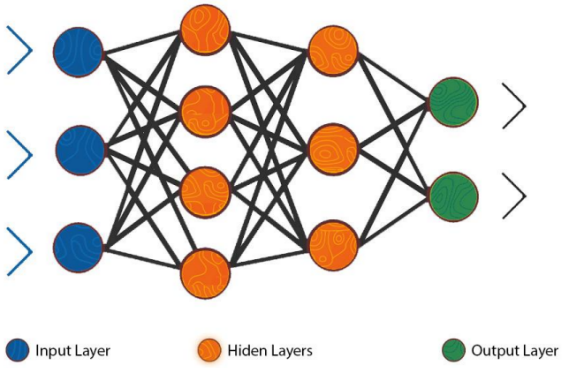
Lezione 2 04/10/2024

Feed Forward Neural Networks

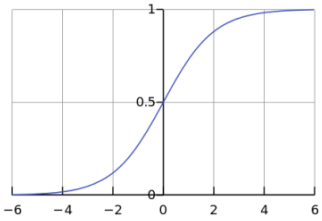
Si chiamano **Feed Forward** perché l'informazione va avanti, l'informazione non fa loop (come nei recurrent neural networks).

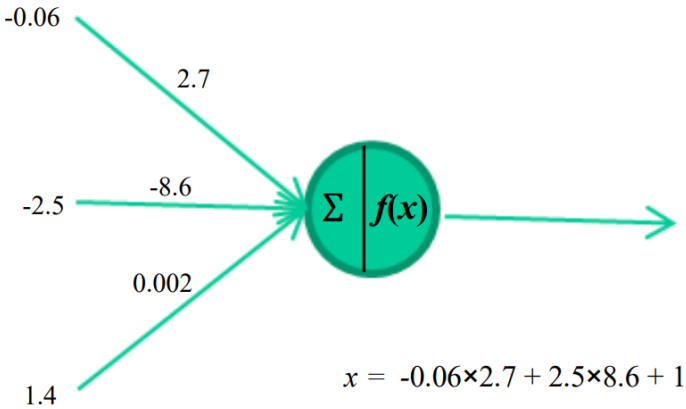


Possiamo pensare a ciascun layer come una funzione vector to vector oppure come un set di unità che operano in parallelo, ciascuna rappresentante un'operazione vector to scalar. Ogni unità (neurone) ha in input un vettore e in output uno scalare, e computa la propria regola di attivazione.

Single neuron example

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





$$x = -0.06 \times 2.7 + 2.5 \times 8.6 + 1.4 \times 0.002 = 21.34$$

$$f(21.34) = 5{,}4 * 10^{-10}$$

FFN hanno una composizione di funzioni tra i diversi layers.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

Come “**profondità**” si usa di solito la lunghezza di questa concatenazione.

Nel training, vogliamo quindi avere un match tra f(x) e la target function f*(x).

I dati di training sono di solito sporchi (noise).

Quindi il comportamento dei layer nascosti non è specificato dai dati di training, ma è l’algoritmo di learning che deve decidere come utilizzarli per approssimare f*.

La dimensione degli hidden layers rappresenta la **larghezza (width)** del modello.

Il training avviene con la back propagation.

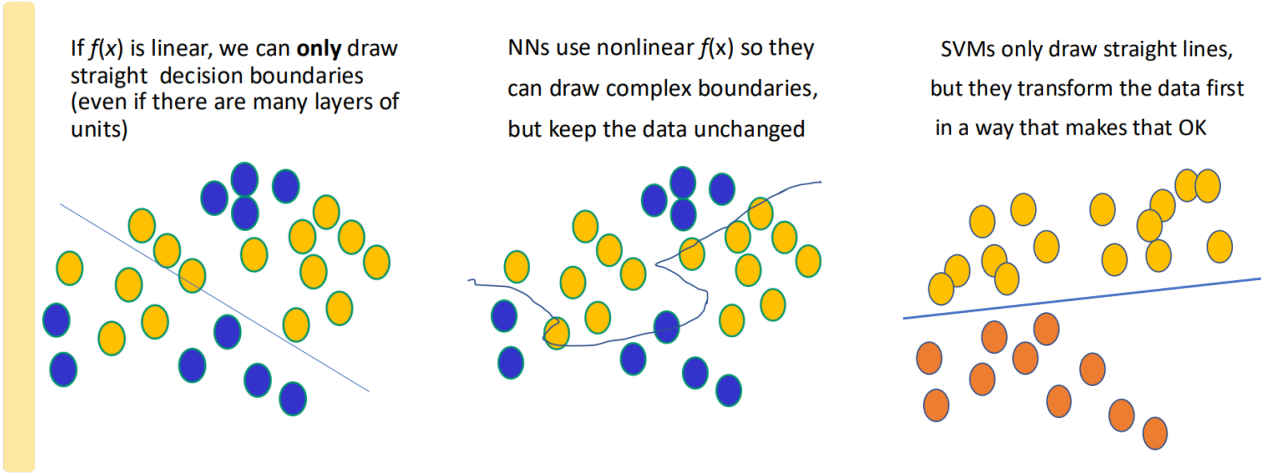
- Inizializziamo la rete con dei pesi random
- Diamo in input un dato e otteniamo un output
- Calcoliamo l’errore rispetto al target
- Aggiorniamo i pesi

L’algoritmo di **training** è basato sul gradiente, è quello che sceglie l’aggiornamento dei pesi.

L’algoritmo di **back propagation** stima il gradiente, è un modo efficiente per stimare il gradiente della loss function. Quindi non è l’algoritmo di learning, è la tecnica che stima il gradiente.

Algoritmi che sistemano i pesi hanno un design tale che l’errore venga ridotto. Questi algoritmi sono stupidi, aggiornano i pesi pian piano, finché si raggiunge un risultato accettabile di classificazione.

Se f(x) è non-lineare, una rete con 1 hidden layer può in teoria imparare perfettamente qualsiasi problema di classificazione. Quindi esiste un set di pesi, il problema è trovarli.



Supervised learning

Le funzioni sono create guardando degli **esempi**, con i loro target, e produrre **generalizzazioni**.

Dato che questo è un problema molto difficile, spesso restringiamo la ricerca tra specifiche famiglie di funzioni chiamate **hypothesis classes**.

Iniziamo con l'ipotesi dei linear models e consideriamo come superare le loro limitazioni.

Per estendere i linear models per rappresentare funzioni non lineari, possiamo applicare il modello lineare ad un input trasformato phi(x) dove phi è una trasformazione non lineare (questa sarà quindi una nuova rappresentazione di x).

> The question is: how to choose the mapping ?
>
> i) Use very generic ϕ (as the one use in the kernel machines)
>
> ii) By manually engineer ϕ
>
> iii) By learning it!!!

Nel deep learning, la strategia è di imparare phi. In questo approccio abbiamo un modello:

$$f(\boldsymbol{x};\ \boldsymbol{\theta}, \boldsymbol{\omega}) = \phi(\boldsymbol{x};\ \boldsymbol{\theta})^{T}\omega$$

Where parameters $\boldsymbol{\theta}$ are used to learn ϕ from a broad class of functions while parameters $\boldsymbol{\omega}$ map from $\phi(\boldsymbol{x})$ to the desired output

In un modello di deep learning, phi definisce un hidden layer.

Linear models

- Binary classification:

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b$$

input vector — weight vector — scalar

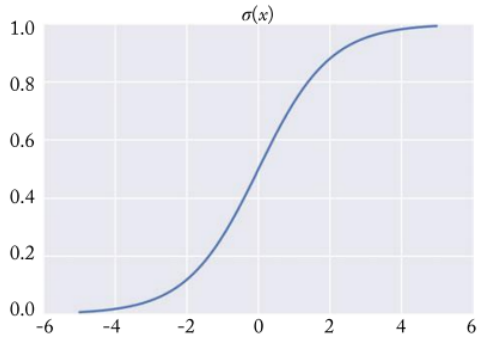
- The range of the linear function is [-∞; +∞]
  - We need to map the output in order to have only values -1 and +1
    - *sign function*

$$\hat{y} = \text{sign}(f(\boldsymbol{x})) = \text{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b)$$

(la freccia nell'immagine è buggata, sarebbe un dot per il "per")

Ma i dataset non sempre lineari, possiamo quindi spostarci ad una dimensione più alta, trasformare il problema...

- Log-Linear Binary Classification: the need of a confidence of the decision
  - Instead of mapping to {-1,+1}, we map to the range [0,1] by using a squashing function such as the *sigmoid function*



  - Our prediction becomes $\hat{y} = (f(\boldsymbol{x})) = \frac{1}{1 + e^{(\boldsymbol{x}\,\boldsymbol{w} + b)}}$

- Multi-class classification:
  - we want to predict an instance as belonging to one of *k* different classes
- A possible solution is to consider *k* weight vectors $w^1, w^2, \ldots w^k$ and biases
  - And predict the class with the highest score:

$$\hat{y} = f(x) = \underset{k \in K}{\operatorname{argmax}} \; \boldsymbol{x} \cdot \boldsymbol{w}^k + b^k$$

  - The sets of parameters $w^k \in R^{in}$ and b^k can be arranged as a matrix $W \in R^{(in \times k)}$ and vector $b \in R^k$, leading to:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$$

$$\text{prediction} = \hat{y} = \underset{k}{\operatorname{argmax}} \, \hat{\boldsymbol{y}}_{[k]}$$

    - where $\hat{y} \in R^k$ is a vector of the scores assigned by the model to each class

- Log-linear multi-class classification: transform the score vector into a probability estimate through the *softmax function*

- Our prediction becomes:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$

$$\hat{\boldsymbol{y}}_{[k]} = \frac{e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[k]}}}{\sum_j e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[j]}}}.$$

  - forcing the values in $\hat{y}$ to be positive and sum to 1, making them interpretable as a probability distribution.

I modelli lineari sono molto utili, e vogliamo usarli perchè sono molto semplici nella computazione del gradiente. Però hanno il problema che sono limitati a funzioni lineari.

Come soluzioni possiamo applicare kernel machines oppure learning della funzione di trasformazione. Vogliamo quindi **linearizzare i dati**.

Però nella maggior parte dei casi, la dimensione del feature space è molto maggiore dello spazio di origine. E dobbiamo stimare la funzione di trasformazione phi.

SVMs approcciano questo problema definendo un set di mappings dove ognuno mappa i dati in uno spazio a dimensionalità molto alta.

- One example is the polynomial mapping $\Phi(\boldsymbol{x}) = (\boldsymbol{x})^a$

$$\Phi(x_1, x_2) = (x_1x_1, x_1x_2, x_2x_1, x_2x_2)$$

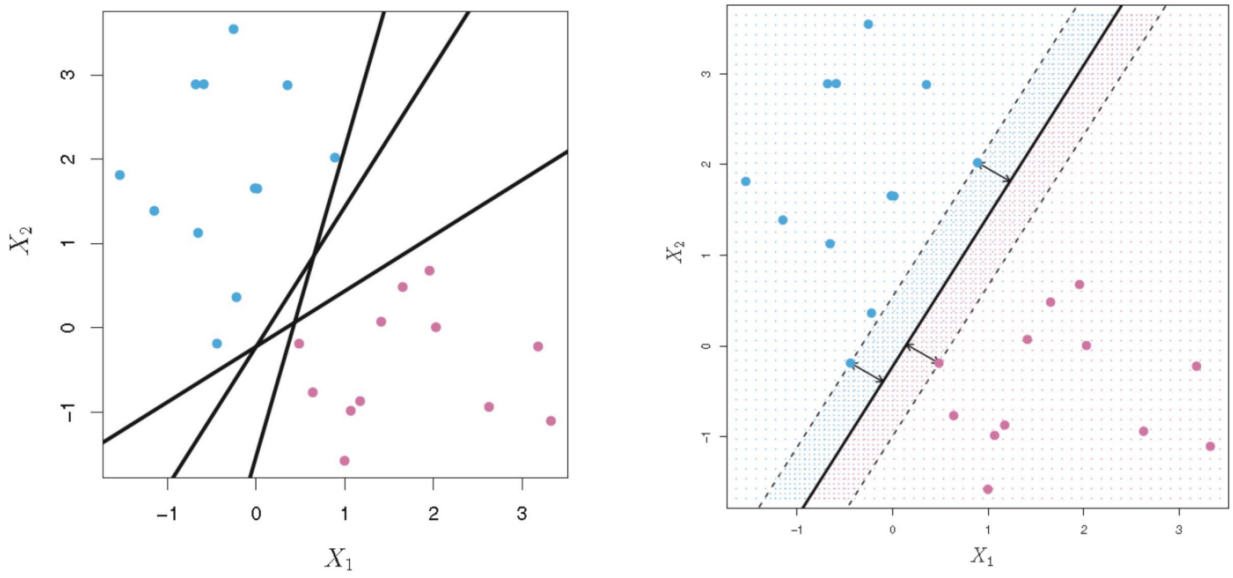
- For d=2, we obtain
  - i.e. all the combinations of the two variables

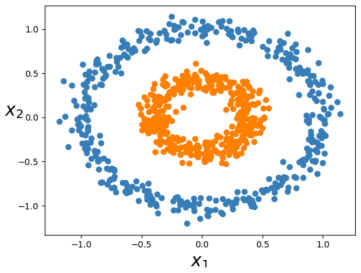
- Although we are now able to train a linear classifier for the XOR problem, we have a polynomial increase in the number of parameters

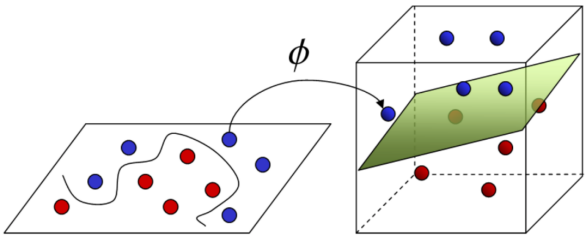
- Let's assume that we have a classification problem with 784 input variables
  - With a simple polynomial mapping we will move from an input space of 784 to a feature space of 784^2=614,656

The kernel trick

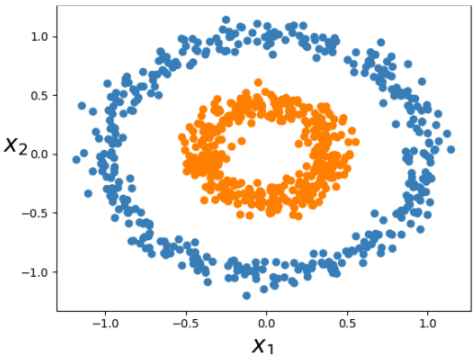
SVM







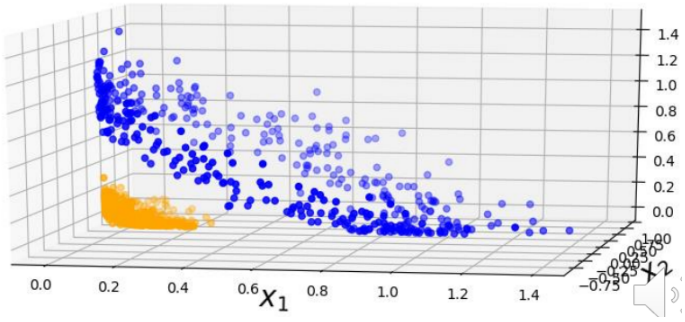
Questo è un esempio di trasformazione, con SVM.



In this case, the kernel is more complex

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

There can be **many transformations** that allow the data to be linearly separated in higher dimensions, but **not all of these functions are actually kernels**.



Se prendo l'input, ed espando la rappresentazione, prima o poi troverò una separazione che separa i dati, però la dimensione aumenta molto, rendendo l'algoritmo non efficiente dal punto di vista computazionale.

Il kernel trick è un metodo che ci lascia operare nel feature space originale, senza computare le coordinate dei dati nello spazio di dimensioni maggiori.

Questo è un metodo più efficiente di trasformazione in dimensioni maggiori (non è limitato solo all'SVM, funziona con qualsiasi computazione che ha dot products).

One key innovation of SVM is associated with the **kernel trick:**

many ML algorithms may be written in terms of dot products beween examples.

For example the linear function used be the SVM can be rewritten as:

$$\boldsymbol{\omega}^{\boldsymbol{T}}\boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{x}^{(i)}$$

Where m is the number of training examples, $x^{(i)}$ is a training example and α a vector of coefficients

In this way we can replace $\boldsymbol{x}$ by the output of a given feature function $\phi(\boldsymbol{x})$

and the dot product with a function $\kappa(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi\left(\boldsymbol{x}^{(i)}\right)$ called **kernel**

And we can make prediction using the function

$$f(\boldsymbol{x}) = \boldsymbol{\omega}^{\boldsymbol{T}}\boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{x}^{(i)} = \mathrm{b} + \sum_{i=1}^{m} \alpha_i \kappa(\boldsymbol{x}, \boldsymbol{x}^{(i)})$$

where the function $f(\boldsymbol{x})$ is nonlinear but the relationship between $\boldsymbol{\alpha}$ and $\phi(\boldsymbol{x})$ is linear

Il kernel trick è potente per due motivi:

- Permette di allenare modelli che non sono lineari come una funzione di x, usando tecniche di ottimizzazione convesse che sono garantite a raggiungere la convergenza efficientemente (questo è possibile perché consideriamo phi() come fissa e ottimizziamo solamente alpha).
- La kernel function k spesso permette un'implementazione che è significativamente più efficiente rispetto alla costruzione di due vettori phi(x) e facendo il dot product esplicitamente.

The most commonly used kernel is the **Gaussian kernel**, also known as **Radial Basis Function (RBF)**

that implies an *infinite dimensional* feature map !

Recap: i kernel sono funzioni che possono computare la distanza tra punti nello spazio di dimensioni più grandi, usando i punti nello spazio di dimensioni originali, quindi senza dover calcolare le due phi che trasformano i dati nello spazio più grande prima di applicare il dot product.

SVM is not the only algorithm that makes use of kernels, the category of algorithms that make use of kernels is called **kernel machines** or **kernel methods**

Since the decision function $f(\boldsymbol{x})$ is linear in the number of training examples, **Kernel machines** soffer of high computational cost of training when the dataset is large

SVM are able to mitigate this problem by learning an $\boldsymbol{\alpha}$ vector that contains mostly zeros, therefore the calssification of a new example requires evaluating the kernel function only for the training examples that have non-zero α_i
These examples are known as **support vectors**.

The modern incarnation of **deep learning** was designed to overcome these limitations of kernel machines !

Gradient-based optimization

L'**ottimizzazione** è la task di minimizzare o massimizzare una funzione f(x) alterando x.

La funzione che vogliamo ottimizzare si chiama **objective function** (**cost function**, **loss function** or error function).

La soluzione è spesso denotata da x*=argmin f(x)

Per le funzioni con input multipli, abbiamo bisogno delle **derivate parziali**, che misurano come la funzione f cambia, al cambiamento di una sola variabile (parametro) x_i . Le derivate parziali ci danno la direzione in cui muoverci per diminuire la loss function.

Il **gradiente** generalizza la nozione di derivata, al caso in cui la derivata si riferisce ad un vettore. Quindi ciascun elemento del vettore è la derivata parziale rispetto a x_i.

$\nabla_x f(x)$ is a vector where the ith element of the gradient is the partial derivative with respect to x_i

La **derivata direzionale** nella direzione u (unit vector) è la pendenza della funzione f nella direzione u e può essere vista come la derivata della funzione $f(x + \alpha * u)$ rispetto ad $\alpha = 0$.

Bisogna quindi muoversi nella direzione del gradiente negativo, questo è il metodo della discesa del gradiente (o steepest descent) (per diminuire f). Quindi partendo da x, troviamo il nuovo x.

$$x' = x - \epsilon \nabla_x f(x)$$

Dove ϵ è il learning rate, ovvero uno scalare positivo che determina la grandezza dello step. Il metodo converge quando ogni elemento del gradiente è zero.

Se abbiamo modelli lineari allora possiamo usare tecniche convesse, mentre per modelli non lineari la procedura di minimizzazione della cost function necessita di un processo iterativo come la discesa del gradiente.

In problemi di dimensionalità alta, ottimizzare (calcolare) lo step size ad ogni step è troppo costoso.

In problemi non convessi, abbiamo tanti minimi locali nella loss function, quindi può essere utile avere un parametro alto all'inizio del training, per esplorare. Quando raggiungo una regione promettente, posso ridurre lo step per trovare il minimo. Questo è un altro motivo per non calcolare questo parametro in modo esatto. Ed è anche per questo che si possono raggiungere risultati diversi in esecuzioni diverse del training.

Il gradiente va computato nell'intero dataset, ma spesso si usano porzioni di dataset quindi non si può calcolare, è anche molto costoso. Di conseguenza, di solito si calcola un'**approssimazione del gradiente**, aggiornandolo usando un subset di parametri. Ad ogni step, l'approssimazione del gradiente migliorerà.

Quasi tutti gli algoritmi di deep learning hanno una combinazione di un dataset, una cost function, una procedura di ottimizzazione e un modello. Possiamo quindi cambiare ciascun componente singolarmente, ottenendo algoritmi diversi.

La cost function può essere difficile da valutare per esempio se abbiamo dataset enormi, ma usiamo dei subset più piccoli. In questi casi possiamo usare una procedura di ottimizzazione numerica iterativa, a patto che possiamo un metodo per approssimare il gradiente.

Training as optimization

L'obiettivo di un algoritmo di training è quello di ritornare una funzione f(x) che mappi correttamente i vettori di input ai label.

Per misurare quanto la funzione sia accurata, usiamo la loss (cost) function che assegna uno score numerico (uno scalare) all'output, conoscendo il valore di output reale (true value).

La loss è computata su ciascuna istanza, e misura la distanza tra il valore predetto e quello reale.

> Given a labelled training set $(x_{1:n}, y_{1:n})$ a per-instance loss function L and a parameterized function $f(\boldsymbol{x}; \boldsymbol{\theta})$ we define the loss as follows:

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

L'obiettivo principale dell'algoritmo di training è quello di trovare i valori dei parametri θ di modo che il valore di $J(\theta)$ è minimizzato.

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

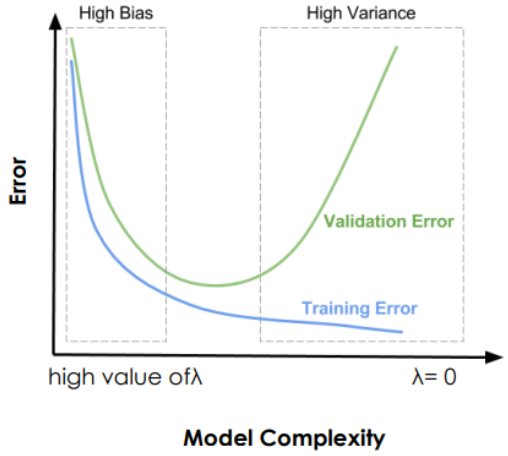
Se minimizzassimo solamente questa funzione, finiremmo per fare overfitting, perchè potremmo non riuscire a generalizzare. Di conseguenza, spesso abbiamo una **funzione di regolarizzazione**. Questa è necessaria soprattutto quando i dati sono limitati. Se il dataset è ampio può non essere necessaria.

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).}_{\text{Loss}} + \underbrace{\lambda R(\boldsymbol{\theta})}_{\text{Regularization}}$$

Dove $R(\theta)$ corrisponde a uno scalare che rispecchia la “complessità” del parametro (che vogliamo mantenere basso/a per questioni di efficienza).

λ è un iperparametro che controlla il trade-off tra bias e varianza.

The optimal value for λ will be probably somewhere around the minimum of the Cross Validation loss function



Un lambda troppo grande causa overfit.