Alma Mater Studiorum – Università di Bologna
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

Tesi di Laurea

---

# Reti Neurali "Physics-Informed" e loro applicazione alle equazioni differenziali

---

**Relatore:**
Prof. Domenico Di Sante

**Candidato:**
Libero Pollini

Anno Accademico 2024/2025

# Indice

# Introduzione

La risoluzione di equazioni differenziali alle derivate parziali (PDEs) è un problema centrale in fisica, poiché esse rappresentano un strumento potente e flessibile per esprimere molte leggi fondamentali della natura. Soluzioni analitiche sono però raramente disponibili, e per decenni sono stati affinati metodi numerici per ottenere approssimazioni sempre migliori. Negli ultimi anni, il *machine learning* e in particolare le *Deep Neural Networks* hanno dimostrato, specie fuori dall'ambito scientifico, una straordinaria capacità di approssimare funzioni complesse. Sulla scia di questo successo, è nato un approccio innovativo: utilizzare una rete neurale per rappresentare direttamente la soluzione di una PDE. Addestrando la rete non solo su dati, ma anche sul rispetto dell'equazione fisica stessa, si ottengono le cosiddette *Physics-informed neural networks* (PINNs).
In questo lavoro di tesi, illustriamo nei capitoli 1 e 2 i concetti principali del *machine learning*, rielaborando e riassumendo (tranne dove esplicitamente indicato) le note e le esercitazioni online di [1]. Nel capitolo 3 spieghiamo come in generale si possano usare le *Physics-informed neural networks* nell'ambito della risoluzione di equazioni differenziali alle derivate parziali (PDEs), basandosi principalmente su [2]. Basandoci sullo stesso articolo, nel capitolo 4 facciamo infine un esempio pratico risolvendo con un'apposita PINN la cosiddetta equazione di Burgers, un caso particolare di quella che in Meccanica dei fluidi è nota come equazione di Navier-Stokes.

# 1. Machine learning

Nell'ambito dell'analisi e gestione di dati nelle scienze statistiche, l'espressione *machine learning* indica in generale una classe di algoritmi che consentono di estrarre informazioni significative da un insieme di dati, senza dover indicare esplicitamente il modo in cui questo va fatto. In questo senso, si fa in modo che un computer (o "macchina") "impari" come estrarre le informazioni volute da un qualsivoglia input di una certa natura, producendo un output desiderato. Per sua natura, quindi, il *machine learning* è una branca della cosiddetta intelligenza artificiale.

## 1.1 Formulazione matematica generale

Da un punto di vista statistico, immaginiamo di avere $m$ vettori $\mathbf{x}_i \in X$ ($i = 1, 2, ..., m$), ciascuno con $n$ caratteristiche, nel cosiddetto spazio delle caratteristiche $X$ (possiamo assumere $X \subseteq \mathbb{R}^n$, cioè ogni caratteristica viene espressa in termini di un numero reale). Assumiamo inoltre che ciascun vettore $\mathbf{x}_i$ sia stato estratto da una distribuzione di probabilità $D(\mathbf{x})$ non nota. Il nostro obiettivo è costruire un "modello" che approssimi la distribuzione $D(\mathbf{x})$ sulla base dei soli dati $\mathbf{x}_i$. Così facendo, saremo in grado di prevedere con quale probabilità un nuovo vettore $\mathbf{x}' \in X$ che ci viene dato sia stato estratto dalla distribuzione $D(\mathbf{x})$; inoltre, potremo anche usare la nostra distribuzione di probabilità approssimata per "estrarre" da essa nuovi vettori simili agli originali.
In alcuni casi, ciascun vettore $\mathbf{x}_i$ viene associato a una e una sola "etichetta" $y_i$ (possiamo assumere $y_i \in L \subseteq \mathbb{R}$; se $L$ è un insieme finito il problema è detto di "classificazione", altrimenti di "regressione"); ci viene chiesto allora di prevedere con quale probabilità a un nuovo vettore $\mathbf{x}' \in X$ sia associata ciascuna delle possibili etichette $y \in L$ (in particolare, ci interesserà in molti casi quale sia l'etichetta "corretta", cioè quella che ha teoricamente probabilità unitaria di essere associata a $\mathbf{x}'$). Possiamo allora procedere in due modi. Il primo consiste nell'approssimare "imparandola" la distribuzione di probabilità condizionata $P(y|\mathbf{x})$, il che va sotto il nome di "allenamento supervisionato". Il secondo consiste invece nell'approssimare invece la distribuzione di probabilità $P(\mathbf{x})$, che lega la distribuzione di probabilità congiunta $P(\mathbf{x}, y)$ a quella condizionata secondo la nota regola di Bayes:

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}, y)}{P(\mathbf{x})} \tag{1.1}$$

e prende il nome di "allenamento non supervisionato" (*unsupervised learning*). Esso comprende anche il caso illustrato sopra, in cui non vengono affatto fornite etichette.

### 1.1.1 Un esempio concreto: *MNIST*

Usiamo un esempio pratico [1] che ci aiuti a capire meglio: immaginiamo che i nostri dati siano delle immagini di cifre (da 0 a 9) scritte a mano, composte da $28 \times 28 = 784$ pixel ciascuno in scala di grigio (con una tonalità che possiamo sempre rappresentare con un numero reale nell'intervallo $[0,1]$, ad esempio; nella realtà computazionale, in genere, essendo necessaria una discretizzazione il numero di valori possibili in questo intervallo sarà finito). Il nostro obiettivo consiste nell'associare a ciascuna immagine, rappresentabile con $\mathbf{x} \in [0,1] \times [0,1] \times \cdots \times [0,1] \subset \mathbb{R}^{784}$ (un vettore con 784 componenti ciascuna fra 0 e 1) il numero da esso rappresentata; avremo perciò 10 etichette, vale a dire i primi dieci interi ($y \in \{0, 1, \cdots, 9\}$). (Alcuni dei campioni sono mostrati in figura 1.1.)
Nel caso dell'allenamento supervisionato, ciascuna immagine ci verrebbe fornita insieme al numero da essa rappresentato (l'etichetta, cioè la "risposta corretta"). Perciò cercheremo semplicemente di predire le probabilità che data un'immagine qualsiasi (un fissato $\mathbf{x}$ generico), essa rappresenti ciascuna delle dieci cifre: $P(y = 0|\mathbf{x})$, $P(y = 1|\mathbf{x})$ e così via.
Nel caso dell'allenamento non supervisionato, invece, trascuriamo le etichette (che potrebbero anche non venirci fornite affatto) e impariamo la probabilità che i pixel siano distribuiti in una qualsiasi maniera $P(\mathbf{x})$ (in altre parole, la probabilità che una certa combinazione di pixel rappresenti o no una cifra scritta a mano). Inoltre, con una qualche strategia automatizzata, cerchiamo di raggruppare le immagini $\mathbf{x}$ con caratteristiche simili, ottenendo nel migliore dei casi 10 raggruppamenti diversi (potremmo imporre questa condizione a priori, se ci venisse fornita), che indichiamo con $C_i$, $i \in I = \{A, B, C, \cdots, J\}$. Impareremo quindi come prima $P(C_i|\mathbf{x})$. Se a questo punto guardiamo le etichette $y$, disponiamo immediatamente della probabilità $P(y|C_i)$ che un'immagine appartenente a $C_i$ abbia etichetta $y$, semplicemente facendo il rapporto fra il numero di immagini del raggruppamento con quella data etichetta e il numero di immagini totali nel raggruppamento. Se ci viene fornita una nuova immagine $\mathbf{x}$, calcoleremo allora, per ogni $y$ fissato, $P(y|\mathbf{x})$ con la regola elementare:

$$P(y|\mathbf{x}) = \sum_{i \in I} P(y|C_i) \cdot P(C_i|\mathbf{x})$$

D'altro lato, possiamo facilmente calcolare anche la probabilità congiunta $P(\mathbf{x}, y)$ con la formula 1.1.
Nel seguito illustriamo come si possa in generale "allenare" un modello ad approssimare la distribuzione voluta.

## 1.2 Supervised learning

In generale, il nostro modello sarà costituito da una funzione che prende in input un vettore $n$-dimensionale nello spazio delle caratteristiche $\mathbf{x}$ e restituisce un vettore (spesso solo un numero reale) $m$-dimensionale $\mathbf{y}$. Dal momento che vorremo modificare questa funzione per far sì che approssimi correttamente la mappatura di ogni vettore di input nel corrispettivo output, consideriamo funzioni parametriche, in cui compaiono cioè molti parametri

[1] Si tratta di un problema ormai classico in *machine learning*, per cui si usa generalmente il database pubblicamente disponibile detto "MNIST" (vedi `http://yann.lecun.com/exdb/mnist`)

**Campioni dal MNIST dataset**


**Figura 1.1:** *Alcuni campioni del dataset MNIST (cifre scritte a mano) con rispettive etichette (in rosso).*

$(\theta_1, \theta_2, ..., \theta_N)$; per brevità indicheremo questo vettore di N componenti con $\boldsymbol{\theta}$. Perciò avremo: $f_\theta : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$.
Vediamo ora come determinare il valore da assegnare ai parametri. Vogliamo naturalmente che la funzione $f_\theta$ approssimi con quanta più precisione possibile la funzione "vera" che mappa un qualsiasi $\mathbf{x} \in X$ in $\mathbf{y}$, $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$. Ovviamente questo non può essere fatto manualmente, tanto più che avremo spesso a che fare con spazi $X$ di grandi dimensioni e con un grande numero (N) di parametri. Perciò introduciamo il concetto di "funzione di costo" (o "di perdita"): una funzione che esprime la "distanza" fra l'etichetta $\mathbf{y}_i$ del vettore $\mathbf{x}_i$ (in altre parole, $\mathbf{y}_i = \mathbf{f}(\mathbf{x}_i)$) e il numero (o vettore) in cui la funzione parametrica mappa $\mathbf{x}_i$ (cioè $\mathbf{f}_\theta(\mathbf{x}_i)$). Esistono molte funzioni di costo, ciascuna adatta a diversi tipi di problemi. Ad esempio, si definisce lo "scarto quadratico medio":

$$\mathrm{MSE}(\boldsymbol{\theta}) = \frac{1}{M} \sum_{i=1}^{M} \|\mathbf{y_i} - \mathbf{f}_\theta(\mathbf{x}_i)\|_\mathbf{2}^2 \tag{1.2}$$

Dove, se $\mathbf{u} \in \mathbb{R}^d$, $\|\mathbf{u}\|_\mathbf{2} \equiv \sqrt{\sum_{j=1}^{d} u_i^2}$ indica la norma euclidea o norma L2, e $M$ il numero di dati. L'obiettivo è minimizzare la "distanza" fra output della funzione parametrica ed etichetta, perciò dovremo minimizzare la funzione di costo, che chiameremo $L(\boldsymbol{\theta}, \mathbf{x}, y)$ e sarà a valori reali, rispetto a ciascuno dei parametri $\theta_i$. Dall'analisi, ovviamente questo equivale a trovare il minimo assoluto $\boldsymbol{\theta}^*$ (se esiste) in cui tutte queste derivate parziali si annullano (e in cui la matrice hessiana è definita positiva):

$$\left(\frac{\partial L}{\partial \theta_j}\right)_{\theta^*} = 0 \quad \forall j = 1, 2, ..., N \tag{1.3}$$

Nei tipici problemi di machine learning, il numero estremamente grande di parametri, unito a espressioni piuttosto complicate della funzione parametrica stessa che in generale

non è "liscia" né convessa, rende computazionalmente molto costosa la ricerca del minimo assoluto. Potremmo anche avere molti minimi locali abbastanza "profondi", per cui c'è solo una piccola differenza nel valore assunto dalla funzione di costo in uno di questi punti rispetto a un altro, tanto che ciascuno di questi punti può essere considerato una buona approssimazione del punto di minimo assoluto. Ricorriamo allora a metodi approssimati. Il più semplice a cui possiamo pensare è il seguente, detto "metodo della discesa del gradiente": scelta una funzione di perdita $L$ come sopra, fissiamo un numero $\eta$ reale positivo sufficientemente piccolo (detto *learning rate)*; inizializziamo tutti i parametri a 0 (o a piccoli numeri casuali, secondo una specifica distribuzione), cioè ci poniamo nel punto $\boldsymbol{\theta}^0 = (0, 0, ..., 0)$; calcoliamo la forma generale del gradiente di $L$ rispetto a $\boldsymbol{\theta}$, cioè $\nabla_\theta L \equiv (\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, ..., \frac{\partial L}{\partial \theta_N})$; calcoliamo il valore del gradiente nel punto $\boldsymbol{\theta}^0$; infine, ci portiamo in un nuovo punto $\boldsymbol{\theta}^1$ assegnando a ciascun parametro il nuovo valore:

$$\theta_j^1 = \theta_j^0 - \eta \cdot \left(\frac{\partial L}{\partial \theta_j}\right)_{\theta_j^0} \tag{1.4}$$

E ripetiamo il procedimento fino alla convergenza desiderata; ad esempio, potremmo decidere di fermarci quando il parametro $\boldsymbol{\theta}$ non varia più in modo significativo, cioè l'iterazione finale $i \geq 1$ sarà quella (la prima) per cui $\|\boldsymbol{\theta}_i - \boldsymbol{\theta}_{i-1}\| \leq \epsilon$, con $\epsilon$ numero reale positivo fissato all'inizio. In alternativa, potremmo fissare all'inizio il numero di iterazioni totali; in questo caso esse vengono dette "epoche".
Una comprensione intuitiva di questa tecnica è la seguente: possiamo pensare alla nostra funzione di perdita come a una (iper)superficie differenziabile in tre (nei problemi reali sono molte di più) dimensioni, dove le variabili indipendenti sono i parametri da ottimizzare; volendo trovarne il minimo assoluto, partiamo da un punto scelto a caso sulla superficie e per ciascuna possibile direzione ci muoviamo "in giù" se la pendenza, data ovviamente dalla derivata parziale prima in quella direzione, è positiva, e viceversa (si veda la figura 1.2). La "lunghezza" del passo è allora il parametro $\eta$, che quindi non può essere né troppo piccolo, altrimenti potremmo rimanere "intrappolati" in minimi locali, né troppo grande, altrimenti rischiamo di "saltare via" il minimo assoluto.
Avvisiamo qui che non daremo una dimostrazione matematica di questo o altri risultati presentati nel seguito. In effetti, per i modelli più complessi di *machine learning* spesso non esistono neppure dimostrazioni del tutto rigorose; piuttosto, si procede per analogie nell'ideare i procedimenti di apprendimento, e per tentativi empirici per mostrarne la loro validità o superiorità rispetto ad altri.
Tornando a noi, come abbiamo visto la funzione di costo oltre che da $\boldsymbol{\theta}$ dipende dalle variabili $\mathbf{x}_i$ (attraverso la funzione parametrica $f_\theta(\mathbf{x})$, la cui espressione viene scelta da noi all'inizio) e $y_i$. Sorge allora il problema di in quale punto $(\mathbf{x}_i, y_i)$ (input-etichetta) dobbiamo calcolarla a ciascuna iterazione. La strada più immediata consiste nel considerare un punto alla volta, e fermare l'iterazione una volta raggiunto l'ultimo ($m$-esimo) punto. Questo però è computazionalmente costoso, dato che così facendo dovremmo "aggiornare" il parametro $\boldsymbol{\theta}$ per ogni punto, ad ogni iterazione. In alternativa, potremmo a ciascuna iterazione calcolare il gradiente della funzione di costo rispetto al parametro $\boldsymbol{\theta}$ come una media dei gradienti calcolati in ciascuno di (tutti) i punti $(\mathbf{x}_i, y_i)$ con $\boldsymbol{\theta}$ fissato, aggiornando $\boldsymbol{\theta}$ un'unica volta per iterazione. Ciò che si fa solitamente è una via di mezzo fra le due: si

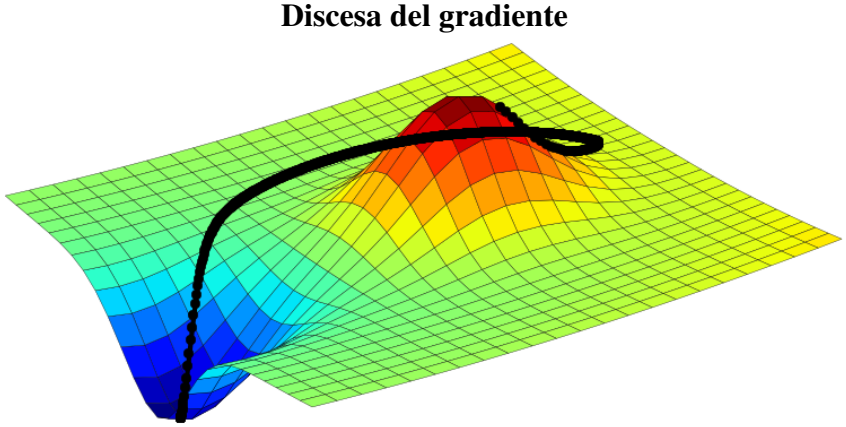


**Figura 1.2:** *Una visualizzazione della discesa del gradiente in tre dimensioni. La quota (rappresentata anche con una scala di colori) corrisponde al valore della funzione di perdita, che in questo caso dipenderebbe semplicemente da due parametri ottimizzabili; in nero è rappresentato il percorso discreto che, seguendo il verso del gradiente, porta al minimo locale.*

suddivide il campione di dati $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ in un certo numero di sottinsiemi di un fissato numero di punti (scelti a caso), detti *mini batches*, e si calcola la media del gradiente su ciascun *mini batch*. Dopo un certo numero di iterazioni, si considera una nuova suddivisione casuale in *mini batches*. Il numero di punti in ogni *minibatch* è scelto all'inizio e viene detto *minibatch size*.
Una nota importante è che parametri come $\eta$, $N_{epoche}$ e $size_{minibatch}$ vengono fissati a priori, cioè in fase di progettazione del modello e in modo non automatizzato, e perciò si dicono "iperparametri".
Alla fine di questo processo iterativo, detto di *training*, la nostra funzione $f_\theta$ mapperà dunque ciascun vettore di input $\mathbf{x}_i$ in un numero reale $f(\mathbf{x}_i)$ "sufficientemente vicino" a $y_i$.
Se tutto va come vogliamo, inoltre, anche nuovi punti in $X$, ad esempio di un insieme $\{(\mathbf{x}_{m+1}, y_{m+1}), (\mathbf{x}_{m+2}, y_{m+2}), ..., (\mathbf{x}_{m+q}, y_{m+q})\}$, saranno mappati "non troppo distante" dalle rispettive etichette. Per verificare che sia così, finito il *training* (quindi con $\boldsymbol{\theta}$ fissato) calcoliamo il cosiddetto "errore di generalizzazione", nient'altro che la media del valore della funzione di perdita in ciascuno dei nuovi punti (che in generale selezioneremo all'inizio fra quelli che ci sono forniti, suddividendo così il nostro insieme di punti in due sottinsiemi distinti), verificando che non sia troppo grande (rispetto al valore medio della funzione di perdita calcolata nei punti di *training*). Questo processo è detto di *testing*. In modo forse poco intuitivo, esso è necessario per evitare di "imparare" un modello che mappi quasi perfettamente i vettori su cui è stato allenato (valore della funzione di perdita quasi nullo), ma con poca accuratezza i nuovi punti. Questo comune problema, detto *overfitting*, può avvenire nei casi di modelli molto complessi (con un elevato numero di parametri relativamente alla funzione da approssimare), che "imparano" caratteristiche non essenziali del sottinsieme di *training* perdendo in generalità.

# 2. Deep neural networks

Un particolare tipo di modello usato in *machine learning* per approssimare distribuzioni di probabilità, con allenamento supervisionato o non supervisionato, sono le "reti neurali" (*neural networks*).

## 2.1 Struttura di una rete neurale

Questi modelli hanno riscosso grande successo negli ultimi anni anche (e soprattutto) fuori dall'ambito puramente scientifico. La loro architettura è stata storicamente progettata per assomigliare, con grossa approssimazione, a quella del cervello umano: si hanno dei "neuroni" connessi fra loro, che quando ricevono un input abbastanza intenso si "accendono", propagando il segnale. A livello matematico, comunque, il modello consiste pur sempre in una funzione parametrica che mappa un input $n$-dimensionale nel corrispettivo output. L'architettura più comune prevede vari "strati" di neuroni computazionali: neuroni di uno stesso strato non sono connessi fra loro, ma ciascuno è connesso unicamente a tutti (nel caso di strati cosiddetti "densi") i neuroni dello strato precedente e a tutti quelli del successivo; la propagazione del segnale è unidirezionale, dallo strato di input a quello di output. Se sono presenti strati intermedi parliamo di "rete neurale profonda" (*deep neural network*). Si tratta di un'architettura molto diffusa per la grande efficacia pratica; intuitivamente, consente di "imparare" rappresentazioni gerarchiche dei dati via via più complesse andando da uno strato al successivo.
Esaminiamo ora questa struttura nel dettaglio. Ogni singolo neurone è descritto da una funzione che mappa un input $k$-dimensionale, dove $k$ è il numero di neuroni dello strato precedente, in un output unidimensionale: $f : \mathbb{R}^k \to \mathbb{R}$. Questa funzione è la composizione di una "funzione di attivazione" $g$, non lineare e la stessa per tutti i neuroni di uno strato, con una funzione puramente lineare nell'input, $q$, a valori reali:

$$\begin{aligned} f(\mathbf{z}) &= g(q(\mathbf{z})), \\ \mathbf{z} \equiv (z_1, z_2, \ldots, z_k), \quad q(\mathbf{z}) &= \sum_{j=1}^{k} w_j z_j + b, \quad g : \mathbb{R} \to \mathbb{R} \end{aligned} \tag{2.1}$$

Dove $w_j$ sono detti "pesi" e $b$ "*bias*", e tutti sono numeri reali. In altre parole, il valore assunto da ciascun neurone è solo una media pesata dei valori assunti da tutti i neuroni dello strato precedente, "aggiustato" in seguito, come vedremo, in modo non lineare. Se consideriamo un intero strato di $l$ neuroni che riceve un input $k$-dimensionale, in altre parole un vettore colonna $k \times 1$, possiamo vedere anche il *bias* e l'output lineare $\mathbf{z}' \equiv \mathbf{q}(\mathbf{z})$ dello strato in questione come vettori $l \times 1$. In questa notazione cioè $\mathbf{q}(\mathbf{z}) \equiv (q_1(\mathbf{z}), q_2(\mathbf{z}), \ldots, q_l(\mathbf{z}))$. Perciò, possiamo disporre i pesi in una matrice $W$ di dimensioni $l \times k$, prendere la funzione $g$ "elemento per elemento", cioè $\mathbf{g}(\mathbf{z}') \equiv (g(z'_1), g(z'_2), \ldots, g(z'_k))$, e scrivere:

$$\mathbf{o} = \mathbf{g}(W\mathbf{i} + \mathbf{b}) \tag{2.2}$$

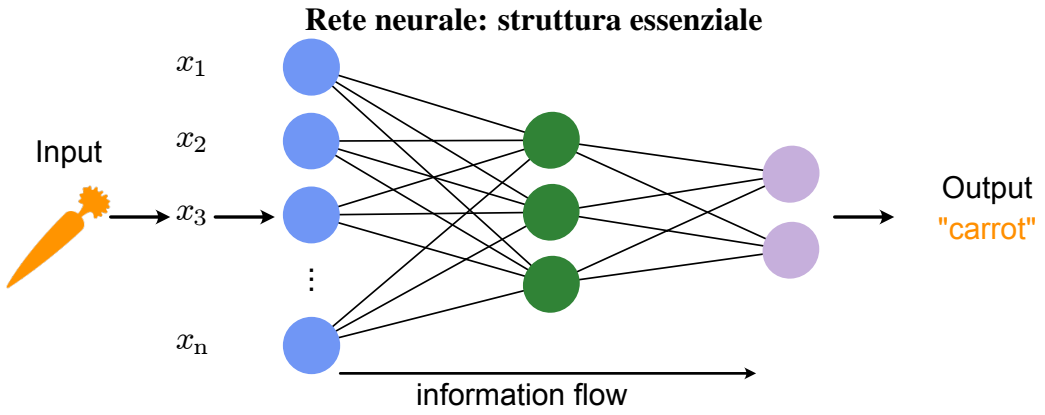


**Figura 2.1:** *Una rappresentazione schematica e semplificata dell'architettura di una rete neurale (da [1]). I punti rappresentano i "neuroni" (possiamo immaginare la funzione di attivazione come immediatamente precedente il singolo neurone) e le linee che li connettono i pesi. In questo caso è presente un solo strato intermedio, denso e senza bias, e l'output è bidimensionale.*

dove **o** e **i** sono rispettivamente l'output (di dimensioni $l \times 1$) e l'input (di dimensioni $k \times 1$) dello strato considerato. In questo formalismo, l'elemento di matrice $(W)_{ij}$ rappresenta allora la "forza" della connessione fra il neurone $i$ dello strato considerato e il neurone $j$ dello strato precedente, con indici $i = 1, 2, \ldots, l$ e $j = 1, 2, \ldots, k$. Ricordando che in questa architettura l'output di uno strato coincide con l'input dello strato successivo, indicando ad apice fra parentesi quadre il numero dello strato a cui ci riferiamo avremo:

$$f^{[n]} = \mathbf{g}^{[n]}(W^{[n]}\mathbf{f}^{[n-1]} + \mathbf{b}^{[n]}) \tag{2.3}$$

perciò una funzione complessiva che è una lunga (o "profonda") composizione di funzioni, ad esempio per 3 strati totali ($n = 1, 2$ poiché l'input $\mathbf{x}$ coincide con lo strato di input):

$$\mathbf{F}(\mathbf{x}) = \mathbf{g}^{[2]}[W^{[2]}\mathbf{g}^{[1]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]})]$$

In questa architettura, i parametri che si vanno poi ad ottimizzare con i metodi già introdotti sono i pesi e i *bias* di tutti gli $N$ strati, che indichiamo per brevità con $\boldsymbol{\theta}$,

$$\boldsymbol{\theta} \equiv (w_{11}^{[1]}, \ldots, w_{l_1 k_1}^{[1]}, b_1^{[1]}, \ldots, b_{l_1}^{[1]}, \ldots, \ldots, w_{11}^{[N]}, \ldots, w_{l_N k_N}^{[N]}, b_1^{[N]}, \ldots b_{l_N}^{[N]})$$

In figura 2.1 è mostrata una schematizzazione di questa architettura. Nel seguito illustriamo più nel dettaglio alcune caratteristiche delle reti neurali profonde, concentrandoci in particolare su quelle rilevanti per i nostri scopi di ricerca.

### 2.1.1 Funzione di attivazione

Abbiamo detto che il valore assunto da ciascun neurone non è semplicemente una media pesata (lineare) dei valori di tutti i neuroni dello strato precedente, ma una sua mappatura non lineare. Il perché può essere visto in due modi: matematicamente, vogliamo che una

rete neurale sia in grado di approssimare funzioni anche fortemente non lineari, quindi perlomeno la sua struttura dev'essere di questo tipo; storicamente, volendo in un certo senso mimare il comportamento dei neuroni biologici, sappiamo che essi si "accendono" (propagano un segnale elettrochimico) solo se il segnale che ricevono supera una certa soglia. Richiederemo perciò che la funzione di attivazione $g(z')$ sia monotona crescente (in senso lato), con estremo inferiore finito. Inoltre che sia differenziabile rispetto a $z'$, dato che dovremo calcolare le derivate parziali della funzione complessiva $\mathbf{F}(\mathbf{x})$, che sarà una composizione anche di queste funzioni di attivazione rispetto ai parametri in fase di ottimizzazione.
Esempi molto usati sono la funzione "ReLu" (che sta per *rectified linear unit*), definita con $\mathrm{ReLu}(z') \equiv \max\{0, z'\}$ (che in effetti non è derivabile in 0, ma questo nella pratica non è un problema: basta assegnare convenzionalmente un valore, ad esempio 0 o 1, alla derivata nel punto $z' = 0$); la funzione "sigmoide" $\sigma(z') \equiv \frac{e^{z'}}{e^{z'}+1}$, che ha estremi 0 e +1 e "sale" piuttosto velocemente in un intorno di $z' = 0$; e la funzione tangente iperbolica, quella di nostro interesse, molto simile ma con estremo inferiore -1:

$$\tanh(z') \equiv \frac{e^{2z'} - 1}{e^{2z'} + 1} \tag{2.4}$$

mostrata nel grafico 2.2. Ricordiamo infine che, a meno di casi particolari, la funzione di attivazione è la stessa per tutti i neuroni di uno strato, in altre parole $g_1(\mathbf{z}') = g_2(\mathbf{z}') = \ldots = g_l(\mathbf{z}') =: g(\mathbf{z}')$.

**Funzione tangente iperbolica**

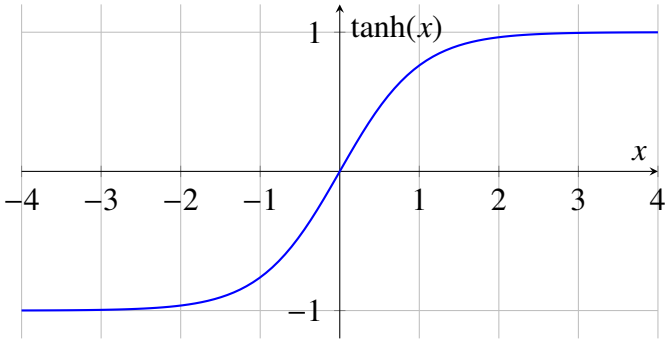


**Figura 2.2:** *Grafico della funzione di attivazione tangente iperbolica* $\tanh(x)$*, utilizzata nelle reti neurali per la sua proprietà di essere limitata e analitica.*

### 2.1.2 Input layer

Qui e nelle due sezioni successive illustriamo alcuni tipi particolari di "strati" (*layers*) di una rete neurale, che utilizzeremo in seguito per i nostri scopi. I dettagli implementativi si riferiscono alla libreria `Keras` (vedi [3]) di *Python*.
Iniziamo dall'*input layer*. Esso consiste semplicemente di $k_0$ neuroni (un parametro da specificare), tanti quanti la dimensione del vettore di input $\mathbf{x} \in \mathbb{R}^{k_0}$, a ciascuno dei quali viene assegnato il valore (in generale reale) di una componente di $\mathbf{x}$. Per sua stessa natura, quindi, questo strato non ha parametri da ottimizzare.

### 2.1.3 Scaling layer

Immediatamente dopo lo strato di input, possiamo porre uno strato di "riscalaggio" (*scaling layer*). Nella nostra implementazione usiamo uno strato "Lambda", in cui con un'appropriata *lambda function* riscaliamo individualmente e linearmente ciascun elemento $x_j$ del vettore di input in uno stesso intervallo reale limitato scelto a priori $[a,b]$. Questo metodo di riscalaggio delle caratteristiche ("*feature scaling*"), uno dei più semplici e più comuni, viene anche detto "min-max" e la nuova componente $\tilde{x}_j$ è data da:

$$\tilde{x}_j = a + \frac{(x_j - \min_j(x_j))(b-a)}{\max_j(x_j) - \min_j(x_j)}$$

Esistono comunque molti altri tipi di *feature scaling*, come ad esempio quello di "centramento sulla media" (per avere dati distribuiti con media nulla) o di "*robust scaling*" (in cui i dati già centrati vengono divisi per lo scarto interquartile del campione, in modo da ridurre l'impatto degli *outliers*). Gli obiettivi di questa procedura sono vari: dando un ugual "peso" a ciascuna componente, si vuole evitare che il modello si convinca che una è più importante delle altre solo in virtù della sua diversa unità di misura; prevenire il rischio di instabilità computazionale dovuto a grandezze troppo piccole o troppo grandi; aumentare la velocità di convergenza in fase di ottimizzazione; insomma, migliorare la performance dell'algoritmo e abbattere i costi computazionali. Va notato comunque che molti di questi risultati sono empirici, e che la procedura di *feature scaling* potrebbe essere ininfluente in certi casi o fortemente influenzato dalla modalità scelta in altri. Una panoramica su vari tipi di *feature scaling* e relativa efficacia in contesti diversi si può trovare in [4].
Osserviamo che neppure questo strato presenta parametri da ottimizzare.

### 2.1.4 Dense layer

Il terzo e ultimo tipo di strato che illustriamo è quello "denso" (*dense*), corrispondente a quello già ampiamente descritto all'inizio di questa sezione 2.1.
Ricordiamo solo brevemente che in questo tipo di architettura ogni neurone computazionale è connesso unicamente a tutti quelli dello strato precedente e del successivo. I parametri da specificare per ciascuno strato sono allora il numero di neuroni di cui è costituito, la funzione di attivazione e il *kernel initializer*, ovvero la maniera in cui i parametri da ottimizzare vadano inizializzati all'inizio del *training*. Ricordiamo infine che, per uno strato di $l$ neuroni preceduto da uno di $k$ neuroni, questi parametri sono tutti i pesi, che raccogliamo in una matrice $W \in M_{l\times k}(\mathbb{R})$, e il *bias*, un vettore $l \times 1$; perciò per ciascuno strato "denso" avremo in totale $(k+1)l$ parametri allenabili.
Osserviamo infine che anche lo strato di output potrebbe essere semplicemente uno strato denso, con tanti neuroni quanti la dimensione desiderata dell'output; in questo caso, l'output del modello sarà un vettore i cui elementi sono uguali al valore di output di ciascuno dei neuroni dell'ultimo strato. Sebbene esistano architetture più sofisticate, come lo strato "softmax" (usato in problemi di classificazione, dove ogni neurone dell'ultimo strato corrisponde a una possibile categoria o etichetta, rende possibile interpretare l'output come la probabilità, normalizzata attraverso esponenziali, che l'input appartenga a ciascuna categoria), nel nostro caso utilizzeremo semplicemente uno strato denso come appena descritto.

## 2.2 Training

Nel seguito illustriamo i dettagli del processo di *training*, già descritto in sintesi nella sezione 1.2, concentrandoci su quelli che utilizzeremo per i nostri scopi.

### 2.2.1 Funzione di costo

Come già detto, per ottenere un modello che riproduca la distribuzione di probabilità desiderata dobbiamo "allenarlo" su un insieme di dati, sia esso provvisto o no di etichette. Per farlo, è necessario minimizzare con metodi variazionali una funzione di perdita. Viene spontaneo chiedersi se la funzione di perdita sia univocamente determinata dal problema o se possa essere scelta liberamente. La risposta non è univoca.
In molti problemi di *machine learning* si tratta di una scelta relativamente libera, guidata da considerazioni del programmatore sul set di dati, a volte procedendo per tentativi. Ad esempio, lo scarto quadratico medio (equazione 1.2) o "norma L2" è una delle più semplici e più comuni funzioni di costo usate nei problemi di allenamento supervisionato. Se nel campione di dati esistono degli *outliers* molto distanti dal valor medio, però, essi possono influire negativamente sul processo di ottimizzazione, spostando di molto il minimo della funzione di perdita: il modello darà così eccessiva importanza al trovare una buona etichetta per pochi dati, anche a costo di assegnarne una peggiore al grosso dei dati benché questi ultimi seguano una distribuzione di probabilità con piccola varianza. Se non vogliamo scartare gli *outliers*, allora, possiamo utilizzare una funzione di perdita ad essi meno sensibile (come può essere verificato) come la "norma L1":

$$\mathrm{L}_1(\boldsymbol{\theta}) = \frac{1}{M}\sum_{i=1}^{M} \|\mathbf{y_i} - \mathbf{f}_\theta(\mathbf{x}_i)\|_\mathbf{1}$$

Dove, se $\mathbf{u} \in \mathbb{R}^d$, $\|\mathbf{u}\|_\mathbf{1} \equiv \sum_{j=1}^{d} |u_i|$ indica la norma $\mathcal{L}^1$.
Un'altra procedura che coinvolge funzioni di perdita modificate consiste nell'introdurre un termine detto di "regolarizzazione" alla funzione di perdita originale. Ad esempio, se si vuole evitare che i parametri da ottimizzare assumano valori troppo grandi, si può effettuare la cosiddetta "regressione lasso", per la quale la funzione di costo è:

$$\mathrm{L}_{lasso}(\boldsymbol{\theta};\lambda) = \frac{1}{M}\sum_{i=1}^{M} \|\mathbf{y_i} - \mathbf{f}_\theta(\mathbf{x}_i)\|_\mathbf{2}^2 + \lambda \sum_{j=1}^{N} |\theta_j|$$
$$\lambda \geq 0$$

in questo caso, si sceglie un valore per l'iperparametro $\lambda$, quindi si effettua come al solito il *training* del modello (ottimizzazione degli $N$ parametri $\theta_j$); poi si valuta la funzione di perdita su un diverso set di dati detto di "validazione" (ovviamente un sottinsieme selezionato dal set di dati a disposizione, disgiunto dal sottinsieme di *training*), il cosiddetto "errore di validazione"; si sceglie un nuovo valore di $\lambda$ e si ripete l'intero processo, fino a trovare il valore dell'iperparametro che minimizza l'errore di validazione.
In altri casi, invece, la funzione di perdita viene dedotta in modo rigoroso. Per fare ciò

spesso si utilizza il "principio di massima verosimiglianza", cioè si cerca di massimizzare la probabilità che l'intero set di dati sia stato estratto dalla distribuzione di probabilità data dal modello. Per problemi di allenamento supervisionato, assumendo che ciascun dato sia stato estratto in modo indipendente dalla stessa distribuzione "vera", si ha allora che la probabilità in questione è il prodotto delle probabilità di ciascun campione di essere stato estratto; per praticità si prende il logaritmo di questa probabilità totale (per le proprietà dei logaritmi, il prodotto diventerà quindi una somma); massimizzare la probabilità totale equivale a massimizzare questo logaritmo (per la monotonicità della funzione logaritmo), e per esprimere questo problema come la minimizzazione di una funzione di perdita, basterà considerare come funzione di perdita l'opposto della somma di logaritmi in questione. Così ad esempio si può derivare la comune funzione di perdita detta "*cross entropy*":

$$L_{bin.ent}(\boldsymbol{\theta}) = -\frac{1}{M}\sum_{i=1}^{M}[y_i \log(f_\theta(x_i)) + (1-y_i)\log(1-f_\theta(x_i))]$$

qui espressa nel caso in cui gli output e le etichette sono numeri reali fra 0 e 1 (esclusi). Nel caso di output ed etichette vettoriali, la scrittura rimarrebbe la stessa, con il significato però che ogni operazione è fatta "elemento per elemento" e che la funzione di perdita è calcolata come la somma (di $m$ termini, dove $m$ è la dimensione del singolo vettore etichetta o output) di queste componenti.
Alcune riflessioni sulle funzioni di perdita e sul modo spesso soggettivo in cui vengono scelte si può trovare in [5].

### 2.2.2 Kernel initializer

Come già detto, nella fase di *training* di una rete neurale cerchiamo il minimo della funzione di perdita rispetto a ciascuno dei parametri con un algoritmo iterativo, che calcola a ciascuno *step* il valore del gradiente della funzione di perdita in quel punto. Sorge allora spontaneo chiedersi da quale punto si debba partire, o in altre parole quale valore dobbiamo assegnare ai parametri all'inizio della procedura. In modo forse poco intuitivo, questa scelta può influire sulla bontà del minimo (locale) a cui si vuole convergere.
Forse il più semplice *kernel initializer* cui si possa pensare è quello "normale": si estrae il valore di ogni peso da una distribuzione gaussiana di media 0 e deviazione standard 1. Una versione appena più elaborata, che evita la possibilità di valori iniziali molto distanti dalla media, consiste nel ri-estrarre i valori lontani più di due deviazioni standard dalla media ed è detta perciò "troncata". Nel nostro caso abbiamo scelto la "normale Glorot" o "Xavier", una troncata in cui però la deviazione standard è $\sigma = \sqrt{\frac{2}{n_i+n_o}}$ dove $n_i$, $n_o$ sono rispettivamente il numero di neuroni di input e di output. In principio, comunque, si sarebbero potuti usare altri *initializers*.
Una panoramica su di essi si può trovare in [6].

### 2.2.3 Automatic differentiation

Affrontiamo ora la parte meno immediata dell'equazione 1.4, cioè il calcolo delle derivate parziali della funzione di perdita, rispetto a ciascuno dei parametri ottimizzabili e nel

punto $\boldsymbol{\theta}^j$ della $j$-esima iterazione. Si potrebbe pensare che ad ogni iterazione ciascuna di queste derivate debba essere calcolata separatamente, il che avrebbe un elevato costo computazionale, visto che specie nelle architetture più moderne il numero di parametri da ottimizzare è estremamente elevato (anche nell'ordine di $10^9$). In realtà, nel caso di una rete neurale densa, che è essenzialmente costituita da funzioni parametriche annidate una dentro l'altra, queste derivate possono essere calcolate facilmente sfruttando una generalizzazione della "regola della catena". Utilizzando la stessa notazione della sezione 2.1, raccogliendo nel vettore $\boldsymbol{\theta}^{[k]}$ i parametri (numeri reali) ottimizzabili dello strato $k$ (abbiamo $n$ strati in totale), e indicando con il pedice $\alpha$ un certo elemento di questo vettore, abbiamo:

$$\frac{\partial L}{\partial \boldsymbol{\theta}_\alpha^{[k]}} = (\nabla_{\mathbf{z}^{[k]}} L)^T \cdot \frac{\partial \mathbf{z}^{[k]}}{\partial \boldsymbol{\theta}_\alpha^{[k]}} =$$

$$((\nabla_{\mathbf{f}^{[n]}} L)^T \cdot \frac{\partial \mathbf{f}^{[n]}}{\partial \mathbf{z}^{[k]}}) \cdot (\frac{\partial \mathbf{g}^{[k]}}{\partial \mathbf{q}^{[k]}} \cdot \frac{\partial \mathbf{q}^{[k]}}{\partial \boldsymbol{\theta}_\alpha^{[k]}}) =$$

$$((\nabla_{\mathbf{f}^{[n]}} L)^T \cdot \prod_{i=k+1}^{n} \left[ \frac{\partial \mathbf{f}^{[i]}}{\partial \mathbf{z}^{[i-1]}} \right]) \cdot (\frac{\partial \mathbf{g}^{[k]}}{\partial \mathbf{q}^{[k]}} \cdot \frac{\partial \mathbf{q}^{[k]}}{\partial \boldsymbol{\theta}_\alpha^{[k]}})$$

Dove abbiamo semplicemente applicato più volte la regola di derivazione della catena per funzioni di funzioni; nell'ultima uguaglianza, ricordando che $\mathbf{z}^{[k]} = \mathbf{f}^{[k]}(\mathbf{z}^{[k-1]}) = \mathbf{g}^{[k]}(\mathbf{q}^{[k]}(\mathbf{z}^{[k-1]}))$ (l'output di uno strato è l'input del successivo) abbiamo scritto la derivata dello strato $k$-esimo come prodotto delle derivate dallo strato $k+1$ allo strato $n$. Per quanto riguarda la notazione qui usata, puramente matriciale, il gradiente di uno scalare rispetto a un vettore equivale alla derivata direzionale dello scalare (che è a sua volta un vettore, colonna), T indica la trasposizione e il punto indica esclusivamente la moltiplicazione di matrici.
Questa procedura, detta *backpropagation* in quanto le derivate di uno strato sono calcolate, a partire dallo strato finale, "all'indietro" rispetto al verso normale di propagazione dell'input, è un caso particolare di *automatic differentiation*, un insieme di metodi computazionali che basandosi sulla regola della catena e sulle espressioni delle derivate elementari consentono di calcolare derivate di qualsiasi ordine di funzioni anche molto complesse.

### 2.2.4 Ottimizzatori

Abbiamo già introdotto un semplice metodo per minimizzare la funzione di perdita $L(\boldsymbol{\theta})$, la cosiddetta discesa del gradiente, nella sezione 1.2. Nella pratica, però, conviene usare versioni più sofisticate di questo algoritmo, o "ottimizzatori".
Uno di quelli più efficienti all'atto pratico e perciò più usati (scelto anche da noi) è il cosiddetto *Adam* (*Adaptive moment estimation*), introdotto per la prima volta in [7]. Lo illustriamo qui evidenziandone le somiglianze con altri più semplici metodi iterativi di ottimizzazione, ricordando l'analogia fisica già vista: consideriamo un oggetto che scivola spinto dalla forza di gravità lungo una superficie bidimensionale con avvallamenti, creste, selle, molti massimi e minimi locali ecc., e vogliamo fare in modo che esso raggiunga un minimo locale abbastanza vicino in quota al minimo assoluto.
Innanzitutto, ricordiamo che ad ogni iterazione successiva, il gradiente viene calcolato non

in tutti i punti del campione, ma in un *minibatch* scelto in maniera casuale, secondo quanto già visto in 1.2. Questa è l'idea alla base del metodo di "discesa stocastica del gradiente" (*SGD*), la più immediata generalizzazione della "semplice" discesa del gradiente. In secondo luogo, viene aggiunto un termine "d'inerzia" $\mathbf{m}$, per far sì che l'oggetto "continui a scivolare", ad esempio anche se la pendenza diventa negativa e, per evitare che esso "prenda troppa velocità", viene anche aggiunto un fattore di "attrito" $\beta_1$ (idea del *Momentum optimizer*). L'ulteriore tipo di approccio che viene incluso in *Adam* è quello dell'ottimizzatore detto *RMS Prop* (a sua volta una modifica di *AdaGrad*): si normalizza il vettore gradiente tenendo conto dei passati valori da esso assunti, nel senso che si accumulano in un vettore $\mathbf{s}$ i quadrati delle componenti del gradiente, più un iperparametro $\epsilon$ per evitare la divisione per 0, e si moltiplica poi il vettore gradiente per l'inverso della radice di questa somma. Anche nel caso di $\mathbf{s}$ s'introduce un ulteriore iperparametro $\beta_2$ (piccolo) del tutto analogo a quello di attrito, che fa sì che nella normalizzazione vengano considerati solo i più recenti valori assunti dal gradiente. Indicando con $T$ il numero dell'epoca considerata (quindi introducendo un ulteriore livello di "adattabilità" all'algoritmo), con un cerchio attorno al loro simbolo le operazioni elemento per elemento, e con una freccia il procedimento di assegnare a una certa variabile il suo nuovo valore, ogni iterazione dell'algoritmo *Adam* consiste in:

$$
\begin{aligned}
&\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\,\nabla_\theta L(\boldsymbol{\theta}),\\
&\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\,\nabla_\theta L(\boldsymbol{\theta}) \otimes \nabla_\theta L(\boldsymbol{\theta}),\\
&\mathbf{m} \leftarrow \frac{\mathbf{m}}{1-\beta_1^T},\\
&\mathbf{s} \leftarrow \frac{\mathbf{s}}{1-\beta_2^T},\\
&\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta(T)\mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}.
\end{aligned}
$$

Dove l'iperparametro $\eta(T)$, detto *learning rate*, è il "passo" da fare all'iterazione $T$; può anch'esso variare da un'iterazione all'altra come illustreremo qui di seguito.

### 2.2.5 Learning rate scheduling

Come appena detto, nel corso del processo di *training*, il *learning rate* $\eta$ (l'ultimo iperparametro di cui ci occuperemo nonché uno dei più importanti) può non rimanere costante ma essere una funzione del numero dell'iterazione corrente $T$; lo indichiamo perciò come appena visto con $\eta(T)$. Come nel caso di altri iperparametri, ci sono molte possibili scelte di questa dipendenza "temporale", che va sotto il nome di *learning rate scheduling*; in generale, comunque, vogliamo che $\eta(T)$ sia decrescente in senso lato. Il perché può essere compreso facilmente con un'analogia: immaginiamo una funzione di perdita convessa e dipendente da una sola variabile, quindi come una parabola rivolta verso l'alto; se all'inizio ci troviamo distanti dal vertice, vogliamo fare dei "passi" lunghi per avvicinarci ad esso nel più breve tempo possibile; ma se una volta vicini il passo rimane troppo lungo, continueremo a "saltare via" il minimo passando continuamente da un ramo all'altro della parabola.

Alcune comuni routine di *learning rate scheduling* sono: il decadimento *time based*, dove cioè $\eta(T)$ è semplicemente una funzione decrescente linearmente in $T$; il decadimento "esponenziale", il cui nome è autoesplicativo; e il decadimento "costante a tratti" o *step decay*, che useremo, formalmente:

$$\eta(T) = \begin{cases} \eta_0 & \text{se} \quad 0 < T \leq T_0 \\ \eta_1 & \text{se} \quad T_0 < T \leq T_1 \\ \cdots & \\ \eta_k & \text{se} \quad T_{k-1} < T \leq T_k \end{cases}$$

dove come detto $\eta_i < \eta_{i+1} \quad \forall i = 0, 1, \cdots, k-1$. Un esempio del grafico di questa funzione è dato in figura 2.3.

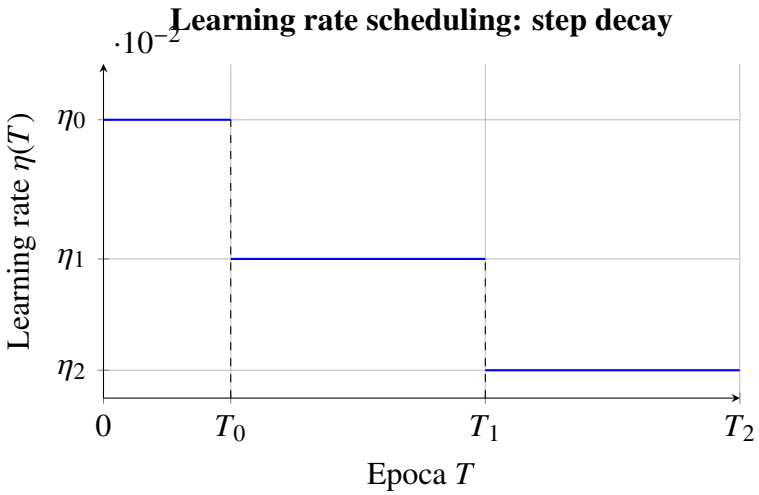


**Figura 2.3:** *Grafico del possibile andamento, fissato a priori, del learning rate $\eta(T)$ con decadimento costante a tratti, in funzione dell'epoca $T$.*

## 2.3 Dopo il *training*

Nella sezione precedente abbiamo illustrato il processo di *training* di una rete neurale, il quale però non basta ad assicurarci di avere un buon modello. Ad esso, infatti, seguono in genere le procedure di *regolarizzazione* e *testing*, che abbiamo già brevemente menzionato nelle sezioni 2.2.1 e 1.2. In questa sezione li illustriamo in maniera più completa, benché sintetica.

### 2.3.1 Regolarizzazione

In generale (si veda [8], cap. 7), se abbiamo un problema di *machine learning*, possiamo scrivere $H \equiv \bigcup_n H_n$ con $H_n \equiv \{h_n : h_n \text{ ipotesi di tipo } n, \quad n = 1, 2, \cdots, N\}$, dove con $n$ indicizziamo modelli "di diverso tipo". Vogliamo trovare la migliore ipotesi $h_n*$ (cioè

quella per cui la funzione di perdita ha valore minimo su un campione di dati $S$ a nostra disposizione, che sappiamo essere stato estratto da una certa distribuzione $D$ non nota) per ogni "classe di ipotesi" $H_n$; poi, trovare fra queste la migliore. Si può mostrare (vedi ad esempio [8]) che, assumendo un campione di dati indipendenti e identicamente distribuiti, è importante che la prima parte del processo sia fatta su dati diversi rispetto alla seconda, se si vuole ottenere un modello che rispecchi veramente la distribuzione $D$.
Nel caso particolare di un problema di allenamento supervisionato di una rete neurale in cui abbiamo a disposizione un campione $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ di dati con rispettive etichette, generalmente suddividiamo $S$ in maniera casuale in tre sottinsiemi disgiunti, di dimensione fissata: quello di *training* (il più grande), quello di *validation* (usato per la regolarizzazione) e quello di *testing*. Si sceglie un iperparametro $\epsilon_n(h_n)$, fissato a priori per ogni $n$ e dipendente in modo esplicito dalla specifica ipotesi, che viene incluso nella funzione di perdita di solito come un termine positivo. Perciò, per ogni $n$ fissato, si minimizza la funzione di perdita rispetto ai parametri (pesi e *bias*) della rete neurale usando i dati del sottinsieme di *training*, trovando così $h_n*$ per ogni $n$; poi, sull'insieme di *validation*, si calcola il valore della funzione di perdita usando di volta in volta come modello $h_1*, h_2*, \cdots, h_N*$; il "miglior" modello sarà quello per cui questa funzione di perdita è minima. Nel caso della procedura detta di "regolarizzazione", $\epsilon_n$ è semplicemente un numero reale diverso per ogni $n$, e l'ipotesi $h_n$ non dipende davvero da $n$ ma solo dal valore di pesi e *bias*. Esistono ovviamente varianti di questa procedura; ad esempio, per sfruttare al massimo il campione di dati disponibile, si possono parzialmente riutilizzare come insieme di validazione dati già usati per il *training* e viceversa, avendo cura di fare una media della funzione di perdita in ciascun caso, e riallenando alla fine il modello sull'intero set di dati (metodo detto *k-fold cross validation*). In ogni caso, comunque, all'interno di un singola iterazione algoritmica validazione e allenamento devono avvenire su insiemi di dati disgiunti.

### 2.3.2 *Testing* e il problema dell'*overfitting*

Il processo di *testing* del modello, come detto, avviene una volta fissati definitivamente i parametri e iperparametri. Esso consiste semplicemente nel calcolare il valore (medio) della funzione di perdita sul sottinsieme di *testing*, il quale va sempre scelto prima del *training*, disgiunto dai sottinsiemi di *testing* e di *validation*, e mai mostrato al modello se non in questa fase finale. Tale valore prende il nome di "errore di generalizzazione".
Il *testing* è necessario per controllare che il modello classifichi correttamente anche campioni di dati mai visti prima. Se ciò non accade, benché la funzione di perdita assuma un valor medio abbastanza piccolo sul sottinsieme di *training*, diciamo che il modello sta facendo *overfitting* dei dati, in altre parole che esso ha imparato a considerare significative per la classificazione delle *features* del singolo dato che in realtà non lo sono (il cosiddetto "rumore"); o, in altri termini ancora, che ha "memorizzato" la classe corretta per molti campioni, senza aver davvero "imparato" nulla. Questo è un problema molto comune nel *machine learning*, che affligge in particolare modelli troppo complessi (ad esempio con molti parametri) per il problema da risolvere. Le strategie per evitarlo sono molte. Una è la procedura di regolarizzazione vista sopra; oppure, in modo simile, potremmo scegliere come classi di ipotesi $H_n$ modelli con gradi di complessità crescenti con $n$, ad esempio

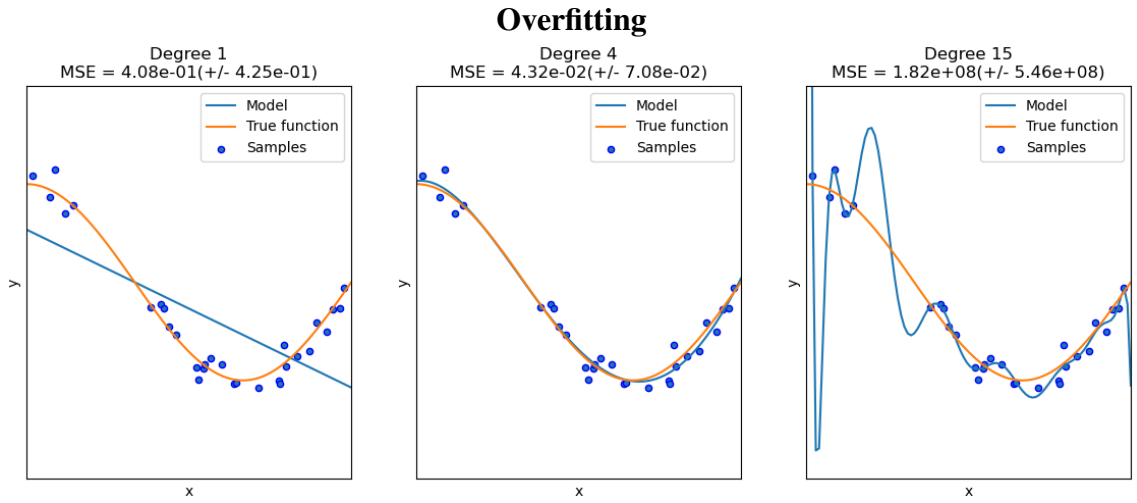


**Figura 2.4:** *Esempio di overfitting. Qui si cerca di "imparare" una funzione (in arancio) a partire da alcune misure con rumore (punti in blu). La funzione di perdita è lo scarto quadratico medio, che dipende dalla distanza fra ciascuna misura e il valore della funzione modello in quel punto. Il modello è un polinomio in cui i parametri ottimizzabili sono i coefficienti, rispettivamente di grado 1, 4 e 15. Passando dal grado 1 al 4 l'approssimazione migliora, ma al grado 15 il modello è essenzialmente una funzione che passa quasi esattamente per tutti i punti dati, rendendo quasi nulla la funzione di perdita. Questa, però, verosimilmente risulterebbe grande se fosse calcolata in nuovi punti (testing) appartenenti alla funzione "vera".*

per numero di parametri, e porre $\epsilon_n(h)$ tanto maggiore quanto più l'ipotesi $h_n \in H_n$ è complessa cioè $n \in N$ grande. Altre tecniche vengono usate in fase di *training*, ad esempio selezionando casualmente per determinati strati un certo numero di neuroni da "spegnere" (che saranno comunque sempre "accesi" nel modello finale). La strategia più elementare, comunque, consiste nello scegliere modelli non troppo complicati per il problema da risolvere; naturalmente, però, essi non devono essere neppure troppo semplici, altrimenti non riusciremo a classificare con buona precisione alcun campione (in statistica ciò equivale a dire che il modello è *biased*). In altre parole, c'è spesso un *trade-off* tra complessità e *bias*; trovare il giusto grado di complessità per avere il minimo errore di generalizzazione possibile è spesso uno dei problemi principali.

# 3. *Physics-informed neural networks* (PINNs)

In questo capitolo mostriamo come le reti neurali discusse finora possano essere usate per risolvere equazioni differenziali, in generale alle derivate parziali (PDEs), che sono comunissime in tutti i campi della fisica: si pensi alla meccanica dei fluidi, alla termodinamica (equazione del calore), alla fisica dei sistemi complessi, all'elettromagnetismo (equazioni di Maxwell), ma anche alla meccanica quantistica (equazione di Schrödinger).
Benché in ambito scientifico il *machine learning* in generale e le reti neurali in particolare vengano già usati da tempo in problemi di "*big data*" (si pensi ad esempio all'uso di questi metodi per la scoperta di nuove particelle come in [9], che è comunque un'analisi dati a posteriori, o per la classificazione morfologica di galassie, [10]), nuove possibili applicazioni sono apparse più di recente (ad esempio, di tipo generativo per prevedere il ripiegamento delle proteine in [11], valsa il premio Nobel, o addirittura nel guidare l'intuizione matematica come descritto in [12]). Molte applicazioni come queste si sono rivelate vincenti perché le reti neurali mostrano una notevole capacità di estrarre informazioni significative da campioni di dati con spazio delle caratteristiche di dimensioni troppo elevate perché un essere umano sia in grado di fare lo stesso. In effetti, è stato dimostrato che alcune architetture di reti neurali possano in linea di principio approssimare funzioni continue con arbitraria precisione (i parametri variazionali hanno il ruolo dei coefficienti in una serie di Taylor o di Fourier); un fondamentale teorema in questo ambito fu provato già negli anni Ottanta in [13] e oggi ne esistono numerose versioni e generalizzazioni. D'altro lato, però, è sempre ben presente il rischio di *overfitting*, insieme al problema dell'"interpretabilità": vale a dire che, nonostante siano oggetto di intensa ricerca, la nostra comprensione matematica di questi modelli è ancora scarsa e nella maggior parte dei casi è quasi impossibile capire "perché" una rete neurale funzioni, o cosa esattamente abbia imparato (visto anche l'elevatissimo numero di parametri). Inoltre, è importante notare che quel teorema garantisce l'esistenza di una rete in grado di approssimare la funzione, ma non fornisce un metodo per trovare i parametri ottimali $\boldsymbol{\theta}$, né garantisce che l'addestramento converga a tale rete.
Comunque, noi siamo interessati a un ambito più ristretto, più strettamente matematico-computazionale: proprio perché con una rete neurale possiamo approssimare funzioni di qualsiasi tipo, anche molto complicate (in altre parole, nella pratica una rete neurale con un sufficiente numero di parametri e architettura adeguata può approssimare qualsiasi tipo di distribuzione di probabilità), vogliamo fare in modo che la funzione approssimata sia la soluzione dell'equazione differenziale da risolvere. Una delle implementazioni possibili, su cui torneremo più sotto, è quello delle *Physics-informed neural networks* (o PINNs), un approccio introdotto per la prima volta in [14], benché idee simili fossero state avanzate già negli anni Novanta del secolo scorso. Insomma, stiamo usando le reti neurali per uno scopo diverso da quello per cui sono state originariamente pensate, più strettamente legato alla loro struttura puramente matematica.

## 3.1 Equazioni differenziali alle derivate parziali

Come detto, il problema di nostro interesse è la soluzione di equazioni differenziali alle derivate parziali (PDEs). Si tratta di un ambito di ricerca vastissimo e attuale, oltre che poco organico: questo tipo di equazioni emergono frequentemente nei più disparati campi della fisica e della matematica, e i tipi di soluzioni sono altrettanto vari. Per questo motivo, decidiamo di non scendere troppo nel dettaglio della teoria (si veda eventualmente [15]), ma ci limitiamo a illustrarne gli aspetti rilevanti per il nostro progetto.
Formalmente, sia $u$ una funzione $u : U \subset \mathbb{R}^n \to \mathbb{R}$, con $U$ aperto, e sia $\xi \equiv (\xi_1, \xi_2, \cdots, \xi_n) \in U$. Chiamiamo un generico vettore $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$, con ogni $\alpha_i$ intero non negativo, "multi-indice di ordine $|\alpha|$", con $|\alpha| = \sum_{i=1}^{n} \alpha_i$. Indichiamo poi per brevità:

$$D^\alpha u(\xi) \equiv \frac{\partial^{|\alpha|} u}{\partial^{\alpha_1}\xi_1 \partial^{\alpha_2}\xi_2 \cdots \partial^{\alpha_n}\xi_n}$$

Se $k$ è un intero non negativo, chiamiamo $D^k u(\xi) \equiv \{D^\alpha u(\xi) : |\alpha| = k\}$ l'insieme di tutte le derivate parziali di $u$ di ordine $k$. Definiamo dunque "equazione differenziale alle derivate parziali di ordine $k$" un'equazione in cui compare come incognita $u$, più precisamente un'espressione del tipo:

$$F(D^k u(\xi), D^{k-1} u(\xi), \cdots, Du(\xi), u(\xi), \xi) = 0 \tag{3.1}$$

con $F$ data, $F : \mathbb{R}^{n^k} \times \mathbb{R}^{n^{k-1}} \times \cdots \times \mathbb{R}^n \times \mathbb{R} \times U \to \mathbb{R}$ (in quanto notiamo che ci sono $n^k$ modi distinti di derivare parzialmente $k$ volte una funzione di $n$ variabili reali). "Risolvere" l'equazione differenziale significa trovare tutte le $u$ che verificano 3.1, (se richiesto) solo fra quelle che soddisfano alcune "condizioni al contorno" su qualche parte di $\partial U$.
Un'equazione differenziale alle derivate parziali si dice inoltre "lineare" se è della forma: $\sum_{|\alpha| \le k} [a_\alpha(\xi) \cdot D^\alpha(u)] = f(\xi)$ (per date funzioni $a_\alpha$, $f$); "non lineare" altrimenti. Fra le PDEs non lineari, chiamiamo "quasi-lineari" quelle della forma:

$$\sum_{|\alpha|=k} [a_\alpha(D^{k-1}, \cdots, Du, \xi) \cdot D^\alpha u] + a_0(D^{k-1}, \cdots, Du, \xi) = 0$$

(in cui cioè si può isolare il termine di ordine massimo in modo che il suo coefficiente dipenda solo dalle derivate di ordine inferiore).

## 3.2 Soluzioni approssimate e residui

Come detto, per risolvere molte PDEs si deve spesso ricorrere a metodi approssimati, che fanno affidamento sui calcolatori per trovare soluzione numeriche, di cui esistono svariati tipi. Alcuni di essi si basano sul semplice concetto di "residuo" di una data PDE rispetto a una certa funzione $u_\theta$ a valori reali, parametrizzata da $\boldsymbol{\theta}$, che supponiamo sia una buona approssimazione della soluzione della PDE, continua e derivabile quante volte necessario su tutto l'aperto in cui cerchiamo soluzioni alla PDE. [1] Chiamiamo in questo

[1] Nel fare ciò, implicitamente stiamo assumendo che la PDE ammetta un'unica soluzione, continua e derivabile almeno $k$ volte, sull'aperto $U$; questa proprietà non è affatto banale, ma mostreremo in seguito che la PDE da noi considerata la soddisfa.

caso "residuo" (forte) della PDE il valore assunto dall'espressione 3.1 sostituendovi $u$ con la soluzione approssimata $u_\theta$:

$$r_\theta(\xi) \equiv F(D^k u_\theta(\xi), D^{k-1} u_\theta(\xi), \cdots, Du_\theta(\xi), u_\theta(\xi), \xi) \tag{3.2}$$

Ovviamente, quanto più piccolo è il residuo in valore assoluto tanto più l'approssimazione sarà considerata buona.
Vari metodi approssimati si basano sui residui; ad esempio, quello delle *Residual Power Series* (RPS) illustrato in [16] si basa su un'idea semplice: si sviluppa la funzione incognita in serie di Taylor fino a un certo ordine; si calcolano le sue derivate necessarie a ottenere l'espressione del residuo; infine, si cerca il minimo della funzione residuo (ponendo le sue derivate prime uguali a zero), riconducendo così equazioni differenziali anche molto complesse a un sistema di equazioni algebriche, assai più facilmente trattabili a livello computazionale.

## 3.3 Approssimare la funzione soluzione con una rete neurale

L'idea alla base delle *Physics-informed neural networks* (PINNs), per approssimare la soluzione di equazioni differenziali alle derivate parziali tramite reti neurali, è che la funzione di perdita da minimizzare comprenda in qualche modo l'equazione differenziale da risolvere (in questo senso includendo la "fisica" nella rete neurale). Per calcolare le derivate parziali presenti nell'equazione si sfrutta poi la differenziazione automatica, che è implementata di default nelle librerie per *machine learning*. Infine, si effettuano *training* e *testing* in allenamento supervisionato, su un campione di dati costituito semplicemente da molte coppie input-output della funzione da approssimare, tipicamente le condizioni al contorno e alcuni punti all'interno del dominio (da soli, o per cui un valore approssimato della funzione soluzione sia stato calcolato con metodi tradizionali), dunque in un contesto di "*small data*".

### 3.3.1 PINNs per PDEs di evoluzione temporale

Per i nostri scopi, ci concentriamo su particolari PDEs dette "di evoluzione temporale", in cui cioè la derivata rispetto al tempo può essere separata (con coefficiente uguale a 1), che siano inoltre definite su un dominio limitato nel tempo e nello spazio, la cui soluzione sia nota al tempo iniziale in ogni punto dello spazio, e lungo tutto il bordo del dominio spaziale per ogni istante temporale (condizioni al contorno di tipo "Dirichlet non omogeneo"); in notazione: detti $U = ]0, T[ \times S$ ; $S \subset \mathbb{R}^s$ aperto, $s \in \mathbb{N}$; $\xi = (t, x) = (t, x_1, \cdots, x_s) \in U$,

$$\begin{aligned} \partial_t u(t,x) + \mathcal{N}[u](t,x) &= 0 \\ \text{in } U, \text{ con } \mathcal{N} &\text{ operatore differenziale;} \\ u(0,x) \equiv u_0(x), \quad u_0 : S &\to \mathbb{R} \text{ (nota)} \\ u(t,x) \equiv u_b(t,x) \text{ se } x \in \partial S, \quad u_b : [0,T] \times \partial S &\to \mathbb{R} \text{ (nota)} \end{aligned} \tag{3.3}$$

(Dove come di consueto le condizioni al contorno sono definite sulla chiusura dell'aperto $U$, benché strettamente parlando l'equazione differenziale con relativi operatori sia definita solo su $U$). Come detto, la rete neurale è un approssimazione della soluzione "vera" parametrizzata dai parametri $\boldsymbol{\theta}$: $u_\theta(t,x) \approx u(t,x)$. Discuteremo l'esatta architettura da noi usata in seguito, riferendoci a quanto illustrato nei capitoli precedenti.
Per quanto riguarda il campione di dati di *training*, indicizzati ciascuno con (un diverso) $i$, esso è costituito semplicemente da punti $\xi_i^r = (t_i, x_i) \in U$, all'interno del dominio spazio-temporale (che sceglieremo in modo casuale); sceglieremo inoltre alcuni punti con $t = 0$, $\xi_i^0 = (0, x_i) \in \{0\} \times S$ e altri sul "bordo" spaziale, $\xi_i^b = (t_i, x_i) \in [0,T] \times \partial S$. Per queste ultime due categorie di punti, dato che ci vengono fornite $u_0$ e $u_b$, potremo anche calcolare le "etichette" corrispondenti, vale a dire $u(\xi_i^0) = u_0(\xi_i^0)$ e $u(\xi_i^b) = u_b(\xi_i^b)$ rispettivamente. Occupiamoci ora della funzione di perdita, il cuore di questo approccio. Per quanto riguarda i punti corrispondenti alle condizioni al contorno, è intuitivo che la funzione di perdita da minimizzare debba essere un qualche tipo di distanza fra la funzione approssimata e quella "vera" (etichetta) calcolate entrambe nel dato punto; la più semplice a cui possiamo pensare è lo scarto quadratico medio (MSE, vedi equazione 1.2). Per i punti interni, invece, non disponendo di etichette ricorriamo al residuo (equazione 3.2): la distanza da minimizzare sarà quella fra il valore dell'equazione differenziale in cui è stata sostituita la funzione approssimata calcolata in un dato punto, per definizione il residuo in quel punto, e lo zero (che può essere considerato l'etichetta). Decidiamo di usare anche in questo caso lo MSE, e di definire la funzione di perdita come somma di questi tre termini (calcolati ovviamente ciascuno su tutti gli $N_r$, $N_0$, $N_b$ punti di ciascun tipo). Definendo $\Xi \equiv \Xi_r \cup \Xi_0 \cup \Xi_b$ dove $\Xi_c \equiv \{\xi_i : i = 1, \cdots, N_c\}$ per ogni $c = r, 0, b$, la funzione di perdita $L_\theta(\Xi)$ per la rete neurale parametrizzata da $\boldsymbol{\theta}$ calcolata su $\Xi$ sarà quindi:

$$L_\theta(\Xi) = \frac{1}{N_r}\sum_{i=1}^{N_r} |r_\theta(\xi_i^r)|^2 + \frac{1}{N_0}\sum_{i=1}^{N_0} |u_\theta(\xi_i^0) - u_0(\xi_i^0)|^2 + \frac{1}{N_b}\sum_{i=1}^{N_b} |u_\theta(\xi_i^b) - u_b(\xi_i^b)|^2 \tag{3.4}$$

Come detto, per calcolare il residuo è necessario calcolare un certo numero (a seconda della PDE) di derivate di $u_\theta$ rispetto al tempo o alle componenti del vettore spaziale, anche di ordine superiore al primo. Nella pratica però, come vedremo in seguito, questo può essere fatto molto facilmente, utilizzando le funzionalità di *automatic differentiation* delle principali librerie di programmazione per *machine learning*. Infine, osserviamo che la procedura di *testing* si può ridurre in questo caso a calcolare il valore della funzione di perdita su ulteriori punti generati in modo casuale, con le stesse caratteristiche dei precedenti (benché in numero minore); un valore della funzione di perdita non troppo grande rispetto a quello ottenuto in fase di *training* assicurerà l'assenza di *overfitting* e indicherà una buona approssimazione della soluzione (attraverso il residuo).

### 3.3.2 Alcune possibili generalizzazioni

Concludiamo questo capitolo con alcune considerazioni generali sulle PINNs, di cui abbiamo evidenziato solo le caratteristiche principali utili per i nostri scopi. La letteratura scientifica anche recente in merito è piuttosto vasta, perciò rimandiamo alla *review* [2].

Diversi tipi di PINNs usano approcci leggermente differenti rispetto a quello illustrato sopra. Ad esempio, il valore della funzione che risolve la PDE può essere noto a certi istanti spazio-temporali, costituendo così un'etichetta, perché misurato sperimentalmente (includerà allora del "rumore" gaussiano) oppure calcolato con approssimazioni numeriche "classiche". Esempi di queste ultime sono i metodi di tipo Runge-Kutta (impliciti o espliciti; generalizzano il metodo di Eulero), algoritmi iterativi relativamente semplici che si fondano sulla discretizzazione del dominio e il calcolo di derivate, o la cosiddetta approssimazione di Picard, basata invece sull'integrazione definita. Nel caso di particolari PDEs in molte dimensioni, invece, per gli stessi scopi si possono usare simulazioni numeriche come quella di Eulero-Maruyama, che sfruttano il legame fra la soluzione di certe PDEs e il valore di aspettazione di processi stocastici, come il moto browniano, dato dalla formula di Feynman-Kac. L'architettura è in questi casi più complessa, prevedendo comunque un uso ibrido di approssimazioni numeriche classiche e di reti neurali allenate minimizzando il residuo. In altri casi, l'equazione differenziale di partenza (dunque anche la funzione di perdita) può includere un parametro non noto; basterà allora minimizzare la funzione di perdita anche rispetto a questo parametro, per "impararlo" alla stregua degli altri. In altri casi ancora, la funzione di perdita viene parametrizzata o resa localmente adattiva per aumentare la velocità di convergenza. Nei cosiddetti *Extended PINNs*, infine, si decompone il dominio usando diverse reti neurali per approssimare parti diverse della funzione soluzione.
Va notato inoltre che la capacità di lavorare in regimi di *small data* è particolarmente utile in ambito fisico-sperimentale, dal momento che spesso l'acquisizione di nuovi dati può essere molto difficile o costosa.
Osserviamo infine che il successo di quest'architettura alla prova dei fatti è giustificato parzialmente anche in ambito matematico: si veda ad esempio il recente [17].

# 4. Equazione di Burgers con PINN

In questo capitolo presentiamo l'equazione di Burgers, un'equazione differenziale alle derivate parziali di evoluzione temporale che può essere ottenuta dalle equazioni di Navier-Stokes (il più generale sistema di PDEs che governa il moto di un fluido). Si tratta di una PDE difficile da trattare numericamente in regime di bassa viscosità perché la soluzione è continua ma presenta dei "fronti" di salita particolarmente ripidi. Illustriamo quindi un approccio con un semplice PINN e mostriamo la bontà della soluzione ottenuta.

## 4.1 Equazione di Navier-Stokes

Si consideri un fluido incomprimibile con viscosità (di taglio) e densità costanti e uniformi che si muove in una ragione fissata $\Omega$ dello spazio tridimensionale $\mathbb{R}^3$, sottoposto a una forza esterna per unità di massa $\mathbf{f}(\mathbf{x}, t)$. Sia informalmente $\nabla$ l'operatore nabla. Se $\mathbf{x} \in \Omega$, indichiamo le varie grandezze fisiche relative al fluido: con $\rho$ la densità, $\mathbf{u}(\mathbf{x}, t) \in \mathbb{R}^3$ la velocità vettoriale (solitamente l'incognita), $p$ la pressione, $\eta$ il coefficiente di viscosità (dinamica, di taglio). Per fluidi detti "newtoniani", vale la legge della viscosità di Newton, che può essere presa come definizione del coefficiente di viscosità: in un modello in cui un fluido ha flusso laminare in direzione $x_1$ (cioè, per ogni fissata quota $x_2$ perpendicolare a $x_1$, velocità costante lungo $x_1$), applicando una forza in direzione $x_2$ di modulo $\tau_{x_1 x_2}$ per unità di superficie, si ha $\tau_{x_1 x_2} = \eta \frac{du_{x_1}}{dx_2}$. Il movimento del fluido rispetto a un sistema di riferimento inerziale è descritto allora dalle "equazioni di Navier-Stokes" (come si può trovare spiegato in qualsiasi testo standard di Meccanica dei fluidi, ad esempio [18]):

$$\begin{cases} \rho(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}) = \eta \nabla^2 \mathbf{u} - \nabla p - \rho \mathbf{f} \\ \nabla \cdot \mathbf{u} = 0 \end{cases} \tag{4.1}$$

(osserviamo che $\mathbf{u} \cdot \nabla \mathbf{u} \equiv \sum_{i=1}^{3} u_i \frac{\partial \mathbf{u}}{\partial x_i}$). Si tratta evidentemente di un'equazione differenziale alle derivate parziali in $\mathbf{u}(t, x_1, x_2, x_3)$.
La seconda equazione, ricavabile dall'equazione di continuità della massa, esprime la condizione di fluidi incomprimibili. Qualora non sia questo il caso, la prima equazione può essere generalizzata a fluidi qualsiasi (non-newtoniani, comprimibili, con densità e viscosità variabili), meglio espressa in termini tensoriali anziché solo vettoriali-matriciali. Le equazioni di Navier-Stokes, ricavate nella prima metà dell'Ottocento, sono ad oggi considerate in questa loro forma più estesa (che qui non daremo) le più generali per la descrizione del moto di un fluido "classico" (vale a dire un corpo continuo senza forma propria). Questa descrizione è valida naturalmente solo a scale macroscopiche, dove l'ipotesi del continuo è applicabile, e non tiene conto di effetti quantistici o relativistici. Nonostante la loro estrema importanza anche dal punto di vista pratico, però, non è stato provato che ammettano soluzioni infinitamente differenziabili (o anche solo limitate; il che corrisponde a una soluzione "fisica" accettabile) in ogni punto del dominio, il che costituisce uno dei più importanti problemi matematici ad oggi ancora aperti.

## 4.2 Caso particolare: l'equazione di Burgers

Ci poniamo ora in un caso particolare dell'equazione di Navier-Stokes: formalmente, se la forza esterna (ad esempio la gravità) e le variazioni di pressione hanno effetti trascurabili rispetto all'inerzia e alla viscosità, ponendo i due ultimi termini dell'equazione di Navier-Stokes uguali a zero otteniamo in una dimensione ($x \in \mathbb{R}$) la cosiddetta "equazione di Burgers" (si veda [19], in particolare la sezione 4):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \mu\frac{\partial^2 u}{\partial x^2} \tag{4.2}$$

Dove abbiamo posto $\mu \equiv \frac{\eta}{\rho}$ (che viene anche detta "viscosità cinematica" ed è strettamente positiva per definizione). Osserviamo che nell'incognita $u = u(t, x)$ (la velocità del fluido nella direzione $x$, al tempo $t$) questa è un'equazione differenziale alle derivate parziali, di secondo ordine, quasi-lineare (considereremo in seguito lo specifico dominio e le condizioni al contorno), di evoluzione temporale (si veda il capitolo precedente).
Quest'equazione, introdotta per la prima volta ai primi del Novecento da H. Bateman, fu oggetto di studi più estesi da parte di J. M. Burgers (1948). Si tratta essenzialmente di un'equazione-prototipo, utile a evidenziare certe caratteristiche matematiche di alcune possibili soluzioni dell'equazione di Navier-Stokes. Ragionando in modo del tutto euristico, i due termini a sinistra rappresentano evidentemente un'onda che si propaga nel verso positivo dell'asse $x$ con velocità $u$. Se la viscosità fosse nulla (cosiddetta "equazione di Burgers inviscida") e la velocità decrescente lungo $x$, avremmo perciò (portando il secondo termine a destra dell'uguaglianza) che la variazione di velocità nel tempo, in un fissato punto dello spazio, sarebbe proporzionale al quadrato della velocità stessa (visto che $u\frac{\partial u}{\partial x} = \frac{1}{2}\frac{\partial(u^2)}{\partial x}$). Perciò regioni dell'onda inizialmente a velocità maggiore raggiungono regioni inizialmente più lente poste più avanti; questo porta al formarsi di "onde d'urto" (*shock waves*), matematicamente delle singolarità nel senso che il gradiente della velocità diverge in certi punti del dominio spaziale. Osserviamo anche che il termine a destra dell'equazione di Burgers completa (o "viscosa"), quello in cui compare la viscosità, ha segno opposto a quello in cui compare il gradiente (qualora $u$ diminuisca in modo più che lineare in $x$, che è evidentemente il nostro caso in un intorno delle singolarità); è quasi intuitivo allora che esso renda il sistema fisicamente realistico, impedendo che il gradiente della velocità aumenti in modo indefinito. Se $\mu$ è piccola, però, questa correzione sarà minima, e avremo delle regioni in cui il gradiente è molto ripido, pur essendo la soluzione analitica, il che costituisce spesso una sfida nel calcolo approssimato.

### 4.2.1 Soluzione analitica

Descriviamo ora brevemente come si possa ricavare, in modo relativamente semplice, una soluzione analitica dell'equazione di Burgers, con le proprietà di esistenza, unicità e analiticità. Ci rifacciamo a [15], in particolare alla sezione 4.4.1.
Partendo dall'equazione 4.2, effettuiamo un cambio di variabile detto "trasformazione di Cole-Hopf". Sia dunque $u =: \frac{\partial w}{\partial x}$, che sostituita nella 4.2 può essere scritta facilmente, previo scambio dell'ordine di derivazione del termine temporale, come $\frac{\partial[...]}{\partial x} = 0$; integrando quest'espressione otteniamo $[...] =: c(t) =: 0$ (questa costante sarebbe comunque del

tutto irrilevante in seguito essendo la derivazione rispetto $x$). A questo punto poniamo $f := \exp(-\frac{w}{2\mu})$ e sostituiamo in [...]; calcolando le derivate di $f$ rispetto a $t$ e $x$ (due volte), è facile vedere come [...] = 0 equivalga a:

$$\frac{\partial f}{\partial t} - \mu \frac{\partial^2 f}{\partial x^2} = 0 \tag{4.3}$$

Che è la nota "equazione del calore" (o "della diffusione", con coefficiente di diffusione $\frac{D}{2} = \mu$). Com'è noto, sulla retta reale essa ammette la "soluzione fondamentale" (per condizioni iniziali $f(0,x) = \delta(x)$):

$$f_{fond}(x,t) = \frac{1}{\sqrt{2\pi}\sqrt{2\mu t}} \exp\left(-\frac{x^2}{2 \cdot 2\mu t}\right) \equiv \mathcal{N}(0, 2\mu t) \tag{4.4}$$

(una distribuzione gaussiana con media 0 e varianza $2\mu t$). Data una generica condizione iniziale $f(0,x) = f_0(x)$, la soluzione dell'equazione del calore è invece data dalla convoluzione (operazione che indichiamo con $*$, come mostrato qui di seguito) di $f_0$ con $f_{fond}$:

$$f(x,t) = \frac{1}{\sqrt{4\pi\mu t}} \int_{-\infty}^{+\infty} f_0(x') \exp\left[-\frac{(x-x')^2}{4\mu t}\right] dx' \equiv f_0 * \mathcal{N}(0, 2\mu t) \tag{4.5}$$

Invertendo via via i cambi di variabile fatti all'inizio (l'inverso della funzione esponenziale è il logaritmo, della derivata è l'integrale; si noti però che l'estremo d'integrazione inferiore, arbitrario, va scelto in modo che la primitiva di $w$, cioè $u$, si annulli in quel punto) otteniamo facilmente la soluzione dell'equazione di Burgers.
Nel nostro caso, per uniformarci al problema risolto in [20] considereremo come dominio della PDE:

$$\begin{aligned} S &= ]-1, 1[ \\ T &= 1 \end{aligned} \tag{4.6}$$

con le condizioni al contorno:

$$\begin{aligned} u_0(x) &= -\sin(\pi x) \\ u_b(t,-1) &= u_b(t,+1) = 0 \end{aligned} \tag{4.7}$$

In questo caso particolare, procedendo come spiegato sopra, sceglieremo come estremo inferiore d'integrazione $x = -1/2$: visto che $u_0(-1/2) = 0$, avremo che $w(0,-1/2)$ è una costante, che si semplificherà all'ultimo passaggio. Scriveremo infine $u(x,t) = \frac{\partial}{\partial x}\left[-2\mu \ln(f(x,t))\right] = -2\mu \frac{1}{f}\frac{\partial f}{\partial x}$ (la costante $2\mu$ si semplifica in seguito, al momento di derivare il coseno dentro l'integrale). Così l'equazione di Burgers (4.2) sul dominio (4.6) e con condizioni al contorno (4.7) ha soluzione:

$$u(x,t) = \frac{-\mathcal{N}(0, 2\mu t) * \left[\sin(\pi x) \cdot \exp\left(-\frac{\cos(\pi x)}{2\mu}\right)\right]}{\mathcal{N}(0, 2\mu t) * \left[\exp\left(-\frac{\cos(\pi x)}{2\mu}\right)\right]} \tag{4.8}$$

(la convoluzione è un integrale su tutta la retta reale nella variabile $x$).
Si può anche dimostrare che, in maniera abbastanza intuitiva, essendo la soluzione dell'equazione del calore unica e infinitamente differenziabile (vedi [15], sezione 2.3), e dato che la trasformazione di Cole-Hopf è invertibile e preserva l'analiticità, anche la soluzione all'equazione di Burgers "viscosa" ha quelle buone proprietà.

### 4.2.2 Approssimazioni classiche

Approcci tradizionali per ottenere approssimazioni numeriche di PDEs come l'equazione di Burgers (si veda anche il relativamente datato [20]) sono ad esempio i cosiddetti "schemi alle differenze finite" (basati sulla discretizzazione del dominio spaziale, nella stessa logica di metodi Runge-Kutta; possono essere modificati tramite una "griglia" o *mesh* per dare minor peso ai punti ottenuti nei pressi delle singolarità, come nel caso dei cosiddetti *WENO*) e i "metodi spettrali" (fondati sulla decomposizione spettrale della funzione soluzione in un appropriata base). Essi risultano in realtà significativamente superiori (si veda ad esempio [21]) alle più recenti PINNs per la risoluzione diretta di PDEs come quella di Burgers su domini semplici, garantendo accuratezze estremamente elevate e tempi computazionali ridotti.
Il nostro scopo qui, comunque, rimane mostrare concretamente il funzionamento delle PINNs, per le quali come detto esiste tuttora un certo interesse nella comunità scientifica.

## 4.3 Soluzione dell'equazione di Burgers con un PINN

Consideriamo ora l'equazione di Burgers unidimensionale 4.2 sul dominio 4.6 con condizioni al contorno 4.7 e piccola viscosità $\mu = \frac{1}{100\pi}$. Vediamo la soluzione approssimata ottenuta con un'apposita *Physics-informed neural network*. Il programma da noi implementato, in *Pyhton*, si basa su [14] e nello specifico sul *Jupyter notebook* disponibile pubblicamente al link [22]. La nostra soluzione, che include solo alcune piccole aggiunte a quest'ultima, è invece disponibile nell'appendice A, oltre che a: `https://github.com/LiberoP/PINNsolverBurgersPDE/blob/main/burgers.ipynb`. Nel seguito descriviamo brevemente la logica nostro programma, seguendo l'ordine del codice.
Usiamo le librerie `Tensorflow` (per il *machine learning*), `Numpy`, `Matplotlib` (per i grafici) e `time` (per misurare il tempo di *training*).

### 4.3.1 Set di *training* e di *testing*

Dopo aver definito le funzioni $u_0(x)$, $u_b(t,x)$ e $r_\theta(t,x,\frac{\partial u}{\partial t},\frac{\partial u}{\partial x},\frac{\partial^2 u}{\partial x^2}$, consideriamo un set di training con $N_0 = 5000$, $N_b = 50$, $N_r = 10000$, con i punti $\xi$ scelti in maniera pseudo-casuale (secondo una distribuzione uniforme) grazie alle apposite funzionalità di `Tensorflow`.
Per il set di *testing*, vengono scelti nello stesso modo rispettivamente 1000, 10 e 2000. Il set di punti di *training* così ottenuto è mostrato nella figura 4.2.

### 4.3.2 Struttura della rete neurale utilizzata

Passiamo ora a definire il modello di rete neurale che useremo. Consideriamo essenzialmente una NN "profonda", composta da uno strato di input bidimensionale (dato che $u_\theta = u_\theta(t,x)$), seguito da uno strato di *scaling* che mappa separatamente, in modo lineare, il valore di ciascuno dei due neuroni di input nell'intervallo $[0,1]$. Questi strati sono seguiti da 8 strati "densi" di 20 neuroni ciascuno, con funzione di attivazione tanh e inizializzati tramite una Glorot normale. Abbiamo infine uno strato di output "denso" costituito da un

solo neurone (poiché $u_\theta$ è a valori reali).
Nel complesso abbiamo perciò (si veda anche la sezione 2.1.4): $[2 \cdot 20 + (20 \cdot 20) \cdot 7 + 20] + [20 \cdot 8 + 1] = 3021$ parametri allenabili (nella prima parentesi quadra abbiamo contato i *weights*, nella seconda i *bias* i quali sono presenti solo "dopo" ciascuno strato denso).

### 4.3.3 *Training* e *testing*

Per allenare il modello, usiamo l'ottimizzatore "Adam". Il gradiente di $u_\theta$ rispetto ai parametri allenabili $\boldsymbol{\theta}$, così come le derivate parziali che compaiono nel residuo $r_\theta$, vengono calcolati usando la differenziazione automatica implementata nella classe `GradientTape` di `Tensorflow` e il suo metodo `watch`. Viene quindi definita la funzione di perdita già descritta in precedenza. Viene infine definito il *learning rate*: costante a tratti, con $\eta_0 = 10^{-2}, \quad \eta_1 = 10^{-3}, \quad \eta_2 = 5 \cdot 10^{-4}$ per $T_0 = 1000, \quad T_1 = 3000$ e numero totale di epoche (fissato in seguito) $T_2 = 5000$.
Ad ogni epoca $T$ viene calcolata la funzione di perdita sul set di *training* (per successiva ottimizzazione) e su quello di *testing* (per verificare l'assenza di *overfitting*). Questi valori sono mostrati in funzione dell'epoca nella figura 4.3.

### 4.3.4 Risultati

Illustriamo ora brevemente i risultati da noi ottenuti.
Abbiamo allenato il nostro modello in poco più di 15 minuti, usando un semplice processore Intel Core i5-5300 a 2.30 GHz con 8.00 GB di RAM (senza GPU dedicata), un tempo tutto sommato accettabile. Già dal piccolo valore della funzione di perdita calcolato sul set di *testing* (errore di generalizzazione, circa $3.23 \cdot 10^{-4}$), solo di poco superiore a quello sul set di *training* (circa $2.03 \cdot 10^{-4}$), si evince come la soluzione trovata sia una buona approssimazione di quella reale. (Ricordiamo infatti che la funzione di perdita usata include già il residuo quadrato medio.) La funzione $u_\theta(t, x)$ (a valori reali) ottenuta, approssimazione della soluzione "vera" della PDE considerata, è mostrata in figura 4.4.

#### Possibili migliorie

Questi risultati potrebbero essere migliorati in vari modi, per avvicinarsi maggiormente a quelli di [14] (pagine 6-9). Innanzitutto, per ridurre i tempi di calcolo si potrebbe banalmente usare un hardware più performante. In secondo luogo, si potrebbe utilizzare la soluzione analitica 4.8 (approssimando con alta precisione gli integrali con un qualche metodo come quello "dei trapezi", che richiede comunque una discretizzazione del dominio, implementato di default in `Numpy`) per stimare l'errore relativo della soluzione ottenuta rispetto ad essa. In letteratura questo è espresso di solito come "errore relativo $\mathcal{L}^2$", intendendo con ciò il rapporto fra la norma $\mathcal{L}^2$ (vale a dire naturalmente la radice dell'integrale su tutto il dominio del modulo quadro della funzione) della differenza fra la soluzione ottenuta e quella vera e la norma $\mathcal{L}^2$ della soluzione vera. Questo consentirebbe un confronto più diretto fra i risultati ottenuti qui e quelli presenti in letteratura; ad esempio, lo stesso articolo riporta per l'architettura da noi usata un errore relativo $\mathcal{L}^2$ nell'ordine di $10^{-4}$. In terzo luogo, entrando più nello specifico del metodo usato, si potrebbero estrarre

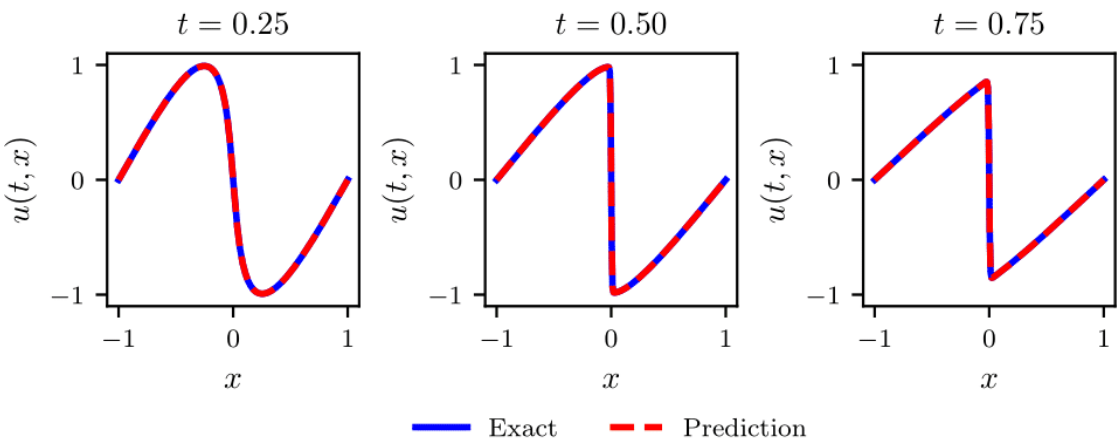


**Figura 4.1:** *Confronto, da [14], tra le soluzioni approssimata (tratteggiata in rosso) da una rete neurale, con architettura quasi identica alla nostra, ed esatta (in blu) dell'equazione di Burgers corrispondenti ai tre istanti temporali indicati. L'errore relativo* $\mathcal{L}^2$ *per questo caso è circa* $6.7 \times 10^{-4}$.

i punti di *training* non da una distribuzione uniforme, ma con il metodo detto di "*latin hypercube sampling*", che consiste essenzialmente nel costruire una griglia equispaziata (nel nostro caso semplicemente bidimensionale) ed estrarre all'incirca lo stesso numero di punti da ciascuna maglia (vedi anche [23]). Questo metodo aumenterebbe la velocità di convergenza, secondo quanto affermato sempre in [14]. Il confronto diretto fra la soluzione analitica e quella approssimata, calcolata dagli autori di [14] con metodologie leggermente più sofisticate rispetto alle nostre, è mostrato nella figura 4.1. In quarto e ultimo luogo, si potrebbero sperimentare architetture leggermente diverse per quanto riguarda funzione di attivazione, *kernel initializer*, *learning rate scheduling* etc.

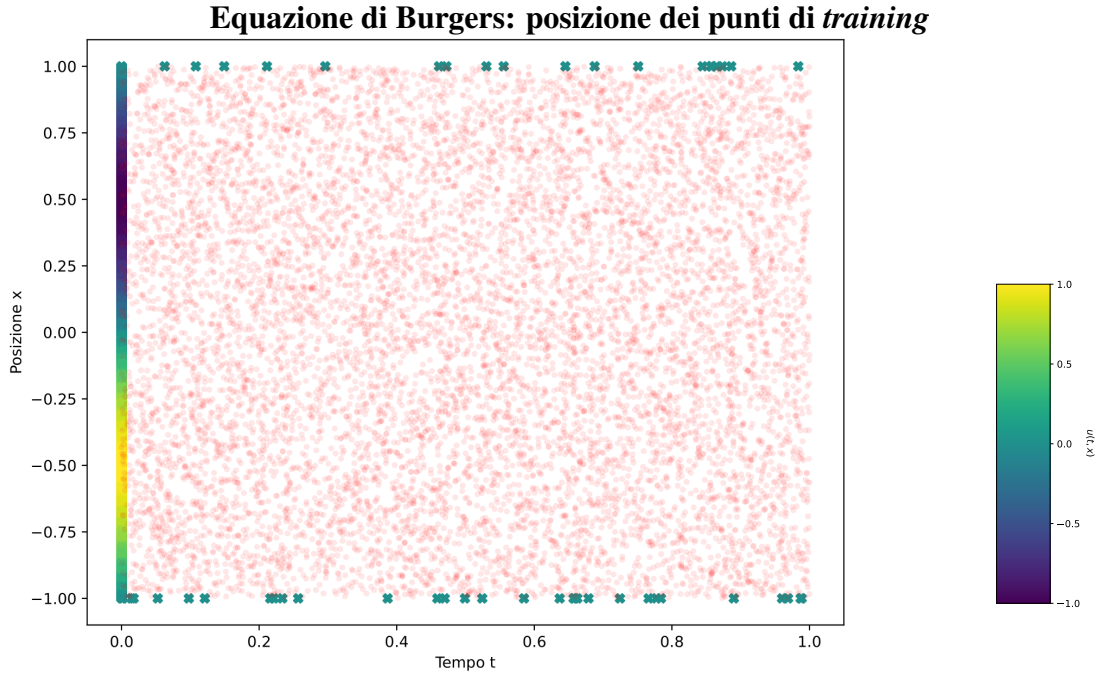


**Figura 4.2:** *Equazione di Burgers: posizione, nel dominio spazio-temporale della funzione $u(t,x)$, dei punti di training, scelti in modo casuale. In rosso i 10000 punti interni al dominio. Quelli corrispondenti alle condizioni al contorno $|x| = 1$ (50 punti, in cui la funzione è nulla) e $t = 0$ (5000 punti, in cui la funzione ha andamento sinusoidale) sono invece rappresentati con croci, il cui colore corrisponde al valore assunto della funzione in quel punto (a destra la legenda).*

**Equazione di Burgers: evoluzione della funzione di perdita nel corso delle iterazioni**

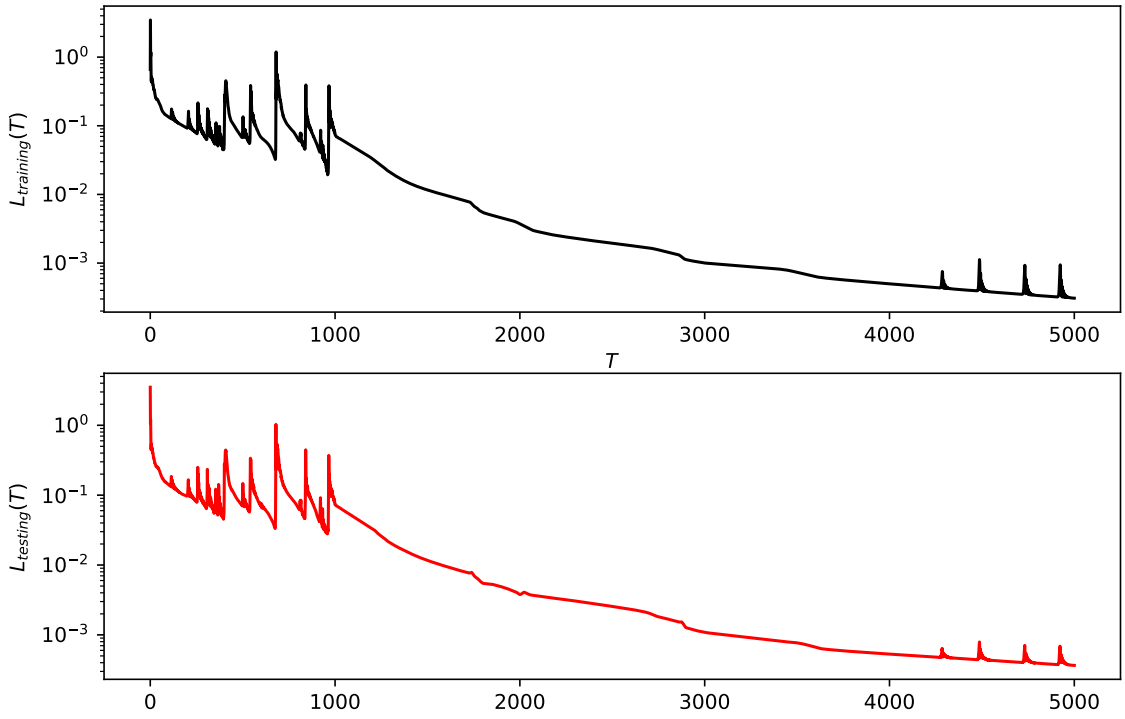


**Figura 4.3:** *Equazione di Burgers: valore in scala logaritmica della funzione di perdita L, calcolata rispettivamente su punti di training e di testing, in funzione dell'epoca T (5000 epoche totali, con learning rate costante a tratti che diminuisce a* $T = 1000$ *e a* $T = 3000$*). Si noti che l'andamento delle due è molto simile, indice dell'assenza di overfitting.*

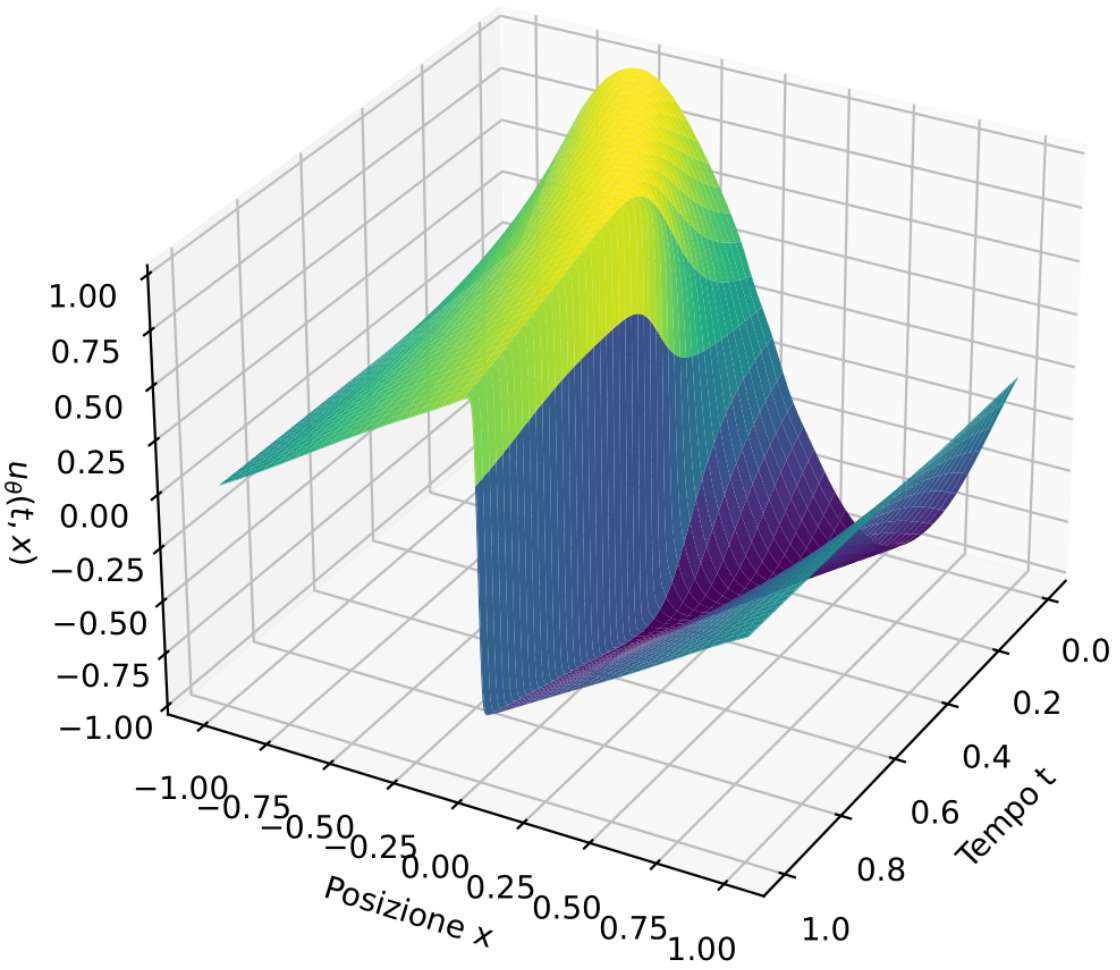


**Figura 4.4:** *Equazione di Burgers: la funzione soluzione ottenuta $u_\theta(t,x)$. Si vede chiaramente come al passare del tempo l'onda inizialmente sinusoidale diventi "a dente di sega", con il formarsi di un "fronte" nell'origine a partire da circa $t = 0.4$.*

# Conclusioni

In questo lavoro abbiamo illustrato il principio di funzionamento delle *Physics-informed neural networks* e la loro applicazione alla risoluzione dell'equazione di Burgers. I risultati ottenuti forniscono un chiaro esempio di come una rete neurale possa effettivamente apprendere, con relativa facilità, una soluzione approssimata che rispetta una data equazione differenziale alle derivate parziali con relative condizioni al contorno. Nella sezione precedente (4.3.4) abbiamo anche fornito alcune strategie che potrebbero essere messe in campo per migliorare ulteriormente la precisione della soluzione.

# Bibliografia


[1] Titus Neupert et al. *Introduction to Machine Learning for the Sciences*. 2022. arXiv: 2102.04883 [physics.comp-ph]. URL: https://arxiv.org/abs/2102.04883.

[2] Jan Blechschmidt e Oliver G. Ernst. *Three Ways to Solve Partial Differential Equations with Neural Networks – A Review*. 2021. arXiv: 2102.11802 [math.NA]. URL: https://arxiv.org/abs/2102.11802.

[3] François Chollet et al. *Keras*. https://github.com/keras-team/keras. 2015.

[4] João Manoel Herrera Pinheiro et al. *The Impact of Feature Scaling In Machine Learning: Effects on Regression and Classification Tasks*. 2025. arXiv: 2506.08274 [cs.LG]. URL: https://arxiv.org/abs/2506.08274.

[5] Christian Hennig e Mahmut Kutlukaya. «Some thoughts about the design of loss functions». In: *REVSTAT-Statistical Journal* 5.1 (2007), pp. 19–39.

[6] Sooyoung Jang e Youngsung Son. «Empirical Evaluation of Activation Functions and Kernel Initializers on Deep Reinforcement Learning». In: *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. 2019, pp. 1140–1142. DOI: 10.1109/ICTC46691.2019.8939854.

[7] Diederik P. Kingma e Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[8] Shai Shalev-Shwartz e Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[9] Claire Adam-Bourdarios et al. «The Higgs boson machine learning challenge». In: *NIPS 2014 workshop on high-energy physics and machine learning*. PMLR. 2015, pp. 19–55.

[10] Jorge De La Calleja e Olac Fuentes. «Machine learning and image analysis for morphological galaxy classification». In: *Monthly Notices of the Royal Astronomical Society* 349.1 (2004), pp. 87–93.

[11] John Jumper et al. «Highly accurate protein structure prediction with AlphaFold». In: *Nature* 596.7873 (2021), pp. 583–589.

[12] Alex Davies et al. «Advancing mathematics by guiding human intuition with AI». In: *Nature* 600.7887 (2021), pp. 70–74.

[13] Kurt Hornik, Maxwell Stinchcombe e Halbert White. «Multilayer feedforward networks are universal approximators». In: *Neural networks* 2.5 (1989), pp. 359–366.

[14] Maziar Raissi, Paris Perdikaris e George Em Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. 2017. arXiv: 1711.10561 [cs.AI]. URL: https://arxiv.org/abs/1711.10561.

[15] Lawrence C Evans. *Partial differential equations*. Vol. 19. American mathematical society, 2022.

[16] O Abu Arqub. «Series solution of fuzzy differential equations under strongly generalized differentiability». In: *Journal of Advanced Research in Applied Mathematics* 5.1 (2013), pp. 31–52.

[17] Pierfrancesco Beneventano et al. *High-dimensional approximation spaces of artificial neural networks and applications to partial differential equations*. 2025. arXiv: 2012.04326 [math.NA]. URL: https://arxiv.org/abs/2012.04326.

[18] L. D. Landau e E. M. Lifshitz. *Fluid Mechanics*. Vol. 6. Course of Theoretical Physics. Pergamon, 1987.

[19] G.B. Whitham. *Linear and nonlinear waves*. Wiley, 1974.

[20] C. Basdevant et al. «Spectral and finite difference solutions of the Burgers equation». In: *Computers & fluids* 14.1 (1986), pp. 23–41.

[21] Aditi Krishnapriyan et al. «Characterizing possible failure modes in physics-informed neural networks». In: *Advances in Neural Information Processing Systems*. A cura di M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 26548–26560. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/df438e5206f31600e6ae4af72f2725f1-Paper.pdf.

[22] Jan Blechschmidt. *Using physics informed neural networks (PINNs) to solve parabolic PDEs*. https://github.com/janblechschmidt/PDEsByNNs/blob/main/PINN_Solver.ipynb. [Online; consultato il 20-novembre-2025]. 2021.

[23] R. J Beckman, M. D. McKay e W. J. Conover. «A comparison of three methods for selecting values of input variables in the analysis of output from a computer code». In: *Technometrics* 21.2 (1979), pp. 239–245.

# A. Codice in Python usato per la *PINN* soluzione dell'equazione di Burgers

```python
import tensorflow as tf

tf.keras.backend.clear_session()
print("GPU available:", tf.config.list_physical_devices('GPU'))
# limit GPU memory growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    tf.config.experimental.set_memory_growth(gpus[0], True) # for accurate timing of training later

from time import time
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

DTYPE = "float32"
tf.keras.backend.set_floatx(DTYPE)
pi = tf.constant(np.pi, dtype=DTYPE)
viscosity = 0.01 / pi

# initial condition
def fun_u_0(x):
    return -tf.sin(pi * x)

# boundary condition
def fun_u_b(t, x):
    n = x.shape[0]
    return tf.zeros(
        (n, 1), dtype=DTYPE
    )  # n (as many as x) rows and 1 column of zeros because u(t, +-1) = 0

# residual of PDE
def fun_r(t, x, u, u_t, u_x, u_xx):
    return u_t + u * u_x - viscosity * u_xx

# number of data points
N_0 = 5000
N_b = 50
N_r = 10000

tmin = 0.0
tmax = 1.0
xmin = -1.0
xmax = 1.0

# lower and upper bounds
lb = tf.constant(
```

```
    [tmin, xmin], dtype=DTYPE
)  # creates constant tensor, here from python list
ub = tf.constant([tmax, xmax], dtype=DTYPE)

tf.random.set_seed(0)

# boundary, ie u(0, x) = -sin(pi*x) for random x's
t_0 = tf.ones((N_0, 1), dtype=DTYPE) * lb[0]
x_0 = tf.random.uniform((N_0, 1), lb[1], ub[1], dtype=DTYPE)
X_0 = tf.concat([t_0, x_0], axis=1)  # concatenate in a N_0 x 2 matrix

u_0 = fun_u_0(x_0)

# boundary, ie u(t, +-1) = 0 for random t's
t_b = tf.random.uniform((N_b, 1), lb[0], ub[0], dtype=DTYPE)
bernoulli_tensor = tf.cast(
    tf.less(tf.random.uniform([N_b, 1]), 0.5), dtype=DTYPE
)  # picks x = -1 or x = 1 with equal probabilty
x_b = lb[1] + (ub[1] - lb[1]) * bernoulli_tensor
X_b = tf.concat([t_b, x_b], axis=1)

u_b = fun_u_b(t_b, x_b)

# collocation points, ie u(t,x) for random t's and x's
t_r = tf.random.uniform((N_r, 1), lb[0], ub[0], dtype=DTYPE)
x_r = tf.random.uniform((N_r, 1), lb[1], ub[1], dtype=DTYPE)
X_r = tf.concat([t_r, x_r], axis=1)

X_data = [X_0, X_b]
u_data = [u_0, u_b]

fig = plt.figure(figsize=(9, 7))
plt.scatter(
    t_0, x_0, c=u_0, marker="X", vmin=-1, vmax=1
)  # color is a representation of values of the almost continuous
   # (1000s of data points) function u = -sin(pi*x) for t=0
plt.scatter(
    t_b, x_b, c=u_b, marker="X", vmin=-1, vmax=1
)  # color is uniform because u = 0 for x=+-1
plt.scatter(t_r, x_r, c="r", marker=".", alpha=0.1)
plt.xlabel("Tempo t")
plt.ylabel("Posizione x")
plt.title("Posizione dei punti di training")

plt.savefig('Xdata_burgers.pdf', bbox_inches='tight', dpi = 300)

# standalone colorbar matching the plot's colormap and range
fig, ax = plt.subplots(figsize=(1, 6))
norm = plt.Normalize(vmin=-1, vmax=1)
sm = plt.cm.ScalarMappable(cmap='viridis', norm=norm)
sm.set_array([])
cbar = plt.colorbar(sm, cax=ax)
cbar.set_label('$u(t,x)$', rotation=270, labelpad=15)
cbar.set_ticks([-1, -0.5, 0, 0.5, 1])
```

```
101
102 plt.savefig('colorbar_burgers.pdf', bbox_inches='tight', dpi=300)
103
104 def init_model(num_hidden_layers=8, num_neurons_per_layer=20):
105     model = tf.keras.Sequential()
106     model.add(tf.keras.Input(shape=(2,)))  # each input is of kind (t,x
    )
107     scaling_layer = tf.keras.layers.Lambda(
108         lambda x: 2.0 * (x - lb) / (ub - lb) - 1.0
109     )  # maps t and x linearly, from [0,1] and [-1,1], to [-1,1]
110     model.add(scaling_layer)
111
112     for i in range(num_hidden_layers):
113         model.add(
114             tf.keras.layers.Dense(
115                 num_neurons_per_layer,
116                 activation=tf.keras.activations.get("tanh"),
117                 kernel_initializer="glorot_normal",
118             )
119         )  # determines how weights are initialized (once) at the
    beginning of training.
120     # This one draws samples from a truncated normal distribution
    centered on 0 with stddev = sqrt(2 / (fan_in + fan_out))
121     # where fan_in is the number of input units in the weight tensor
    and fan_out is the number of output units in the weight tensor:
122     # few weights -> large variance is sensible.
123     model.add(tf.keras.layers.Dense(1))  # output layer
124
125     return model
126
127 def get_r(model, X_r):
128     with tf.GradientTape(
129         persistent=True
130     ) as tape:  # "persistent" is a boolean simply to check if the
    gradient has been created.
131         # same as:
132         # tape = tf.GradientTape(persistent=True)
133         # for i in tape:
134         # ...
135         t, x = X_r[:, 0:1], X_r[:, 1:2]  # not just 0 and 1 in order to
     keep dimension
136         tape.watch(t)  # gradient computed in t
137         tape.watch(x)
138         u = model(tf.stack([t[:, 0], x[:, 0]], axis=1))
139         u_x = tape.gradient(
140             u, x
141         )  # gradient of u with respect to x. In the with-as block
    because needed for second derivative because all gradient
    computations must happen inside
142     u_t = tape.gradient(u, t)
143     u_xx = tape.gradient(u_x, x)
144     del tape
145     return fun_r(
146         t, x, u, u_t, u_x, u_xx
```

```
    )  # residual of parametrized u, in sampled points

def compute_loss(model, X_r, X_data, u_data):
    r = get_r(model, X_r)
    phi_r = tf.reduce_mean(tf.square(r))  # technically tf.square(r-tf.zeros(...))
    loss = phi_r  # phi_r
    for i in range(len(X_data)):  # adds phi_b and phi_0, because X_data = [X_0, X_b]
        u_pred = model(X_data[i])
        loss += tf.reduce_mean(tf.square(u_data[i] - u_pred))
    return loss

def get_grad(
    model, X_r, X_data, u_data
):  # "usual" gradient with respect to trainable parameters
    with tf.GradientTape(persistent=True) as tape:
        loss = compute_loss(model, X_r, X_data, u_data)
    g = tape.gradient(loss, model.trainable_variables)
    del tape
    return loss, g

model = init_model()
lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay(
    [1000, 3000], [1e-2, 1e-3, 5e-4]
)
optim = tf.keras.optimizers.Adam(learning_rate=lr)
# tf.executing_eagerly()

#TESTING BEGINS HERE
# number of data points
N_0_t = 1000
N_b_t = 10
N_r_t = 2000

# boundary, ie u(0, x) = -sin(pi*x) for random x's
t_0_t = tf.ones((N_0_t, 1), dtype=DTYPE) * lb[0]
x_0_t = tf.random.uniform((N_0_t, 1), lb[1], ub[1], dtype=DTYPE)
X_0_t = tf.concat([t_0_t, x_0_t], axis=1)  # concatenate in a N_0 x 2 matrix

u_0_t = fun_u_0(x_0_t)

# boundary, ie u(t, +-1) = 0 for random t's
t_b_t = tf.random.uniform((N_b_t, 1), lb[0], ub[0], dtype=DTYPE)
bernoulli_tensor_t = tf.cast(
    tf.less(tf.random.uniform([N_b_t, 1]), 0.5), dtype=DTYPE
)  # picks x = -1 or x = 1 with equal probabilty

x_b_t = lb[1] + (ub[1] - lb[1]) * bernoulli_tensor_t
X_b_t = tf.concat([t_b_t, x_b_t], axis=1)

u_b_t = fun_u_b(t_b_t, x_b_t)

```

```
# collocation points, ie u(t,x) for random t's and x's
t_r_t = tf.random.uniform((N_r_t, 1), lb[0], ub[0], dtype=DTYPE)
x_r_t = tf.random.uniform((N_r_t, 1), lb[1], ub[1], dtype=DTYPE)
X_r_t = tf.concat([t_r_t, x_r_t], axis=1)

X_data_t = [X_0_t, X_b_t]
u_data_t = [u_0_t, u_b_t]

fig = plt.figure(figsize=(9, 7))
plt.scatter(
    t_0_t, x_0_t, c=u_0_t, marker="X", vmin=-1, vmax=1
)  # color is a representation of values of the almost continuous
#  (1000s of data points) function u = -sin(pi*x) for t=0
plt.scatter(
    t_b_t, x_b_t, c=u_b_t, marker="X", vmin=-1, vmax=1
)  # color is uniform because u = 0 for x=+-1
plt.scatter(t_r_t, x_r_t, c="r", marker=".", alpha=0.1)
plt.xlabel("Tempo t")
plt.ylabel("Posizione x")
plt.title("Posizione dei punti di testing")
plt.savefig('Xdata_burgers_t.pdf', bbox_inches='tight', dpi = 300)

def compute_loss_t(model, X_r_t, X_data_t, u_data_t): # same but for
    testing
    r_t = get_r(model, X_r_t)
    phi_r_t = tf.reduce_mean(tf.square(r_t))  # technically tf.square(r
    -tf.zeros(...))
    loss_t = phi_r_t  # phi_r
    for i in range(len(X_data_t)):  # adds phi_b and phi_0, because
    X_data = [X_0, X_b]
        u_pred_t = model(X_data_t[i])
        loss_t += tf.reduce_mean(tf.square(u_data_t[i] - u_pred_t))
    return loss_t

if __name__ == "__main__": # for more efficient timing

    @tf.function  # python decorator: converts following function to
    tensorflow function for efficiency
    def train_step():
        loss, grad_theta = get_grad(model, X_r, X_data, u_data)
        optim.apply_gradients(
            grads_and_vars=zip(grad_theta, model.trainable_variables)
        )  # takes a list of gradients and trainable variables and
    returns a tf.Variable, representing the current iteration.
        return loss

    N = 5000
    hist = []
    hist_t = []
    t0 = time()
    for i in range(N + 1):
        loss = train_step()
        loss_t = compute_loss_t(model, X_r_t, X_data_t, u_data_t)
        hist.append(loss.numpy())
```

```
247         hist_t.append(loss_t.numpy())
248 
249         if i % 50 == 0:
250             print(
251                 f"Iteration {i}: training loss = {loss}"
252             )  # prints loss function value every 50 iterations
253 
254         if i % 500 == 0:
255             print(
256                 f"Iteration {i}: validation error = {loss_t}"
257             )
258 
259     print(f"Computation time: {time()-t0} s")
260 
261 N = 600  # number of grid marks
262 tspace = np.linspace(lb[0], ub[0], N + 1)
263 xspace = np.linspace(lb[1], ub[1], N + 1)
264 T, X = np.meshgrid(tspace, xspace)
265 # note: flattened X has 600 elements = -1, then 600 elements =
    -0.99666...,
266 # while T goes from 0 to 1 for 600 times; opposite behavior if indexing
    ='ij'.
267 stack = np.vstack([T.flatten(), X.flatten()])  # first row is T, second
     is X
268 Xgrid = stack.T  # transposes stack
269 upred = model(tf.cast(Xgrid, DTYPE))  # float type conversion
270 
271 U = upred.numpy().reshape(N + 1, N + 1)
272 
273 fig = plt.figure(figsize=(9, 6))
274 ax = fig.add_subplot(
275     111, projection="3d"
276 )  # place subplot in position 1-1-1; with custom projection '3d'
277 ax.plot_surface(T, X, U, cmap="viridis")
278 # type of color map (default is in one tone)
279 ax.view_init(29, 29)  # pov, in degress up and to the side (rotated
    clockwise)
280 ax.set_xlabel("Tempo t")
281 ax.set_ylabel("Posizione x")
282 ax.set_zlabel("$u_\\theta(t,x)$")
283 ax.set_title("Soluzione dell'equazione di Burgers")
284 
285 plt.savefig('Burgers_Solution.pdf', bbox_inches = 'tight', pad_inches
    =0.5, dpi = 300)
286 
287 fig = plt.figure(figsize=(9, 6))
288 ax1 = fig.add_subplot(211)
289 ax1.semilogy(
290     range(len(hist)), hist, "k-"
291 )  # k is for black marker, semilogy is analog of plot
292 ax2 = fig.add_subplot(212, sharex=ax1)
293 ax2.semilogy(range(len(hist_t)), hist_t, 'r-')
294 
295 ax1.set_xlabel("$T$")
```

```
296 ax1.set_ylabel("$L_{training} (T)$")
297 ax2.set_ylabel("$L_{testing} (T)$")
298 plt.savefig('loss_evolution.pdf', bbox_inches='tight', dpi = 300)
```