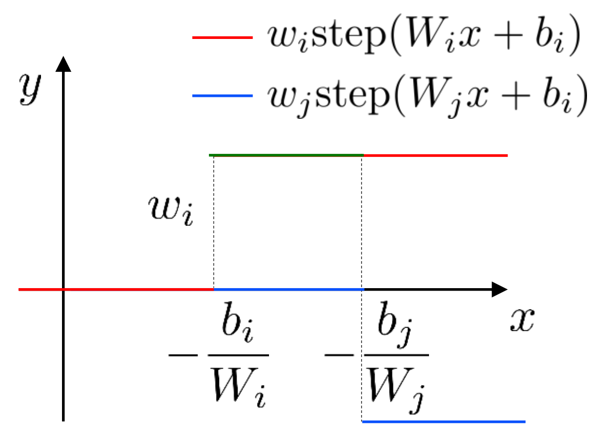
where each step occurs at

$$W_i \cdot x + b_i = 0 \implies W_i \cdot x = -b_i \implies x = -\frac{b_i}{W_i}$$

Consider *pairs* of steps i and j and impose

$$-\frac{b_i}{W_i} < -\frac{b_j}{W_j}, \quad W_i, W_j > 0, \quad w_i = -w_j$$

in this way we can construct $\frac{h}{2}$ *such function steps*



Learning
Feed-Forward Neural Networks

Learning with FF Neural Networks

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

Optimization problem (learning)

Given a dataset $D := \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^N, \quad y^{(i)} = f^*(\boldsymbol{x}^{(i)}), \ \forall i$

the dimension of the hidden layer is pre-defined

we want to find parameter values $\boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$

that *minimize* the loss function $L(D) := \frac{1}{N} \sum_D (\tilde{y}^{(i)} - y^{(i)})^2$

where: $\tilde{y}^{(i)} := \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b$

Learning with FF Neural Networks

- Approximating a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d$$

Second attempt: ***(shallow) feed-forward neural network***

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b, \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \ \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^h, b \in \mathbb{R}$$

Difficulty

In general, *minimizing* the loss function

$$L(D) = \frac{1}{N} \sum_D ((\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b) - y^{(i)})^2$$

this loss function is <u>not</u> convex, in general

cannot be done directly, since

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) = 0$$

cannot be solved analytically

We need to find another way…

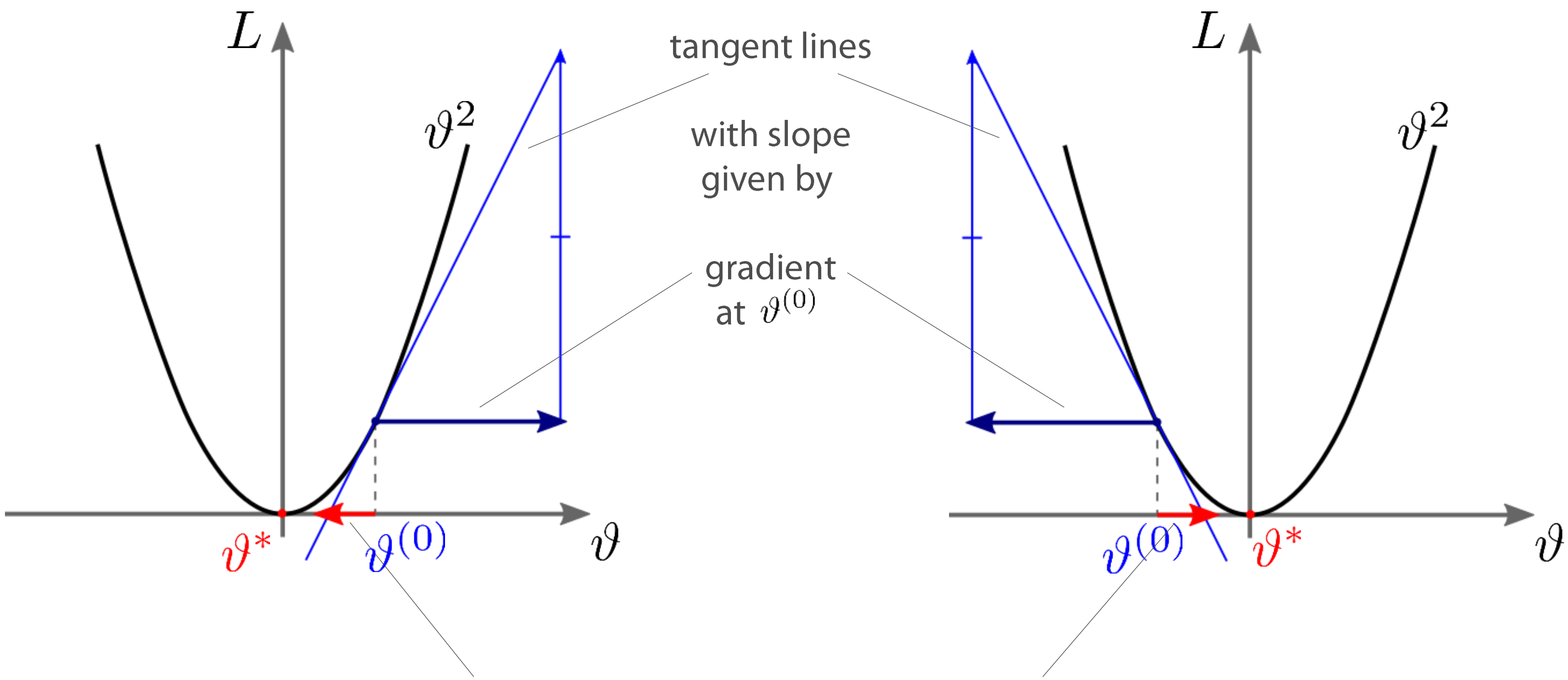
Gradient Descent (GD): intuition

- *Optimization problem*

$$\boldsymbol{\vartheta}^* := \mathrm{argmin}_{\boldsymbol{\vartheta}}\ L(D, \boldsymbol{\vartheta})$$

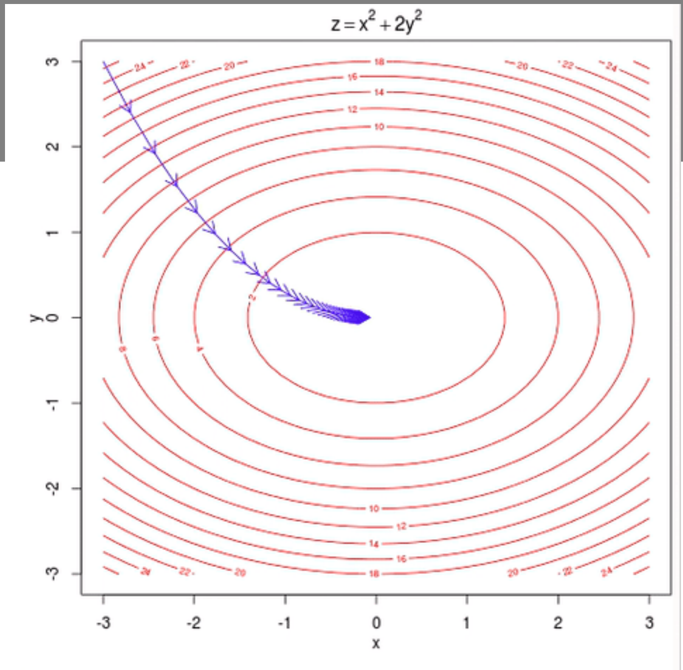
Just making the dependence explicit

- *Minimizing a generic function*



Follow the opposite of the gradient!

Gradient Descent (GD): intuition



- *Optimization problem*

$$\boldsymbol{\vartheta}^* := \text{argmin}_{\boldsymbol{\vartheta}}\ L(D, \boldsymbol{\vartheta})$$

Just making the dependence explicit

- *Iterative method*

Step in the method

1. Initialize $\boldsymbol{\vartheta}^{(0)}$ at random
2. Update $\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \dfrac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}^{(t-1)})$
3. Unless some termination criterion has been met, go back to step 2.

where

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}) := \frac{1}{N} \sum_{D} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(\hat{y}^{(i)}, y^{(i)}, \boldsymbol{\vartheta})$$

The gradient of the loss over the dataset *D* is the average of gradients over each data item

$$\eta \ll 1$$

A *learning rate*, it is arbitrary (i.e. an *hyperparameter*)

Gradient Descent (GD): convergence

- *Convergence*

When $L(D, \boldsymbol{\vartheta})$ is *convex*, *derivable,* and its gradient is *Lipschitz continuous*, that is

$$\left\| \frac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}_1) - \frac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}_2) \right\| \leq C \, \|\boldsymbol{\vartheta}_1 - \boldsymbol{\vartheta}_2\|, \quad C > 0$$

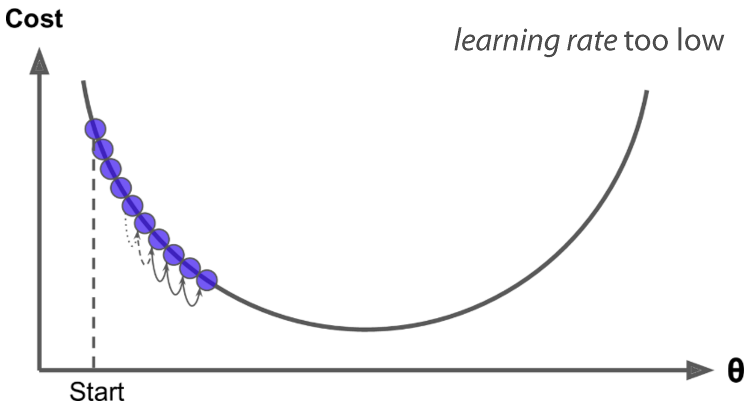
the gradient descent method converges to the optimal $\boldsymbol{\vartheta}^*$ for $t \to \infty$
provided that $\eta \leq 1/C$

When $L(D, \boldsymbol{\vartheta})$ is *derivable* but <u>not</u> *convex,* and its gradient is *Lipschitz continuous,*
the gradient descent method converges to a <u>local minimum</u> of $L(D, \boldsymbol{\vartheta})$
under the same conditions

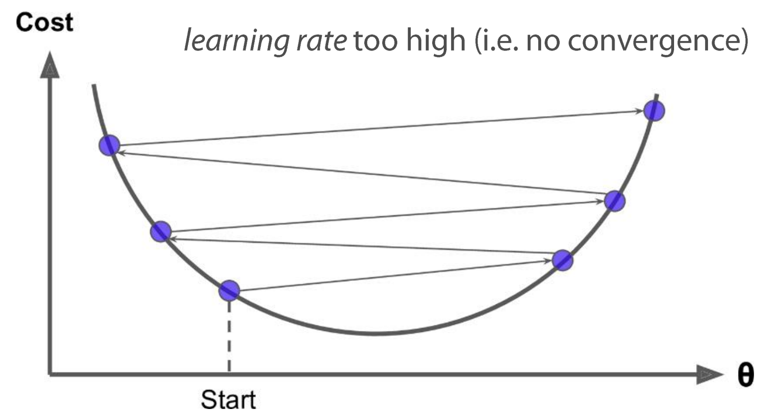
Gradient Descent (GD): practicalities

- *Convergence in practice*

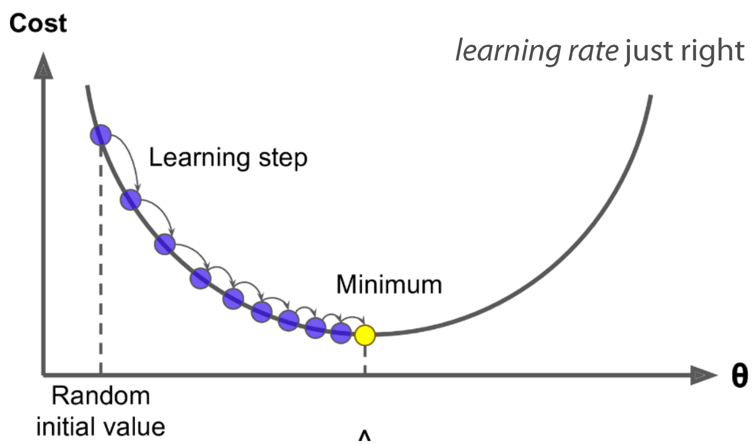
  The choice of the *learning rate* η is crucial



learning rate too low



learning rate too high (i.e. no convergence)



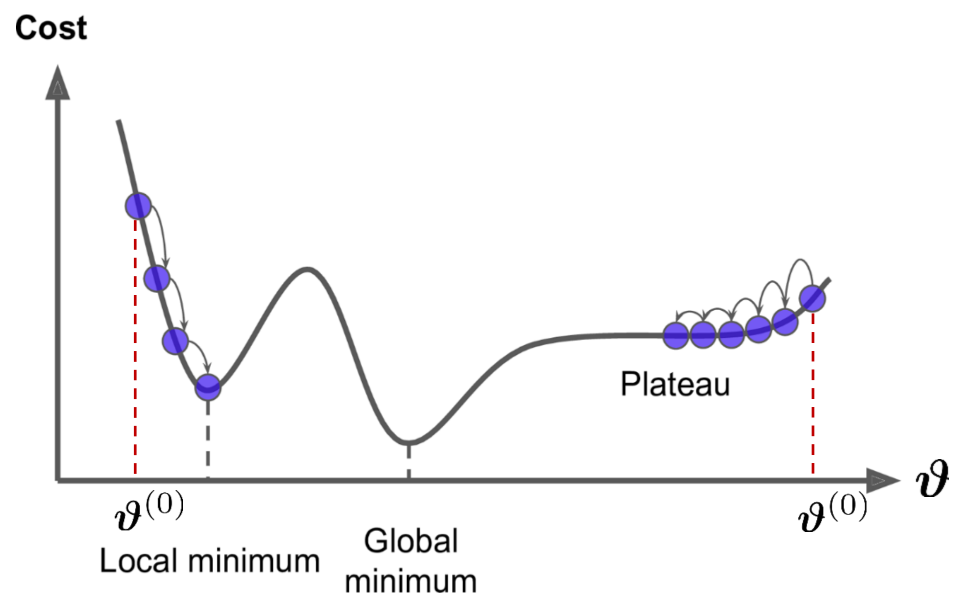
learning rate just right

Images from https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch04.html

Gradient Descent (GD): practicalities

- *Convergence in practice*

  When $L(D, \boldsymbol{\vartheta})$ is <u>not</u> convex, the **initial estimate** $\boldsymbol{\vartheta}^{(0)}$ is crucial



The outcome of the method will depend on which $\boldsymbol{\vartheta}^{(0)}$ *is picked*

Image from https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch04.html

Computing Gradients

Gradient Descent for FF Neural Networks

Recall that the *item-wise* loss for a specific data item in the dataset is

$$L(\tilde{y}^{(i)}, y^{(i)}) := (\tilde{y}^{(i)} - y^{(i)})^2$$

then

$$L(D) = \frac{1}{N} \sum_{D} L(\tilde{y}^{(i)}, y^{(i)})$$

and the gradient of the loss function is

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(D) = \frac{\partial}{\partial \boldsymbol{\vartheta}} \frac{1}{N} \sum_{D} L(\tilde{y}^{(i)}, y^{(i)})$$

$$= \frac{1}{N} \sum_{D} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(\tilde{y}^{(i)}, y^{(i)})$$

Moral: we must be capable to compute the gradient on each data item

Gradient Descent for FF Neural Networks

Suppose we can compute the four *item-wise gradients*, w.r.t. to the parameters:

$$\frac{\partial}{\partial \boldsymbol{W}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \frac{\partial}{\partial \boldsymbol{b}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \frac{\partial}{\partial \boldsymbol{w}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \frac{\partial}{\partial b} L(\tilde{y}^{(i)}, y^{(i)})$$

we can then apply a *gradient descent* method

- ***Gradient Descent***
  1. Assign initial values to the four parameters $\boldsymbol{W}^{(0)},\ \boldsymbol{b}^{(0)},\ \boldsymbol{w}^{(0)},\ b^{(0)}$
  2. Update the four parameters by adding

$$\Delta \boldsymbol{W} = -\eta\ \frac{1}{N} \sum_{D} \frac{\partial}{\partial \boldsymbol{W}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta \boldsymbol{b} = -\eta\ \frac{1}{N} \sum_{D} \frac{\partial}{\partial \boldsymbol{b}} L(\tilde{y}^{(i)}, y^{(i)})$$

$$\Delta \boldsymbol{w} = -\eta\ \frac{1}{N} \sum_{D} \frac{\partial}{\partial \boldsymbol{w}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta b = -\eta\ \frac{1}{N} \sum_{D} \frac{\partial}{\partial b} L(\tilde{y}^{(i)}, y^{(i)})$$

  3. Unless complete, return to step 2.

Computing Gradients

All we need to apply the descent method is computing the item-wise gradients

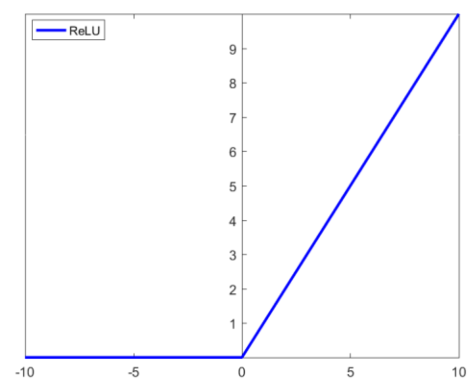
For instance:

$$\frac{\partial}{\partial \boldsymbol{W}} L(\tilde{y}^{(i)}, y^{(i)}) = \frac{\partial}{\partial \boldsymbol{W}} (\tilde{y}^{(i)} - y^{(i)})^2$$

$$= \frac{\partial}{\partial \boldsymbol{W}} ((\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b) - y^{(i)})^2$$

(similar expressions hold for the other three gradients)

$$g(x) = \max(0, x)$$



Assume

$$g(x) = \mathrm{ReLU}(x) := \max(0, x)$$

i.e. the non-linearity is ReLU

Easy, huh?

Learning
Feed-Forward Neural Networks
(contd.)

Function Approximation: FF Neural Networks

- Loss minimization

  Approximator:
  (shallow) feed-forward neural network

$$\tilde{y} = \boldsymbol{w} \cdot \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b$$

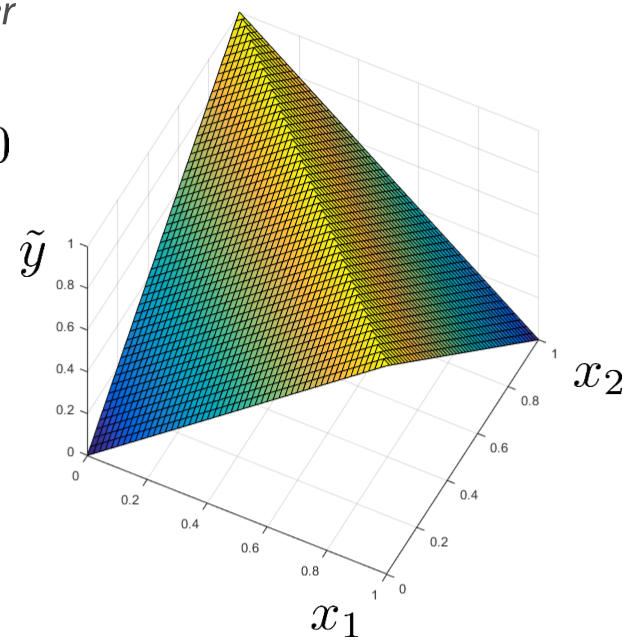
XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Optimal values for XOR and $h = 2$:

dimension of the hidden layer

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad b = 0$$



Function Approximation: FF Neural Networks

- Loss minimization

  Approximator:
  (shallow) feed-forward neural network

  $$\tilde{y} = \boldsymbol{w} \cdot \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + b$$

  In this case our dataset was tiny… ($N = 4$)

  What if the dataset was very large?

XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

this is our *dataset*

Stochastic Gradient Descent for FF Neural Networks

With very large datasets, the sum in:

$$\Delta\boldsymbol{\vartheta} = -\eta\,\frac{1}{N}\sum_{D}\frac{\partial}{\partial\boldsymbol{\vartheta}}L(\tilde{y}^{(i)}, y^{(i)})$$

may take very long to compute (and this must be repeated at each iteration)

- ***Stochastic*** **Gradient Descent (SGD)** (i.e. *"you don't actually need to sum up them all"*)
  1. Assign initial values to the four parameters $\boldsymbol{W}^{(0)},\ \boldsymbol{b}^{(0)},\ \boldsymbol{w}^{(0)},\ b^{(0)}$
  2. Pick up a data item $(\boldsymbol{x}^{(i)}, y^{(i)})$ from D with uniform probability and update the four parameters (with $\eta \ll 1.0, \quad \eta \to 0$ as iterations progress)

$$\Delta\boldsymbol{W} = -\eta\,\frac{\partial}{\partial\boldsymbol{W}}L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta\boldsymbol{b} = -\eta\,\frac{\partial}{\partial\boldsymbol{b}}L(\tilde{y}^{(i)}, y^{(i)})$$

$$\Delta\boldsymbol{w} = -\eta\,\frac{\partial}{\partial\boldsymbol{w}}L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta b = -\eta\,\frac{\partial}{\partial b}L(\tilde{y}^{(i)}, y^{(i)})$$

  3. Unless complete, return to step 2.

Stochastic Gradient Descent (SGD): convergence

- *Convergence*

When $L(D, \boldsymbol{\vartheta})$ is *convex*, *derivable,* and its gradient is *Lipschitz continuous*, that is

$$\left\| \frac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}_1) - \frac{\partial}{\partial \boldsymbol{\vartheta}} L(D, \boldsymbol{\vartheta}_2) \right\| \leq C \, \|\boldsymbol{\vartheta}_1 - \boldsymbol{\vartheta}_2\|, \quad C > 0$$

the stochastic gradient descent method converges to the optimal $\boldsymbol{\vartheta}^*$ for $t \to \infty$ provided that

$$\eta^{(t)} \leq \frac{1}{Ct}$$

Note that $\eta^{(t)} \to 0$ for $t \to \infty$

When $L(D, \boldsymbol{\vartheta})$ is *derivable,* and its gradient is *Lipschitz continuous* but not *convex* the stochastic gradient descent method converges to a local minimum of $L(D, \boldsymbol{\vartheta})$ under the same conditions

Speed of Convergence

Perhaps surprisingly, ***stochastic gradient descent*** *shares the same properties and could be faster than GD …*

Consider a generic loss function $L(\boldsymbol{\vartheta})$ which is *convex* in the parameter $\boldsymbol{\vartheta}$

Define *accuracy* as an upper bound:

optimal value

current parameter estimate

$$|L(\boldsymbol{\vartheta}^*) - L(\tilde{\boldsymbol{\vartheta}})| < \rho$$

N size of the dataset

q number of (scalar) parameters in $\boldsymbol{\vartheta}$

[*from Bottou & Bousquet, 2008*]

Algorithm	***Cost per iteration***	***Iterations to reach accuracy ρ***	***Time to reach accuracy ρ***
Gradient descent (GD)	$\mathcal{O}(N\,q)$	$\mathcal{O}\left(\log \frac{1}{\rho}\right)$	$\mathcal{O}\left(N\,q \log \frac{1}{\rho}\right)$
Stochastic gradient descent (SGD)	$\mathcal{O}(q)$	$\mathcal{O}\left(\frac{1}{\rho}\right)$	$\mathcal{O}\left(q \frac{1}{\rho}\right)$

Mini-batch Gradient Descent for FF Neural Networks

- ***Mini-batch*** **Gradient Descent (MBGD)**

1. Assign initial values to the four parameters $\boldsymbol{W}^{(0)},\ \boldsymbol{b}^{(0)},\ \boldsymbol{w}^{(0)},\ b^{(0)}$
2. Pick a *mini-batch* $B \subseteq D$ with uniform probability
   and update the four parameters (with $\eta \ll 1.0, \quad \eta \to 0$ as iterations progress)

$$\Delta \boldsymbol{W} = -\eta \, \frac{1}{|B|} \sum_{B} \frac{\partial}{\partial \boldsymbol{W}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta \boldsymbol{b} = -\eta \, \frac{1}{|B|} \sum_{B} \frac{\partial}{\partial \boldsymbol{b}} L(\tilde{y}^{(i)}, y^{(i)})$$

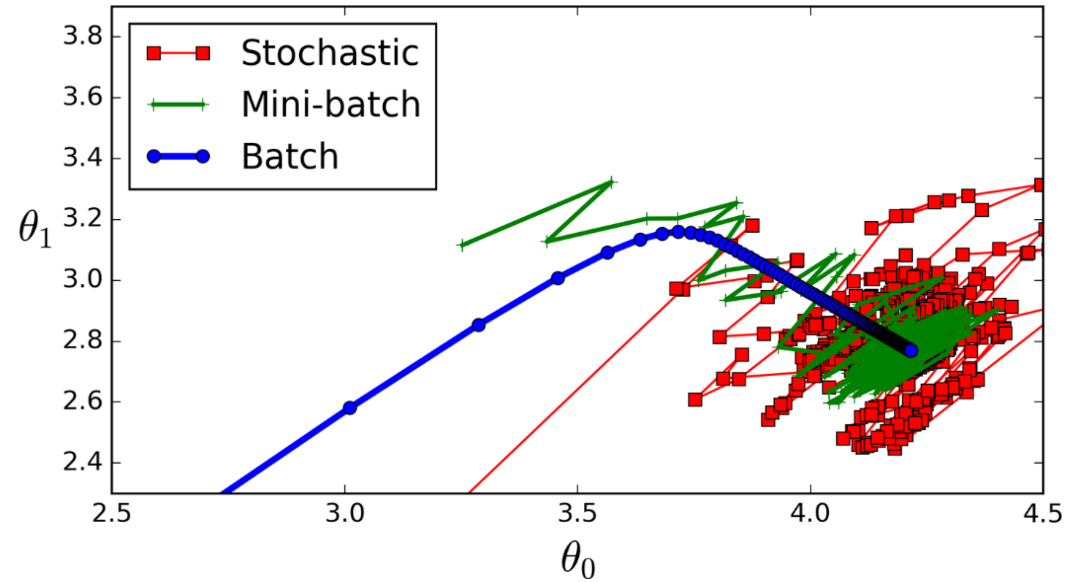
$$\Delta \boldsymbol{w} = -\eta \, \frac{1}{|B|} \sum_{B} \frac{\partial}{\partial \boldsymbol{w}} L(\tilde{y}^{(i)}, y^{(i)}) \qquad \Delta b = -\eta \, \frac{1}{|B|} \sum_{B} \frac{\partial}{\partial b} L(\tilde{y}^{(i)}, y^{(i)})$$

3. Unless complete, return to step 2.

This method has the same convergence properties of SGD

Qualitative comparison of GD methods

Typical traces
of the three methods
(batch = GD)



In general:

- GD is more regular but slower (with large datasets)
- SGD is faster (with large datasets) but noisy
- MBGD is often the right compromise in practice…

Image from https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch04.html