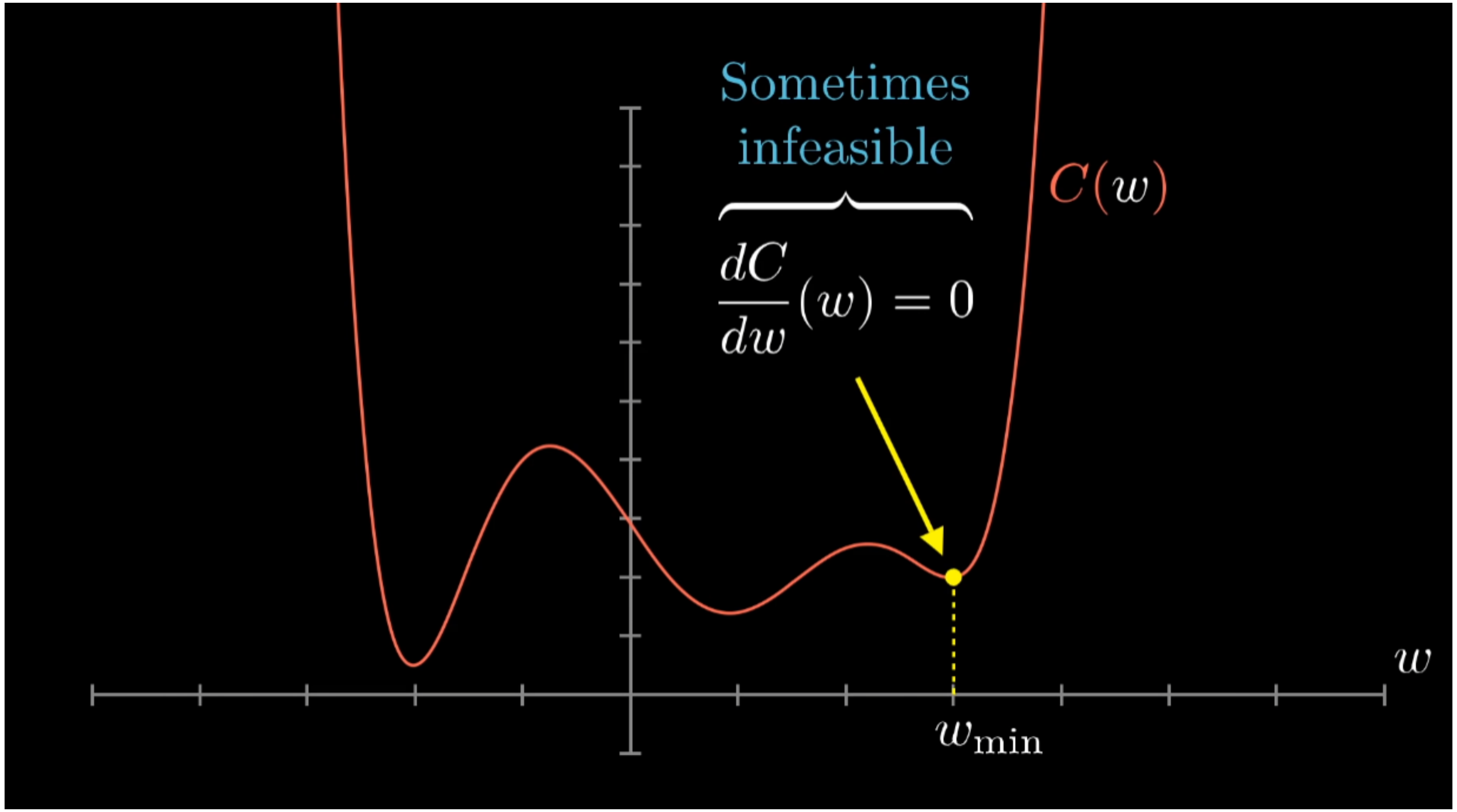
Discesa del gradiente e back-propagation

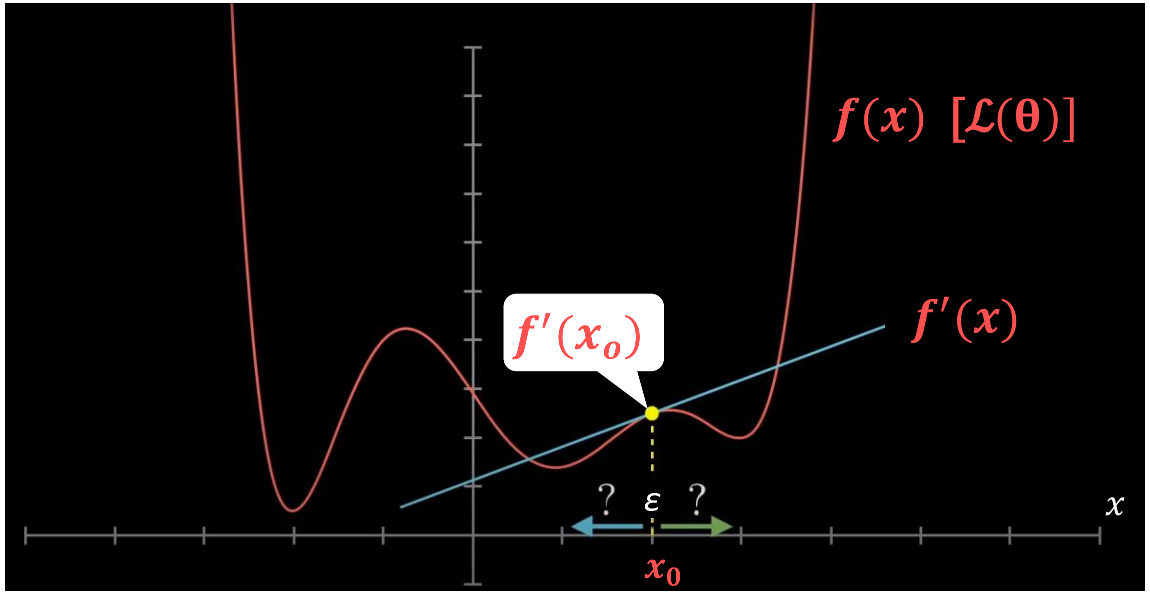
Il calcolo del gradiente nelle Reti Neurali

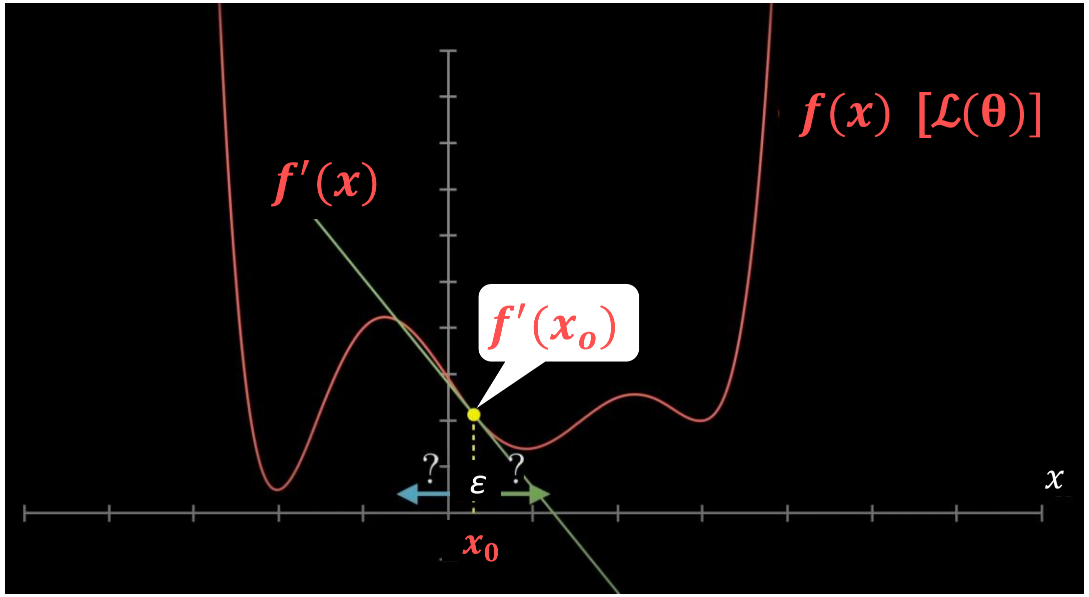


Infattibile con $10^3 - 10^9$ variabili!

Sebbene esistano formule analitiche per calcolare il gradiente, spesso è complesso da un punto di vista computazionale per funzioni con molte variabili. Vedi le slide successive, che mostrano superfici di loss vere (2D!)

Le derivate prime nell'univariato: come usarle per seguire la funzione *downhill*





Fonte: 3blue1brown series

$\boldsymbol{\varepsilon f'(x_0)}$

che segno??

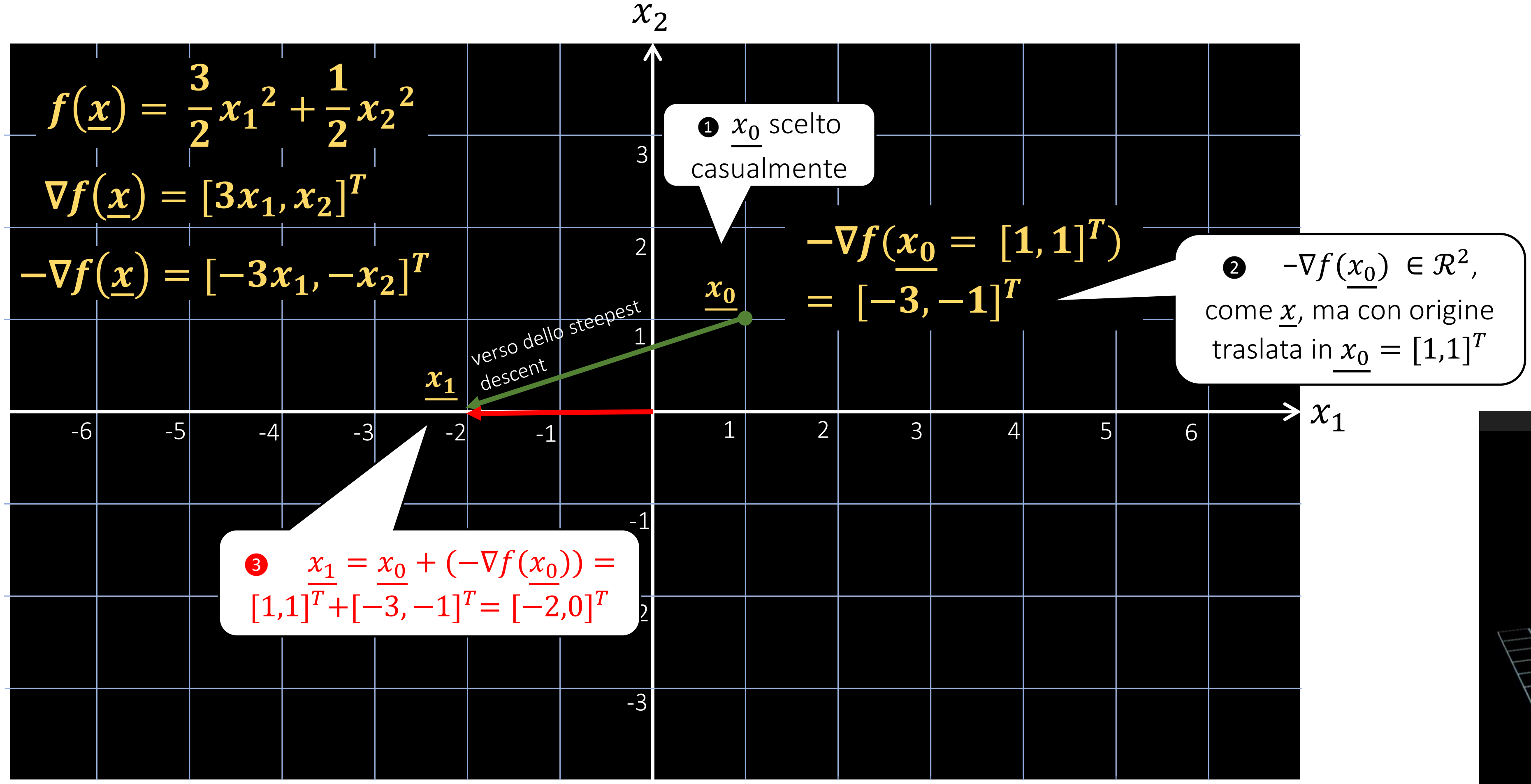
Punto chiave: La derivata di una funzione univariata è utile per minimizzarne il valore perché ci dice come modificare x per ridurre un pò y (step infinitesimali). Cioè, in che direzione andare! (aumentare o diminuire x?). In gergo, si dice «muoversi nella direzione della derivata prima»: cioè ε positivo con $f'(x)$ positiva ed ε negativo con $f'(x)$ negativa.

Ciò vale, con qualche modifica, anche nel multi-variato (si usa la derivata direzionale).

Se gli step size sono proporzionali allo slope della tangente, mano a mano che ci avviciniamo al punto critico, lo step diminuisce e così si evita l'*overshooting* *[3blue1brown]*.

L'algoritmo della discesa del gradiente – 3/20

il piano $(x_1, x_2) = \mathcal{R}^2$ è il $\underline{\theta}$ space



$$f(\underline{x}) = \frac{3}{2}{x_1}^2 + \frac{1}{2}{x_2}^2$$

$$\nabla f(\underline{x}) = [3x_1, x_2]^T$$

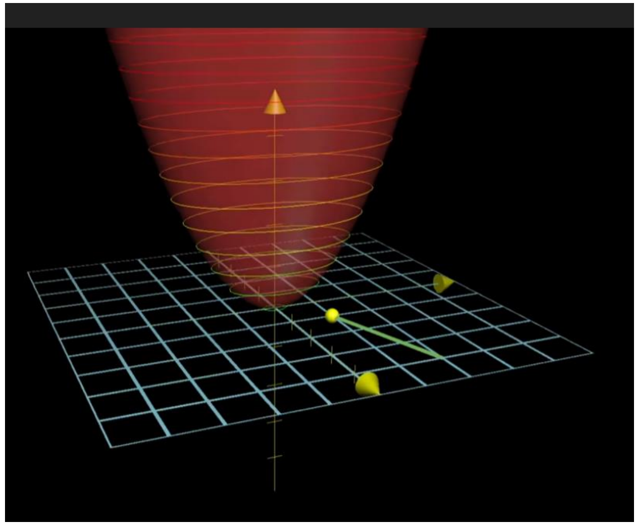
$$-\nabla f(\underline{x}) = [-3x_1, -x_2]^T$$

❶ $\underline{x_0}$ scelto casualmente

$$-\nabla f(\underline{x_0} = [1,1]^T) = [-3,-1]^T$$

❷ $-\nabla f(\underline{x_0}) \in \mathcal{R}^2$, come $\underline{x}$, ma con origine traslata in $\underline{x_0} = [1,1]^T$

❸ $\underline{x_1} = \underline{x_0} + (-\nabla f(\underline{x_0})) = [1,1]^T + [-3,-1]^T = [-2,0]^T$



Ricordare: $\nabla \in \mathcal{R}^2, \notin \mathcal{R}^3$

(f esemplificativa, non è quella di sx)

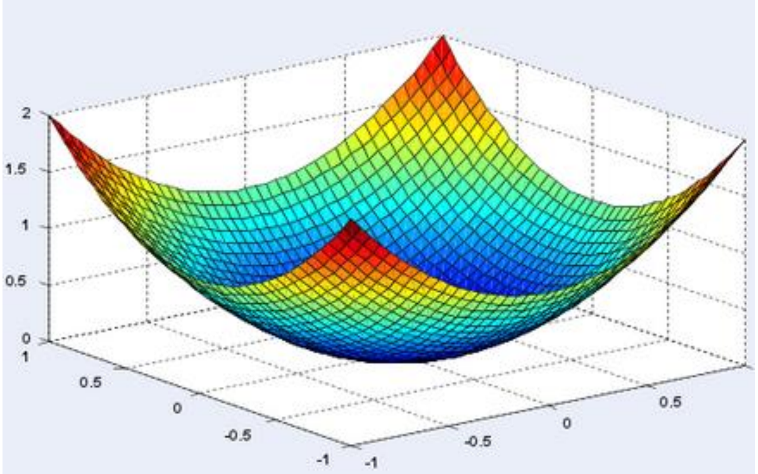
Semplice esempio con un polinomio di ordine 2 $[p = 2 \equiv dim(\theta)]$ ed $\eta = 1$ (per semplicità), non è una vera loss function.

NB. La norma (euclidea) di $\nabla f(\underline{x_0})$, cioè: $|\nabla f(\underline{x_0})|_2$, qui $\sqrt{{x_1}^2 + {x_2}^2} = \sqrt{(-3)^2 + (-1)^2} = \sqrt{10}$ è la ripidità.

Are all quadratic programming problems convex? (Quora)

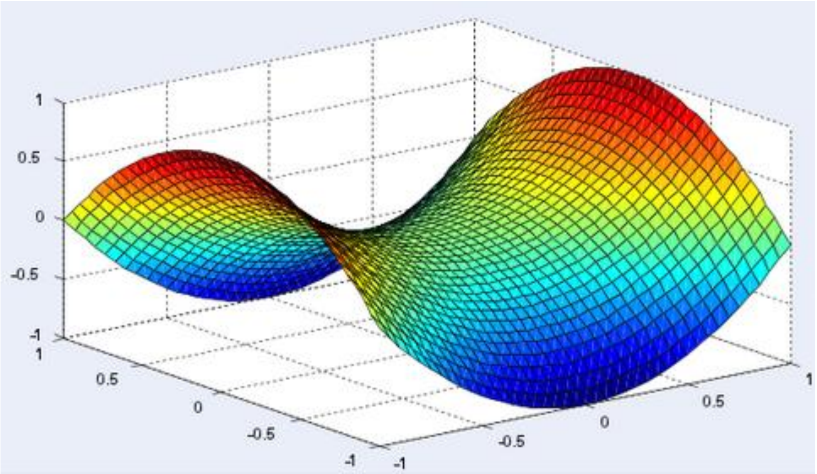
The function x^TAx defines a quadratic surface. If A is positive definite then the surface looks like a bowl. So like this:

Paraboloide (ellittico)



If A is indefinite then the surface looks like some kind of saddle ("sella"), so like:

Paraboloide (iperbolico)



And if the surface is negative definite then it looks like an upside down bowl.

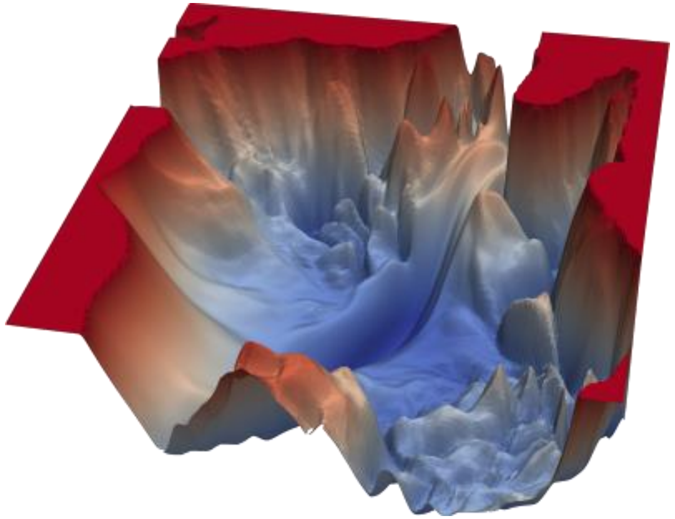
if you are maximizing x^TAx under box constraints and A is positive definite (so it looks like a bowl) then each corner of the box is a different local maximizer, so again it's not convex obviously. It's also really difficult since the number of corners is exponential in the number of dimensions.

Video sulle Forme Quadratiche (ciotole e selle)

E' la presenza di punti di sella che rende una funzione quadratica non-convessa.

Are all quadratic programming problems convex? (Quora)

If A is indefinite then the surface looks like some kind of saddle (“sella”), so like:



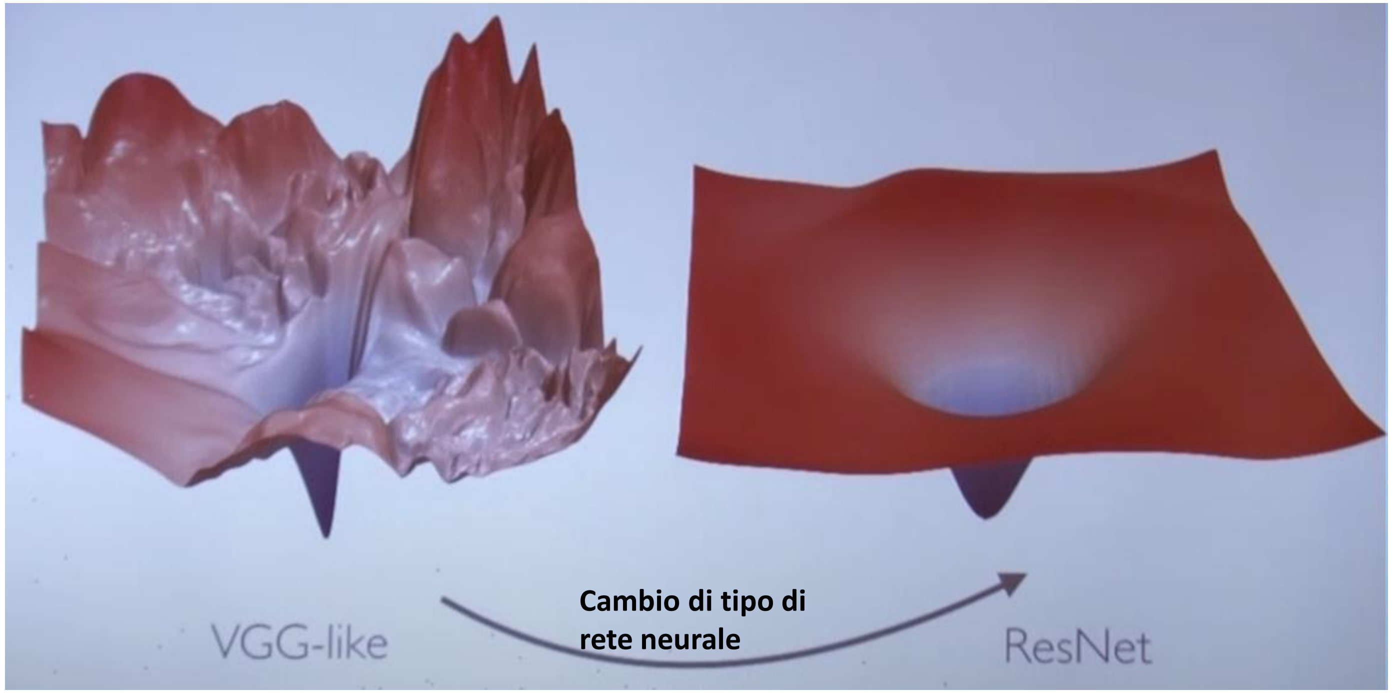
https://i.stack.imgur.com/4H2vs.png

Example of real-life, non-convex loss landscape (from a 56 layer NN fitted over the SyFy10 dataset). It looks like a very irregular valley in the mountains, with a lot of ups and downs, many smaller valleys and peaks. Clearly non-convex.

E’ la presenza di punti di sella che rende una funzione quadratica non-convessa.

Surface plot (veri) della loss function di due semplici reti neurali

(Le due dimensioni sono scelte casualmente, ma riescono a render bene la non-convessità del paesaggio)



Cambio di tipo di rete neurale

Rete neurale convolutiva a 56 livelli. Il minimo centrale è circondato da una regione molto rischiosa (selle e minimi locali)

Rete neurale Res-Net (con skip delle connessioni), più facile da fittare. Sorprendentemente, il paesaggio è molto diverso.

La surface della loss function, dalla quale dipende la facilità/difficoltà del fit della rete, può dipendere anche dall'architettura della rete (e dai dati, ovviamente).

https://www.youtube.com/watch?v=78vq6kgsTa8

La back-propagation

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta) = \sum_{i=1}^{n} \frac{1}{2}(y_i - f_\theta(x_i))^2$$

[*l'half MSE* ,più facile da derivare]

dove, per l'osservazione i-esima:

$$\mathcal{L}_i(\theta) = \frac{1}{2}(y_i - f_\theta(x_i))^2 = \frac{1}{2}(y_i - \beta_0 \; - \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}))^2$$

derivando rispetto a β_k (con la *chain rule*):

$$\boxed{\frac{\partial \mathcal{L}_i(\theta)}{\partial \boldsymbol{\beta}_k} = \frac{\partial \mathcal{L}_i(\theta)}{\partial \boldsymbol{f}_\theta(\boldsymbol{x}_i)} \frac{\partial \boldsymbol{f}_\theta(\boldsymbol{x}_i)}{\partial \boldsymbol{\beta}_k} = \boldsymbol{-(y_i - f_\theta(x_i))g(z_{ik})}}$$

una frazione del residuo è attribuita ad ogni peso…

…proporzionalmente alla funzione *g*

derivando rispetto a w_{kj} (con la *chain rule*) :

$$\boxed{\frac{\partial \mathcal{L}_i(\theta)}{\partial w_{kj}} = \frac{\partial \mathcal{L}_i(\theta)}{\partial f_\theta(x_i)} \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \frac{\partial g(z_{ik})}{\partial z_{ik}} \frac{\partial z_{ik}}{\partial w_{kj}} = \boldsymbol{-(y_i - f_\theta(x_i))\, \beta_k\, g'(z_{ik}) x_{ij}}}$$

Entrambe le derivate contengono il **residuo** $\boldsymbol{y_i - f_\theta(x_i)}$.

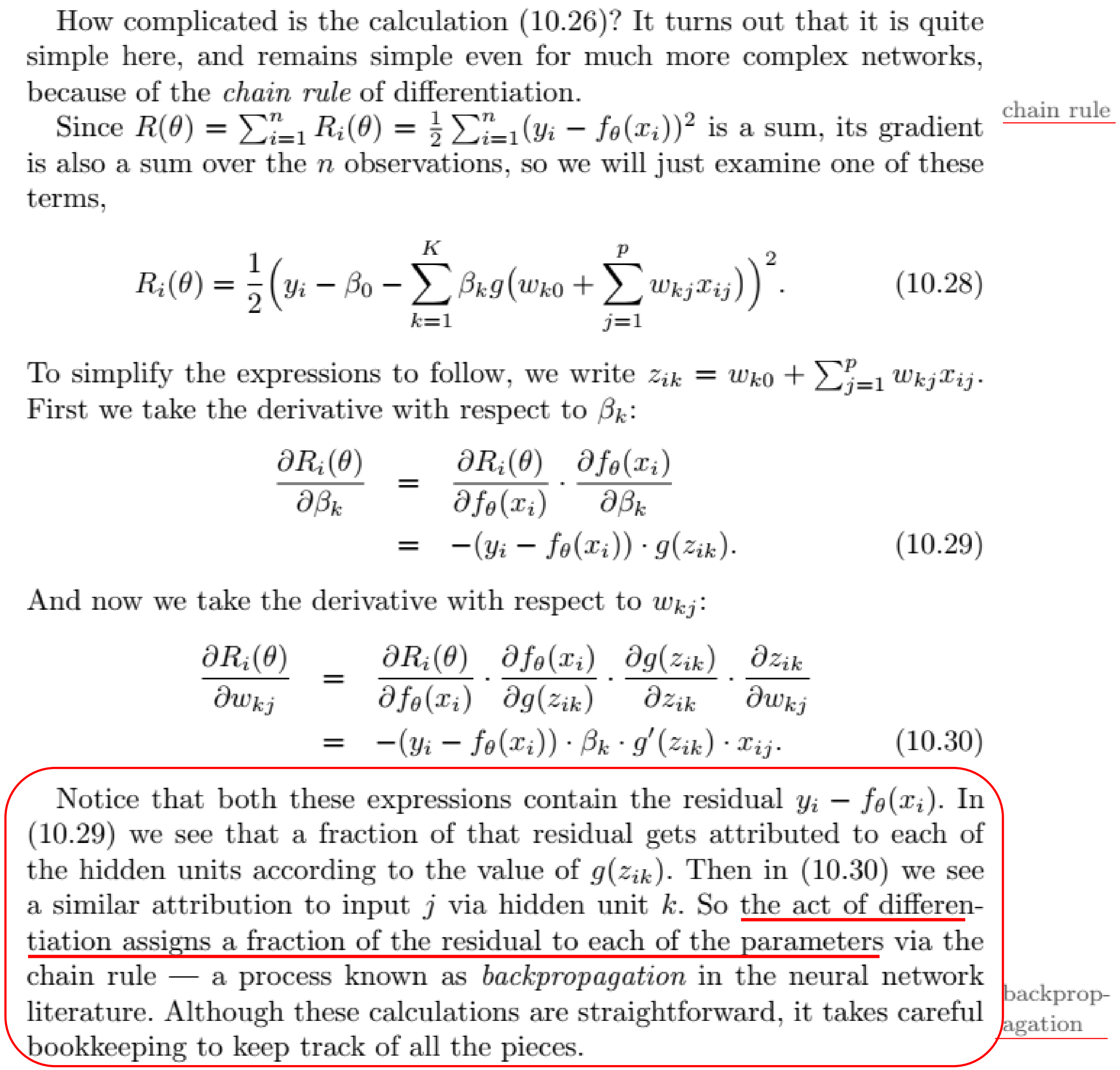
In quella rispetto a β_k una frazione di questo residuo risulta attribuita ad ognuno dei nodi nascosti (quelli appunto in comb.lin. con β_k nella funzione f) a seconda del valore di $g(z_{ik})$.

Nella derivata rispetto a w_{kj} una analoga attribuzione avviene tramite il nodo nascosto k.

In questo modo, per mezzo della chain rule, l'operazione della differenziazione assegna una frazione del residuo ad ognuno dei parametri.

Questo processo è noto appunto come back-propagation dell'errore (il residuo).

Nella SGD ogni parametro del modello riceve a ogni iterazione un aggiornamento proporzionale alla derivata parziale della funzione di costo rispetto al parametro stesso (Wikipedia IT, «Problema della scomparsa dl gradiente»).



How complicated is the calculation (10.26)? It turns out that it is quite simple here, and remains simple even for much more complex networks, because of the *chain rule* of differentiation.

chain rule

Since $R(\theta) = \sum_{i=1}^n R_i(\theta) = \frac{1}{2}\sum_{i=1}^n (y_i - f_\theta(x_i))^2$ is a sum, its gradient is also a sum over the n observations, so we will just examine one of these terms,

$$R_i(\theta) = \frac{1}{2}\Big(y_i - \beta_0 - \sum_{k=1}^K \beta_k g\big(w_{k0} + \sum_{j=1}^p w_{kj}x_{ij}\big)\Big)^2. \qquad (10.28)$$

To simplify the expressions to follow, we write $z_{ik} = w_{k0} + \sum_{j=1}^p w_{kj}x_{ij}$. First we take the derivative with respect to β_k:

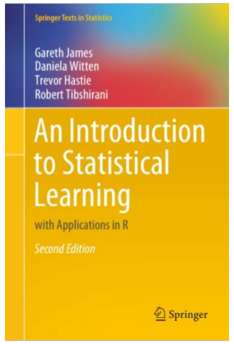
$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k}\\ &= -(y_i - f_\theta(x_i)) \cdot g(z_{ik}). \qquad (10.29)\end{aligned}$$

And now we take the derivative with respect to w_{kj}:

$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}}\\ &= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}. \qquad (10.30)\end{aligned}$$

Notice that both these expressions contain the residual $y_i - f_\theta(x_i)$. In (10.29) we see that a fraction of that residual gets attributed to each of the hidden units according to the value of $g(z_{ik})$. Then in (10.30) we see a similar attribution to input j via hidden unit k. So the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule — a process known as *backpropagation* in the neural network literature. Although these calculations are straightforward, it takes careful bookkeeping to keep track of all the pieces.

backpropagation



Springer Texts in Statistics

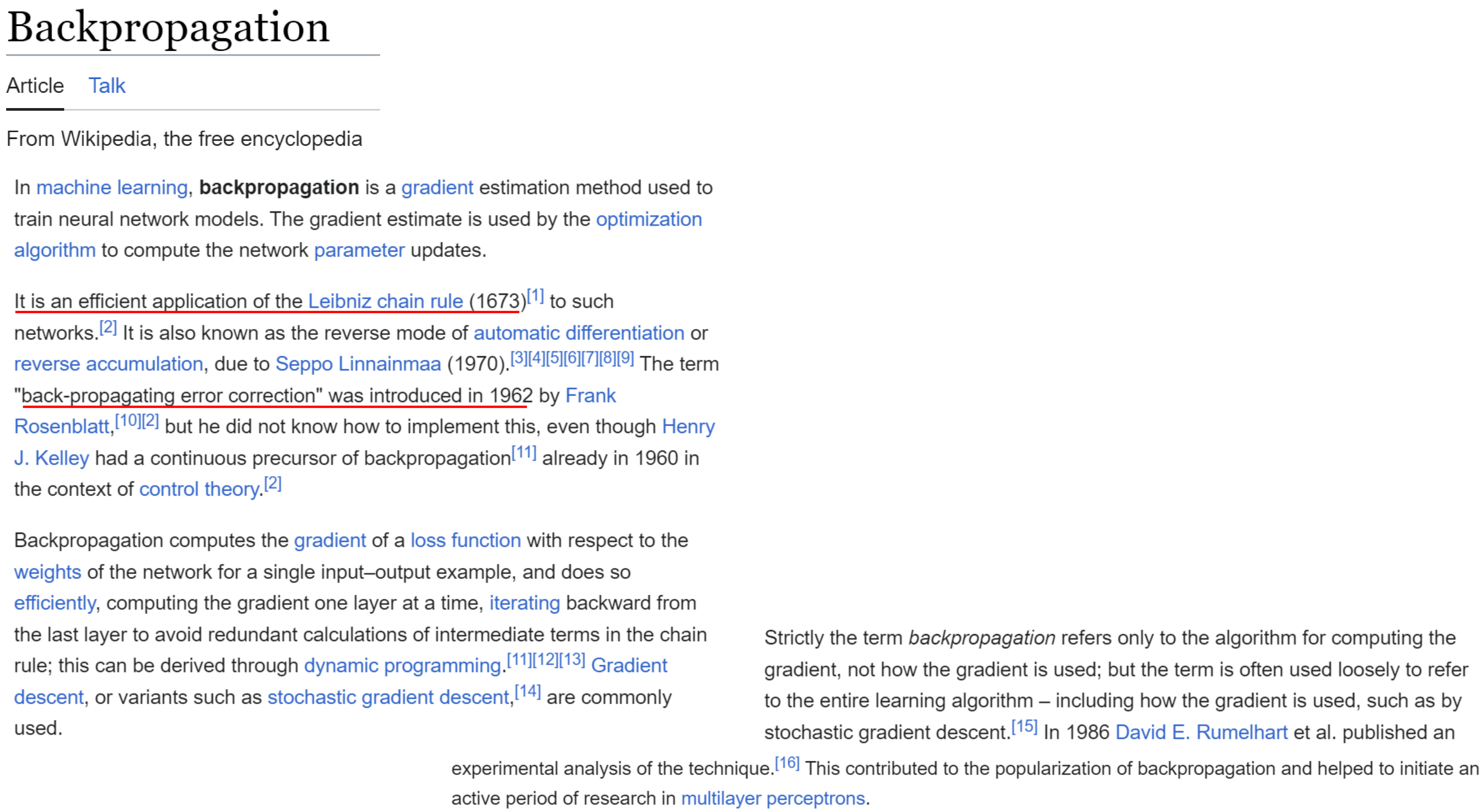
Gareth James
Daniela Witten
Trevor Hastie
Robert Tibshirani

An Introduction to Statistical Learning

with Applications in R

Second Edition

Springer



Backpropagation

From Wikipedia, the free encyclopedia

In machine learning, **backpropagation** is a gradient estimation method used to train neural network models. The gradient estimate is used by the optimization algorithm to compute the network parameter updates.

It is an efficient application of the Leibniz chain rule (1673)[1] to such networks.[2] It is also known as the reverse mode of automatic differentiation or reverse accumulation, due to Seppo Linnainmaa (1970).[3][4][5][6][7][8][9] The term "back-propagating error correction" was introduced in 1962 by Frank Rosenblatt,[10][2] but he did not know how to implement this, even though Henry J. Kelley had a continuous precursor of backpropagation[11] already in 1960 in the context of control theory.[2]

Backpropagation computes the gradient of a loss function with respect to the weights of the network for a single input–output example, and does so efficiently, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this can be derived through dynamic programming.[11][12][13] Gradient descent, or variants such as stochastic gradient descent,[14] are commonly used.

Strictly the term *backpropagation* refers only to the algorithm for computing the gradient, not how the gradient is used; but the term is often used loosely to refer to the entire learning algorithm – including how the gradient is used, such as by stochastic gradient descent.[15] In 1986 David E. Rumelhart et al. published an experimental analysis of the technique.[16] This contributed to the popularization of backpropagation and helped to initiate an active period of research in multilayer perceptrons.

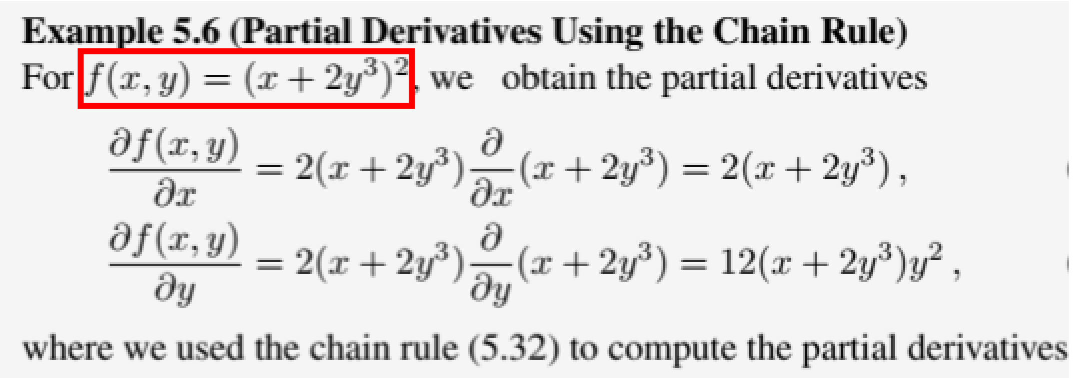
Il calcolo del gradiente di funzioni banali

$$\nabla_{\underline{x}} f = grad\ f = \frac{df}{d\underline{x}} = \left[\frac{\partial f(\underline{x})}{\partial x_1} \quad \frac{\partial f(\underline{x})}{\partial x_2} \quad \dots \quad \frac{\partial f(\underline{x})}{\partial x_n}\right] \in \mathcal{R}^{1xn}$$

Una formula simbolica, quanto facile da calcolare numericamente??

Esempi banali

[da *Mathematics for ML*]

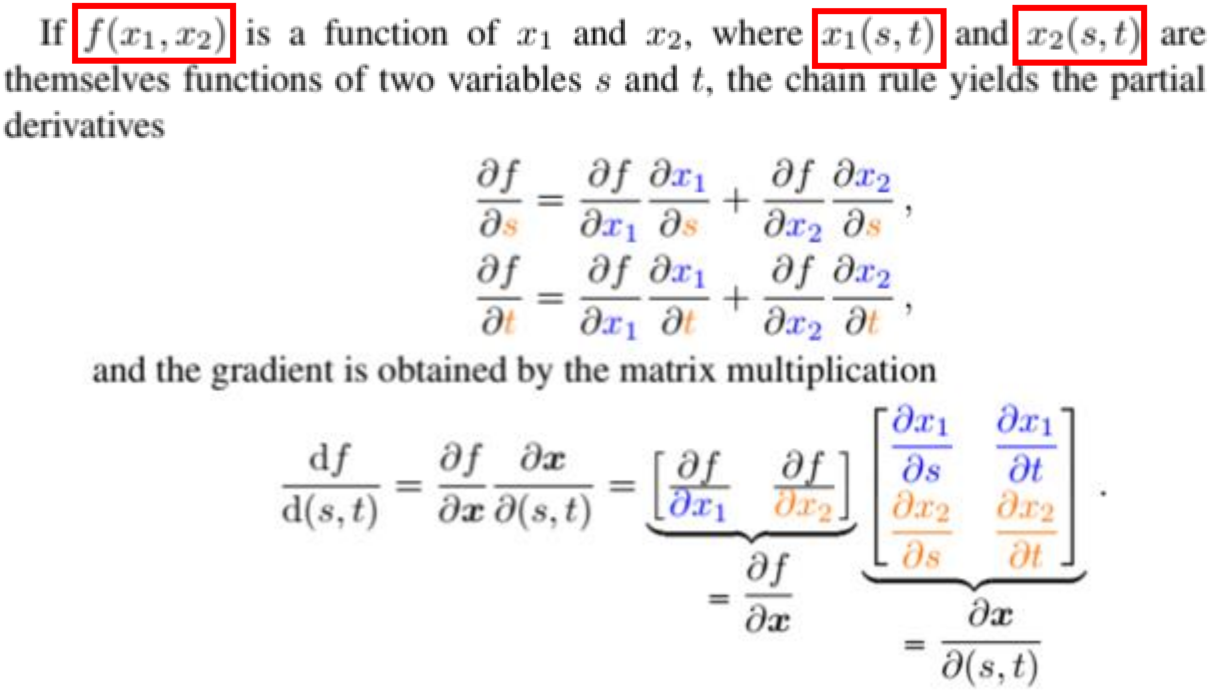


Example 5.6 (Partial Derivatives Using the Chain Rule)
For $\boxed{f(x,y) = (x+2y^3)^2}$, we obtain the partial derivatives

$$\frac{\partial f(x,y)}{\partial x} = 2(x+2y^3)\frac{\partial}{\partial x}(x+2y^3) = 2(x+2y^3)\,,$$

$$\frac{\partial f(x,y)}{\partial y} = 2(x+2y^3)\frac{\partial}{\partial y}(x+2y^3) = 12(x+2y^3)y^2\,,$$

where we used the chain rule (5.32) to compute the partial derivatives



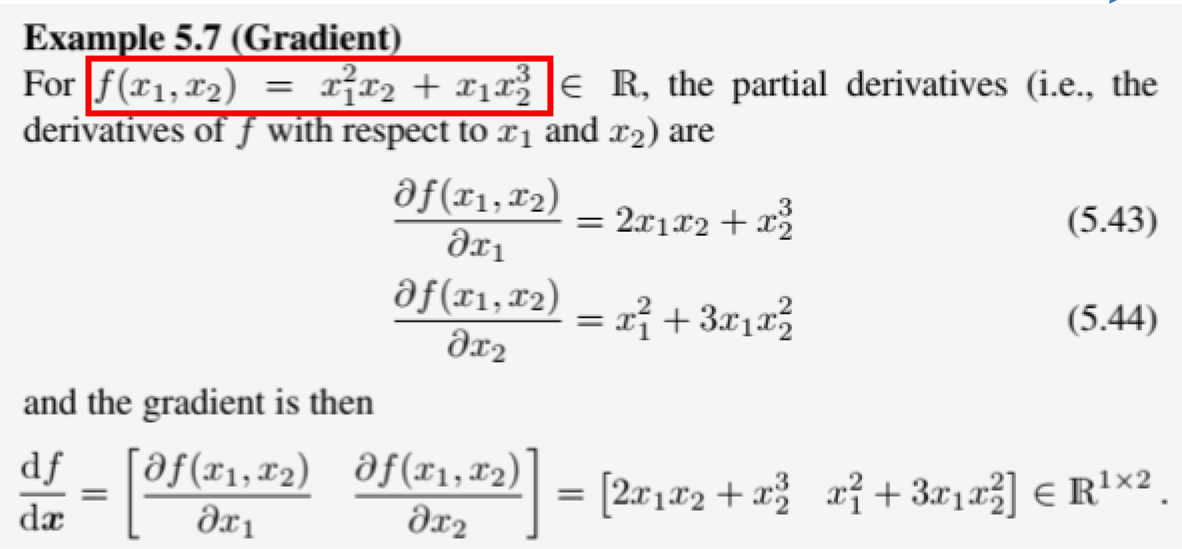
If $\boxed{f(x_1,x_2)}$ is a function of x_1 and x_2, where $\boxed{x_1(s,t)}$ and $\boxed{x_2(s,t)}$ are themselves functions of two variables s and t, the chain rule yields the partial derivatives

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial s} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial s}\,,$$

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial t}\,,$$

and the gradient is obtained by the matrix multiplication

$$\frac{\mathrm{d}f}{\mathrm{d}(s,t)} = \frac{\partial f}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{x}}{\partial(s,t)} = \underbrace{\begin{bmatrix}\frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2}\end{bmatrix}}_{=\frac{\partial f}{\partial \boldsymbol{x}}} \underbrace{\begin{bmatrix}\frac{\partial x_1}{\partial s} & \frac{\partial x_1}{\partial t} \\ \frac{\partial x_2}{\partial s} & \frac{\partial x_2}{\partial t}\end{bmatrix}}_{=\frac{\partial \boldsymbol{x}}{\partial(s,t)}}\,.$$



Example 5.7 (Gradient)
For $\boxed{f(x_1,x_2) = x_1^2x_2 + x_1x_2^3} \in \mathbb{R}$, the partial derivatives (i.e., the derivatives of f with respect to x_1 and x_2) are

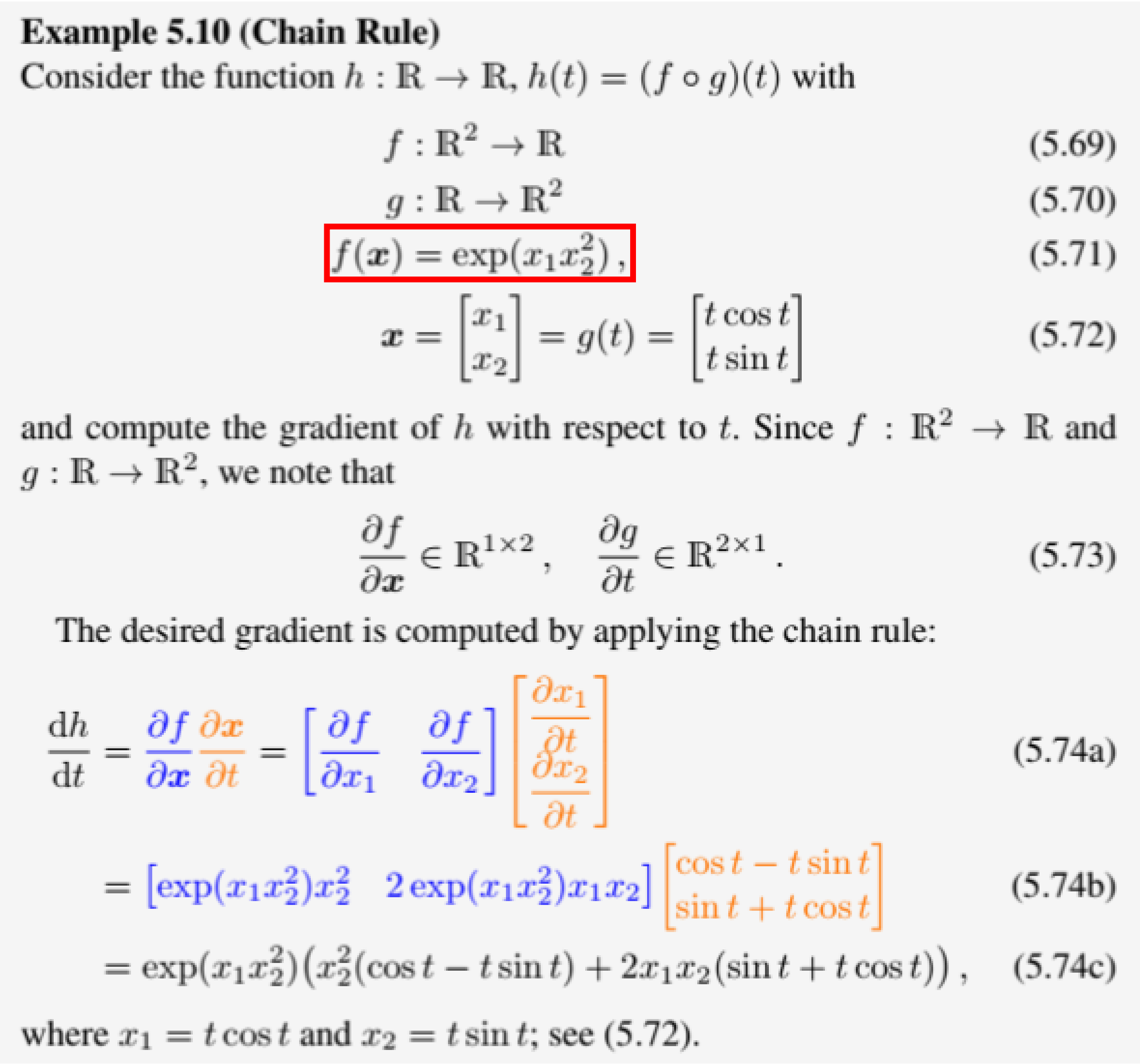
$$\frac{\partial f(x_1,x_2)}{\partial x_1} = 2x_1x_2 + x_2^3 \tag{5.43}$$

$$\frac{\partial f(x_1,x_2)}{\partial x_2} = x_1^2 + 3x_1x_2^2 \tag{5.44}$$

and the gradient is then

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix}\frac{\partial f(x_1,x_2)}{\partial x_1} & \frac{\partial f(x_1,x_2)}{\partial x_2}\end{bmatrix} = \begin{bmatrix}2x_1x_2 + x_2^3 & x_1^2 + 3x_1x_2^2\end{bmatrix} \in \mathbb{R}^{1\times 2}\,.$$

Il calcolo del gradiente di funzioni banali



Example 5.10 (Chain Rule)
Consider the function $h : \mathbb{R} \to \mathbb{R}$, $h(t) = (f \circ g)(t)$ with

$$f : \mathbb{R}^2 \to \mathbb{R} \tag{5.69}$$

$$g : \mathbb{R} \to \mathbb{R}^2 \tag{5.70}$$

$$\boxed{f(\boldsymbol{x}) = \exp(x_1 x_2^2)\,,} \tag{5.71}$$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = g(t) = \begin{bmatrix} t\cos t \\ t \sin t \end{bmatrix} \tag{5.72}$$

and compute the gradient of h with respect to t. Since $f : \mathbb{R}^2 \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}^2$, we note that

$$\frac{\partial f}{\partial \boldsymbol{x}} \in \mathbb{R}^{1\times 2}\,, \quad \frac{\partial g}{\partial t} \in \mathbb{R}^{2\times 1}\,. \tag{5.73}$$

The desired gradient is computed by applying the chain rule:

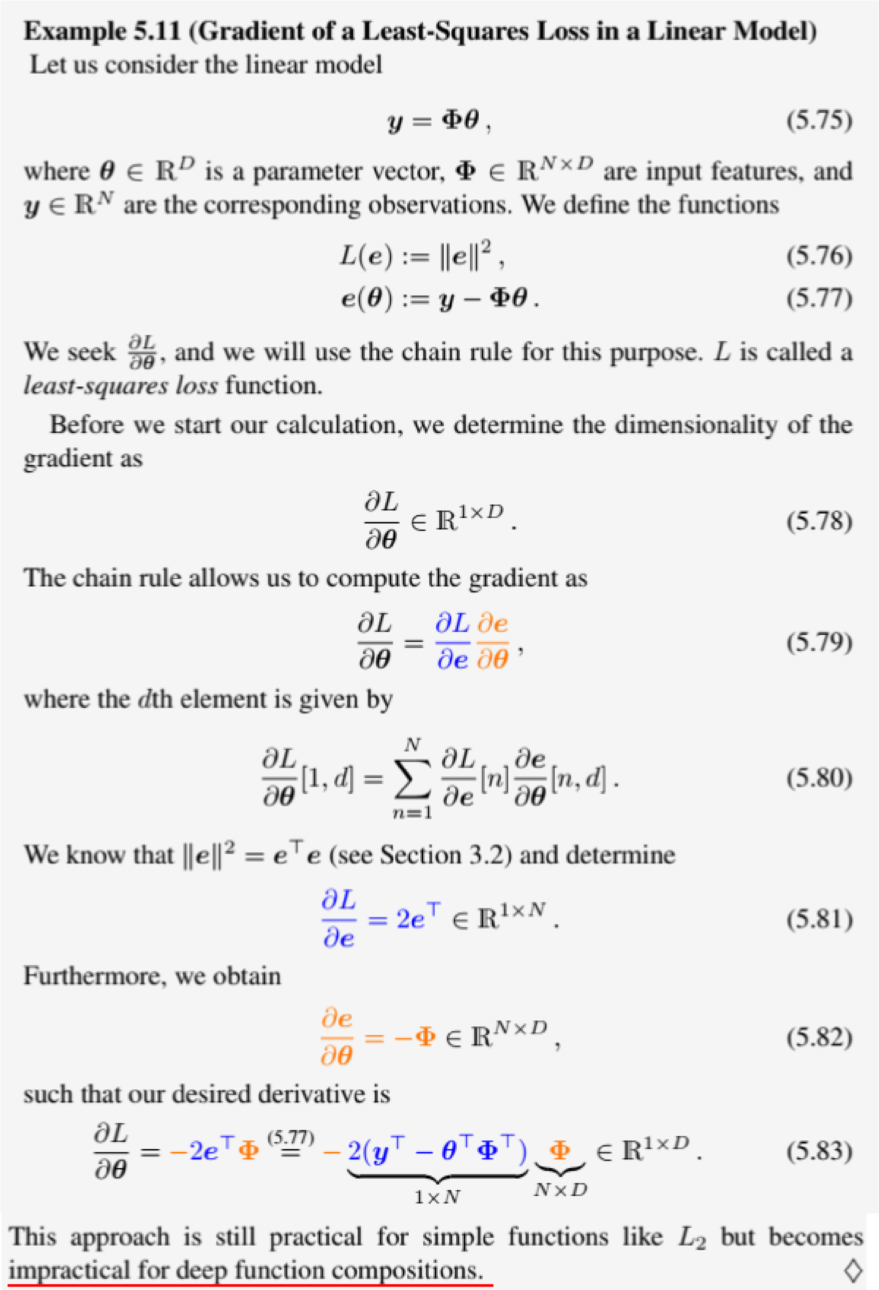
$$\frac{\mathrm{d}h}{\mathrm{d}t} = \frac{\partial f}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial t} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial x_1}{\partial t} \\ \frac{\partial x_2}{\partial t} \end{bmatrix} \tag{5.74a}$$

$$= \begin{bmatrix} \exp(x_1 x_2^2) x_2^2 & 2\exp(x_1 x_2^2) x_1 x_2 \end{bmatrix} \begin{bmatrix} \cos t - t \sin t \\ \sin t + t \cos t \end{bmatrix} \tag{5.74b}$$

$$= \exp(x_1 x_2^2)\big(x_2^2(\cos t - t\sin t) + 2x_1 x_2(\sin t + t\cos t)\big)\,, \tag{5.74c}$$

where $x_1 = t\cos t$ and $x_2 = t \sin t$; see (5.72).

Esempi banali
[da *Mathematics for ML*]



Example 5.11 (Gradient of a Least-Squares Loss in a Linear Model)
Let us consider the linear model

$$\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{\theta}\,, \tag{5.75}$$

where $\boldsymbol{\theta} \in \mathbb{R}^D$ is a parameter vector, $\boldsymbol{\Phi} \in \mathbb{R}^{N\times D}$ are input features, and $\boldsymbol{y} \in \mathbb{R}^N$ are the corresponding observations. We define the functions

$$L(\boldsymbol{e}) := \|\boldsymbol{e}\|^2\,, \tag{5.76}$$

$$\boldsymbol{e}(\boldsymbol{\theta}) := \boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\theta}\,. \tag{5.77}$$

We seek $\frac{\partial L}{\partial \boldsymbol{\theta}}$, and we will use the chain rule for this purpose. L is called a *least-squares loss* function.

Before we start our calculation, we determine the dimensionality of the gradient as

$$\frac{\partial L}{\partial \boldsymbol{\theta}} \in \mathbb{R}^{1\times D}\,. \tag{5.78}$$

The chain rule allows us to compute the gradient as

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \boldsymbol{e}} \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{\theta}}\,, \tag{5.79}$$

where the dth element is given by

$$\frac{\partial L}{\partial \boldsymbol{\theta}}[1, d] = \sum_{n=1}^{N} \frac{\partial L}{\partial \boldsymbol{e}}[n] \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{\theta}}[n, d]\,. \tag{5.80}$$

We know that $\|\boldsymbol{e}\|^2 = \boldsymbol{e}^\top \boldsymbol{e}$ (see Section 3.2) and determine

$$\frac{\partial L}{\partial \boldsymbol{e}} = 2\boldsymbol{e}^\top \in \mathbb{R}^{1\times N}\,. \tag{5.81}$$

Furthermore, we obtain

$$\frac{\partial \boldsymbol{e}}{\partial \boldsymbol{\theta}} = -\boldsymbol{\Phi} \in \mathbb{R}^{N\times D}\,, \tag{5.82}$$

such that our desired derivative is

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = -2\boldsymbol{e}^\top \boldsymbol{\Phi} \overset{(5.77)}{=} -\underbrace{2(\boldsymbol{y}^\top - \boldsymbol{\theta}^\top \boldsymbol{\Phi}^\top)}_{1\times N} \underbrace{\boldsymbol{\Phi}}_{N\times D} \in \mathbb{R}^{1\times D}\,. \tag{5.83}$$

This approach is still practical for simple functions like L_2 but becomes impractical for deep function compositions. ◊

Il calcolo del gradiente nelle Reti Neurali

Negli esempi precedenti (banali) siamo stati in grado di calcolare il gradiente, grazie al calcolo infinitesimale (*calculus*) ed alla *chain rule*, ed esprimerlo in *forma esplicita*, come anche qui sotto (ulteriore esempio)

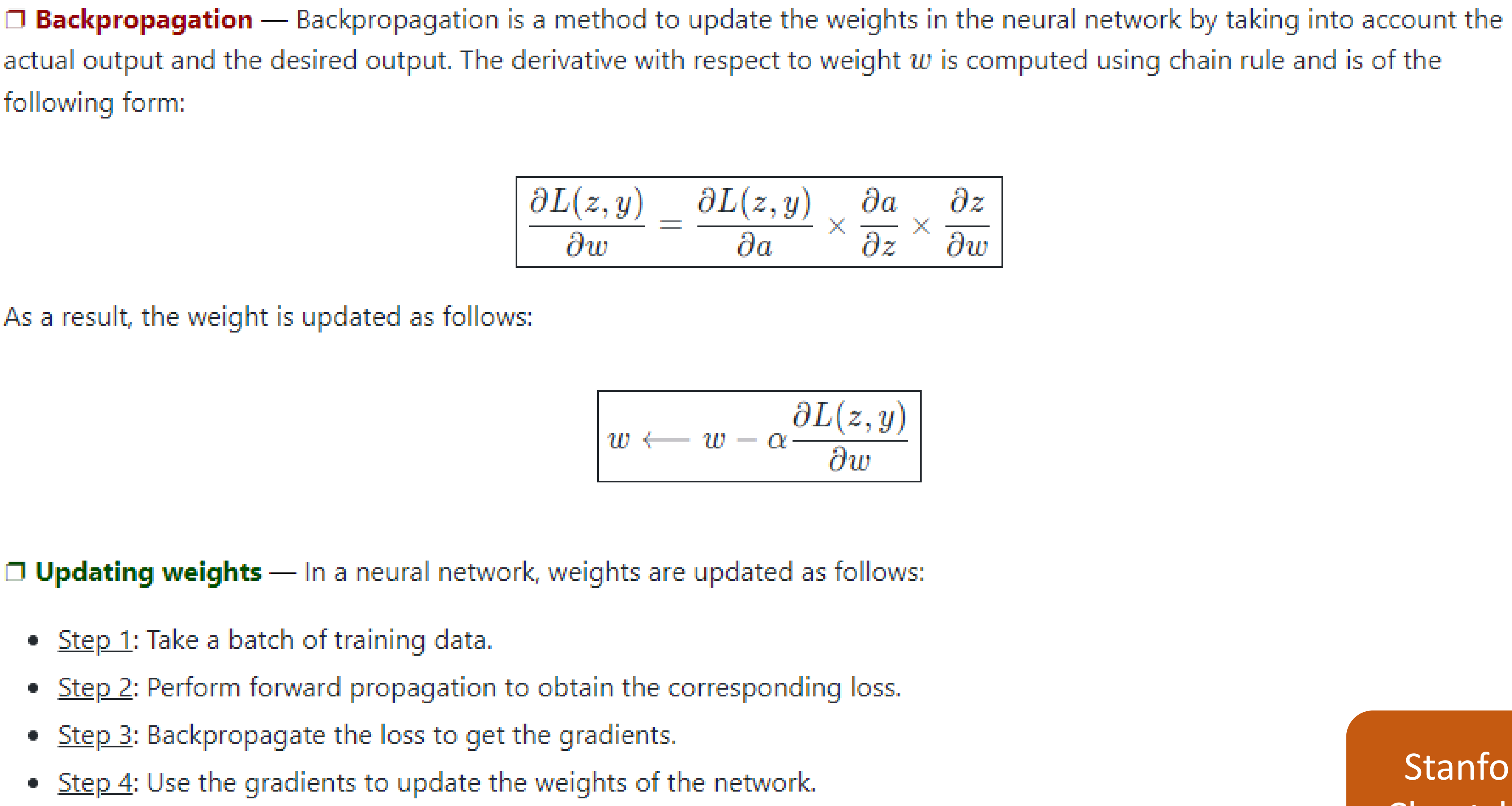
$$f(x) = \sqrt{x^2 + \exp(x^2)} + \cos\left(x^2 + \exp(x^2)\right)$$

⬇ forma esplicita

$$\begin{aligned}\frac{\mathrm{d}f}{\mathrm{d}x} &= \frac{2x + 2x\exp(x^2)}{2\sqrt{x^2 + \exp(x^2)}} - \sin\left(x^2 + \exp(x^2)\right)\left(2x + 2x\exp(x^2)\right) \\ &= 2x\left(\frac{1}{2\sqrt{x^2 + \exp(x^2)}} - \sin\left(x^2 + \exp(x^2)\right)\right)\left(1 + \exp(x^2)\right)\end{aligned}$$

Il gradiente è calcolato nelle reti neurali tramite la back-propagation, un algoritmo che usa la *chain rule* in un modo particolarmente efficiente (back-prop è una tecnica di differenziazione automatica)

La sintesi della back-propagation per Stanford



❐ **Backpropagation** — Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to weight w is computed using chain rule and is of the following form:

$$\boxed{\frac{\partial L(z,y)}{\partial w}=\frac{\partial L(z,y)}{\partial a}\times\frac{\partial a}{\partial z}\times\frac{\partial z}{\partial w}}$$

As a result, the weight is updated as follows:

$$\boxed{w\longleftarrow w-\alpha\frac{\partial L(z,y)}{\partial w}}$$

❐ **Updating weights** — In a neural network, weights are updated as follows:

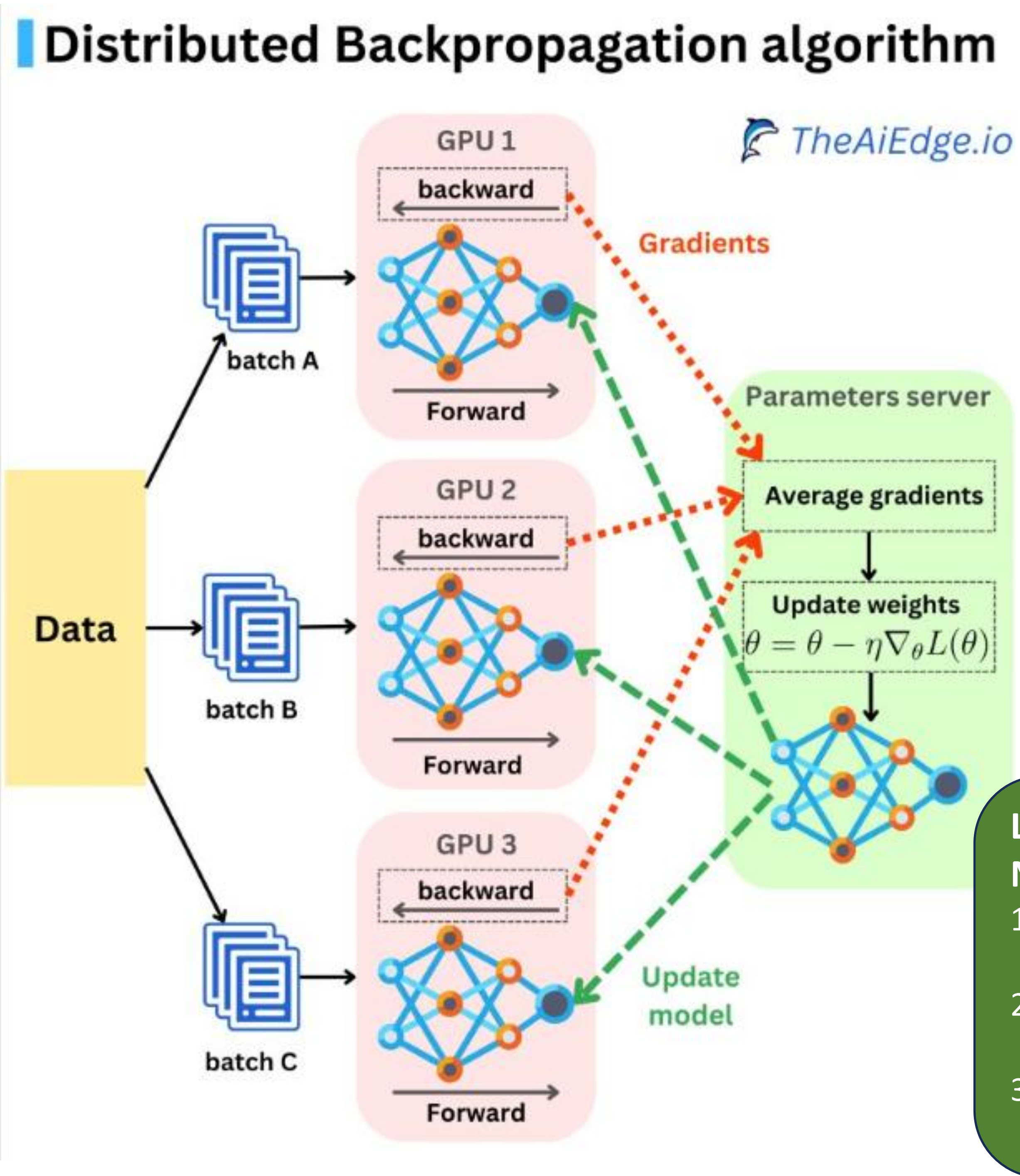
- Step 1: Take a batch of training data.
- Step 2: Perform forward propagation to obtain the corresponding loss.
- Step 3: Backpropagate the loss to get the gradients.
- Step 4: Use the gradients to update the weights of the network.

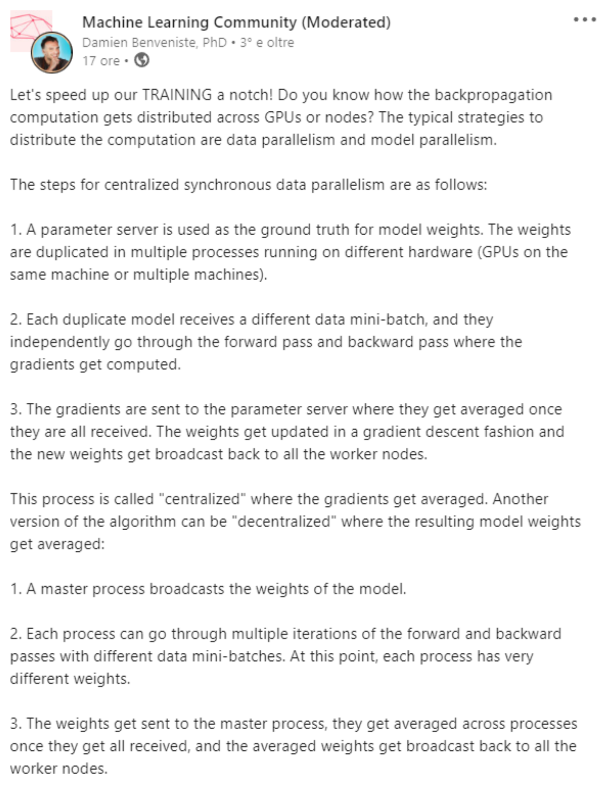
Stanford Deep Learning Cheatsheet (giugno 2023)

Cosa è la *back-propagation* (in letteratura)

- La discesa del gradiente nelle NN? (ESL)
- L'algoritmo di training delle NN usato in combinazione con un metodo di ottimizzazione (ad esempio (S)GD)) – Wikipedia IT
- Un metodo molto efficiente dal punto di vista computazionale per calcolare le derivate parziali di una funzione di costo complessa nelle NN multi-livello (PML)

23 giugno 2023


Distributed Backpropagation algorithm


Machine Learning Community (Moderated)
Damien Benveniste, PhD • 3° e oltre
17 ore •

Let's speed up our TRAINING a notch! Do you know how the backpropagation computation gets distributed across GPUs or nodes? The typical strategies to distribute the computation are data parallelism and model parallelism.

The steps for centralized synchronous data parallelism are as follows:

1. A parameter server is used as the ground truth for model weights. The weights are duplicated in multiple processes running on different hardware (GPUs on the same machine or multiple machines).

2. Each duplicate model receives a different data mini-batch, and they independently go through the forward pass and backward pass where the gradients get computed.

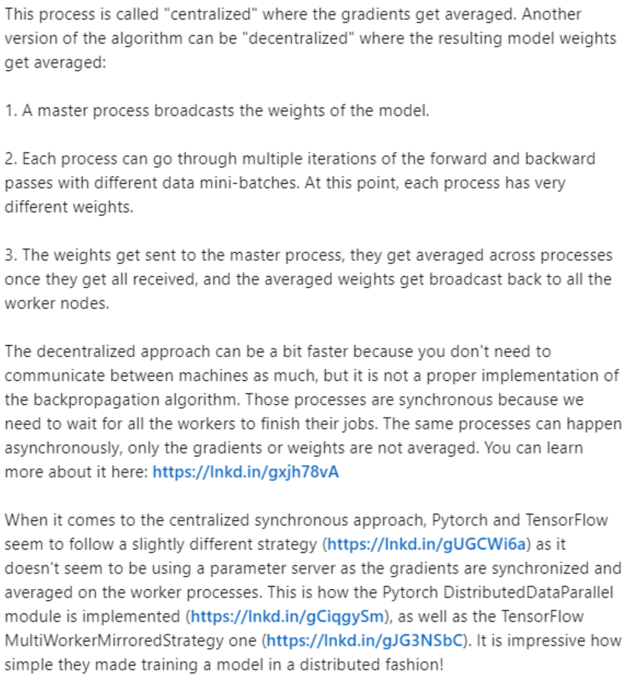
3. The gradients are sent to the parameter server where they get averaged once they are all received. The weights get updated in a gradient descent fashion and the new weights get broadcast back to all the worker nodes.

This process is called "centralized" where the gradients get averaged. Another version of the algorithm can be "decentralized" where the resulting model weights get averaged:

1. A master process broadcasts the weights of the model.

2. Each process can go through multiple iterations of the forward and backward passes with different data mini-batches. At this point, each process has very different weights.

3. The weights get sent to the master process, they get averaged across processes once they get all received, and the averaged weights get broadcast back to all the worker nodes.


This process is called "centralized" where the gradients get averaged. Another version of the algorithm can be "decentralized" where the resulting model weights get averaged:

1. A master process broadcasts the weights of the model.

2. Each process can go through multiple iterations of the forward and backward passes with different data mini-batches. At this point, each process has very different weights.

3. The weights get sent to the master process, they get averaged across processes once they get all received, and the averaged weights get broadcast back to all the worker nodes.

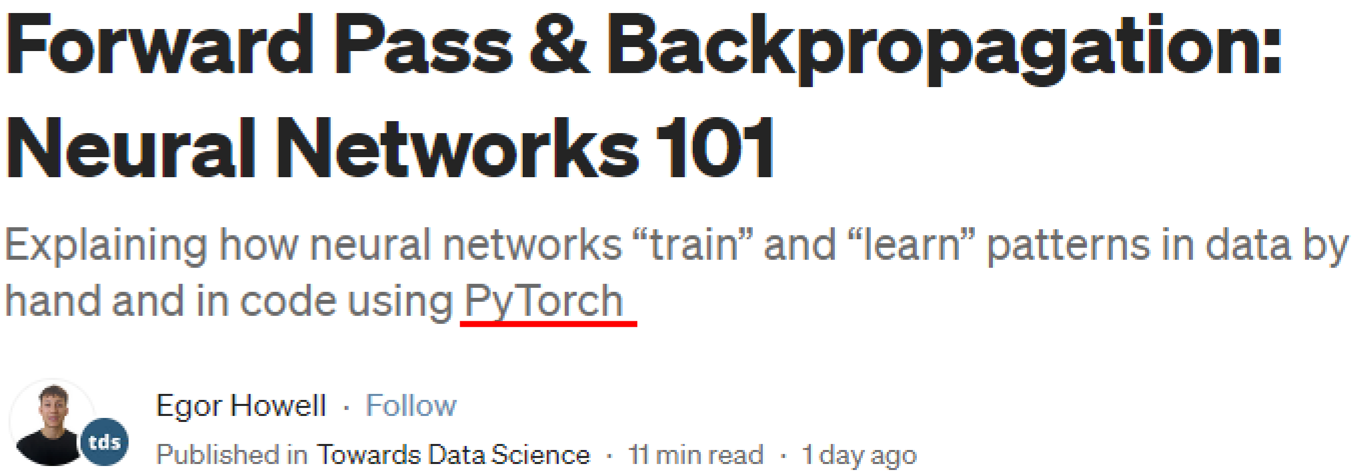
The decentralized approach can be a bit faster because you don't need to communicate between machines as much, but it is not a proper implementation of the backpropagation algorithm. Those processes are synchronous because we need to wait for all the workers to finish their jobs. The same processes can happen asynchronously, only the gradients or weights are not averaged. You can learn more about it here: https://lnkd.in/gxjh78vA

When it comes to the centralized synchronous approach, Pytorch and TensorFlow seem to follow a slightly different strategy (https://lnkd.in/gUGCWi6a) as it doesn't seem to be using a parameter server as the gradients are synchronized and averaged on the worker processes. This is how the Pytorch DistributedDataParallel module is implemented (https://lnkd.in/gCiqgySm), as well as the TensorFlow MultiWorkerMirroredStrategy one (https://lnkd.in/gJG3NSbC). It is impressive how simple they made training a model in a distributed fashion!

La procedura di fitting di una NN MLP ha tre passi:

1. Forward propagation attraverso la rete per calcolare un output
2. Calcolo dei residui (loss function)
3. Error back-propagation per la modifica dei parametri

2 novembre 2023



Forward Pass & Backpropagation: Neural Networks 101

Explaining how neural networks “train” and “learn” patterns in data by hand and in code using PyTorch

Egor Howell · Follow

Published in Towards Data Science · 11 min read · 1 day ago

link