POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Chimica e dei Processi Sostenibili

Tesi di Laurea Magistrale

Accoppiamento di CFD e *machine learning* e applicazione su trasporto in letti impaccati

Relatori

Prof. Gianluca BOCCARDO

Prof. Daniele MARCHISIO

Dott.ssa Nunzia LAURIELLO

Candidato

Domenico PAOLILLI

Luglio 2025

Πάντα χωρεῖ καὶ οὐδὲν μένει.
Tutto si muove, nulla resta fermo.

Platone, Cratilo (402a)

Sommario

La crescente digitalizzazione e automatizzazione dei diversi ambiti dell'ingegneria riveste un ruolo sempre più centrale nell'ottimizzazione dei processi e dei modelli ad essa associati. In questo contesto, l'intelligenza artificiale (AI) si afferma come uno strumento strategico, attraverso le sue applicazioni nelle branche del machine learning e soprattutto del deep learning. Obiettivo di questo lavoro di tesi è esplorare come l'ingegneria chimica, con specifico riferimento alla modellazione multiscala e alla fluidodinamica computazionale (CFD), possa trarre vantaggio dall'impiego di tecnologie basate sull'intelligenza artificiale. L'intento è quello di contribuire a una progressiva evoluzione degli approcci tradizionali, aprendo la strada a nuove prospettive metodologiche. Nel caso specifico analizzato, l'attenzione è rivolta all'applicazione di queste tecniche all'analisi di simulazioni numeriche di flusso e trasporto in letti impaccati, elementi centrali in numerosi processi chimici e operazioni unitarie. Tra questi si annoverano, ad esempio, il processo Fischer-Tropsch, la sintesi di ammoniaca, di idrogeno e di metanolo, considerati veri e propri *building blocks* dell'industria chimica moderna. La necessità di rivedere gli approcci convenzionali nasce dal fatto che, nonostante il continuo progresso nelle tecnologie hardware e software, spesso le simulazioni CFD risultano ancora oggi dispendiose in termini di tempo e risorse computazionali. A tale scopo, l'utilizzo di modelli surrogati basati su machine learning e deep learning rappresenta una soluzione promettente. Tali modelli, opportunamente allenati su dataset contenenti informazioni sia sulla geometria del sistema che su range ampi di condizioni operative, consentono una predizione dei risultati quasi istantanea. Infatti, grazie alla loro natura flessibile, essi possono essere riaddestrati per affrontare nuovi scenari mai esplorati in precedenza. Nel dettaglio, la metodologia adottata prevede l'impiego di reti neurali, sia *fully-connected* che convolutive. Quest'ultime, in particolare, consentono di trattare la geometria porosa direttamente come immagine, superando la necessità di una caratterizzazione morfologica esplicita e offrendo così un approccio più flessibile e comprensivo. Gli strumenti computazionali utilizzati comprendono software sia per la simulazione CFD (`OpenFOAM`), sia per lo sviluppo di modelli di machine learning, inizialmente tramite le librerie `Keras/TensorFlow` e successivamente con `PyTorch`. Inoltre, il software `Blender` è stato utilizzato per

la generazione di un dataset di geometrie di morfologie e distribuzioni diverse, sfruttando le sue funzionalità di scripting in `Python` che ne hanno permesso l'automazione. A complemento di questo processo, è stata impiegata la libreria `Trimesh` in `Python` per la conversione dei file generati in `Blender` in geometrie numeriche discrete (voxel). Le simulazioni effettuate hanno riguardato il trasporto di quantità di moto e di materia, con l'obiettivo di caratterizzare i mezzi porosi in termini di proprietà quali permeabilità ed efficienza di filtrazione, applicate a geometrie di varia morfologia.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

In un mondo in cui la digitalizzazione ed automatizzazione dei processi ingegneristici occupa un ruolo sempre più rilevante, l'integrazione fra modelli numerici tradizionali propri della modellazione di processo e della **fluidodinamica computazionale** (**CFD**) con tecniche innovative, afferenti all'ambito della cosiddetta **AI** (intelligenza artificiale) rappresenta una delle sfide emergenti del nuovo millennio. È qui la chiave della cosiddetta transizione digitale, il processo evolutivo che mira alla trasformazione di sistemi e modelli operativi e decisionali attraverso la sistematica implementazione di tecnologie avanzate.

Nel contesto più specifico di questo lavoro di tesi, il ruolo dell'intelligenza artificiale è assunto da modelli di tipo **machine learning** e **deep learning**, in grado di fornire il giusto compromesso fra robustezza e adattabilità, garantendo al tempo stesso prestazioni idonee all'analisi e all'interpretazione di dati complessi, quali sono quelli fluidodinamici.

La scelta della struttura CFD da analizzare è ricaduta sui **mezzi porosi**, spesso caratterizzati da una complessa morfologia e numerose sfide legate alla modellazione multi-scala. Risulta, difatti, non banale raccordare la descrizione dalla microscala alla macroscala (Figura 1.1). Sono strutture caratterizzate dall'alternarsi di vuoti - *voids*, che rappresentano la fase porosa - e di solido - porzione del mezzo che ne identifica la geometria, anche riferito come *matrice*. [1] Sono identificati da parametri caratteristici, quali, ad esempio, la porosità ϵ (rapporto volume vuoto/totale), la permeabilità k (capacità del mezzo di farsi attraversare da una fase fluida sotto l'azione di un gradiente di pressione) e la tortuosità τ (rapporto tra la lunghezza effettiva del percorso seguito dal fluido attraverso i pori e la lunghezza geometrica del mezzo nella direzione del flusso): queste grandezze risultano esclusivamente funzione della geometria del mezzo poroso e non dipendono dalle condizioni di processo in cui questo si trova ad operare. Possono avere varie configurazioni, da un quasi-regolare impaccamento di sfere a frammenti irregolari e non immediatamente modellabili, fino alle schiume (*foams*).

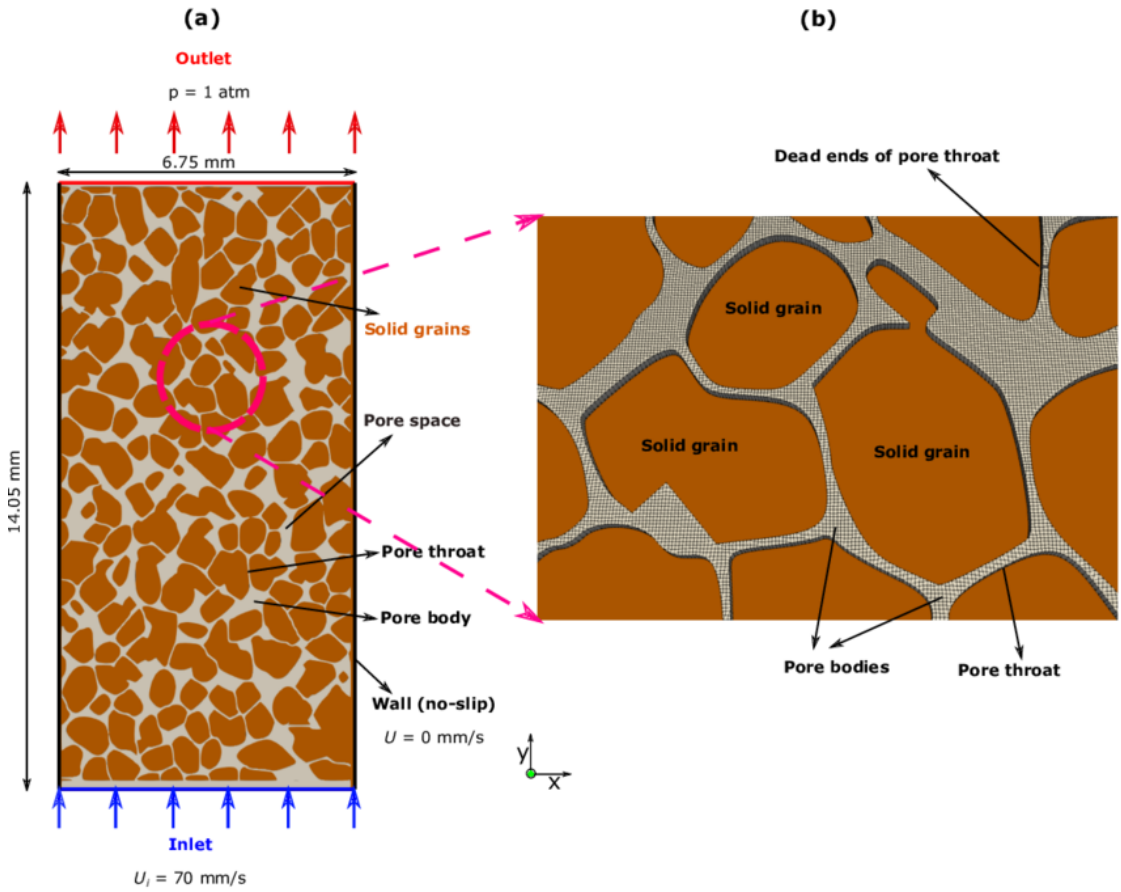


Figura 1.1: Letto poroso impaccato, con raffigurazione in macroscala (a) e microscala (b). Fonte: [2]

I mezzi porosi, siano essi di genesi naturale o artificiale, rappresentano poi una grande risorsa per una vasta gamma di applicazioni ingegneristiche, fra le quali:

- **catalisi**: i mezzi porosi vengono frequentemente impiegati come supporti per la deposizione di specie catalitiche attive (quali nichel, palladio, platino, ecc.), grazie alla loro elevata area superficiale specifica e alla struttura interna ramificata. Queste caratteristiche favoriscono meccanismi di trasporto sia superficiale che intragranulare, ottimizzando la diffusione dei reagenti verso i siti attivi e migliorando l'efficienza dei processi catalitici eterogenei;

- **reattori packed-bed** (Figura 1.2): rappresentano il cuore strutturale dell'unità reattoristica. Oltre ad offrire una vasta superficie specifica per reazioni fondanti nell'ingegneria chimica (quali, ad esempio, sintesi di idrogeno e *syngas*,

processo Fischer-Tropsch, produzione di ammoniaca e metanolo), sono di essenziale importanza in processi di assorbimento e desorbimento, favorendo, grazie ad un'elevata area superficiale e bagnabilità, il trasferimento di materia fra le due fasi. La loro morfologia influenza in modo significativo la distribuzione del flusso, la caduta di pressione, la diffusione interna e l'efficienza complessiva del trasporto di massa (ed in minor parte, di calore) all'interno del reattore;

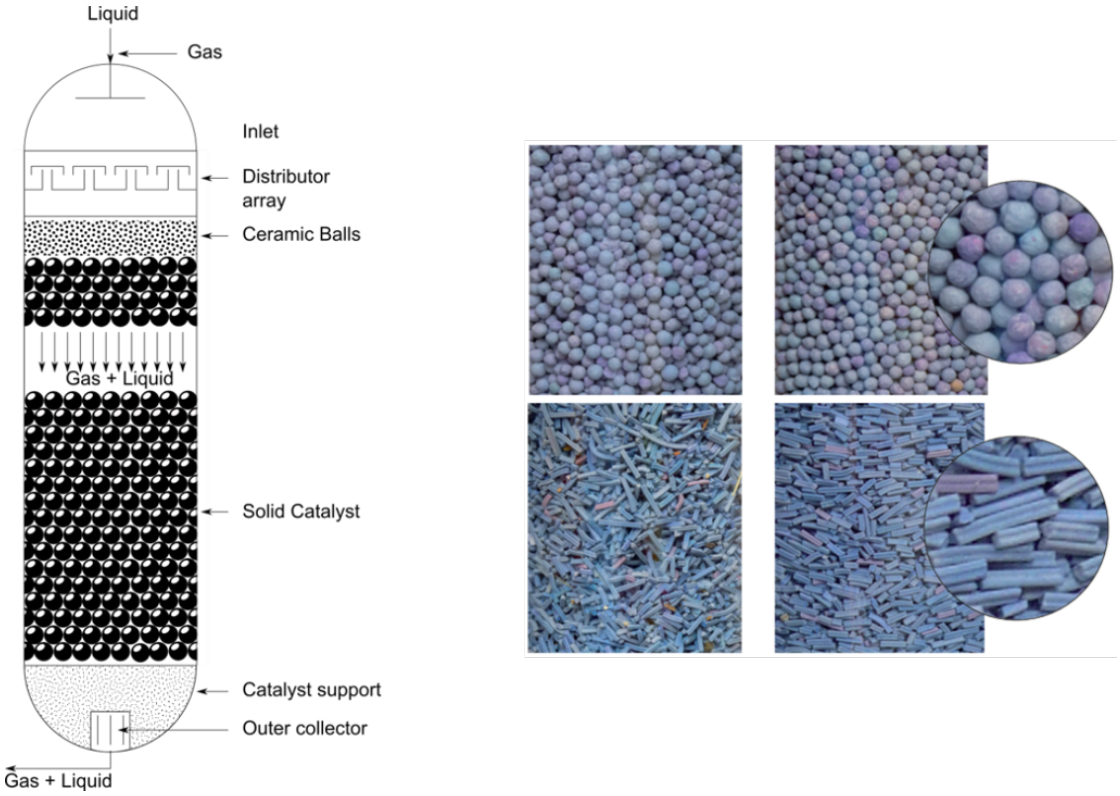


Figura 1.2: Colonna a letto impaccato, a geometria sferica e trilobata. Fonte: [3]

- **separazione e filtrazione**: trovano impiego in numerose tecniche di separazione fisico-chimica, tra cui la filtrazione meccanica (microfiltrazione e ultrafiltrazione), l'adsorbimento selettivo su superfici funzionalizzate e i processi a membrana (ad esempio, osmosi inversa e pervaporazione). L'efficacia di questi processi è determinata da un parametro chiave, il *cut-off*, che corrisponde alla dimensione massima delle particelle o molecole che possono attraversare la membrana. Questo valore è strettamente legato alla conformazione porosa e alla distribuzione dimensionale dei pori del materiale. Un'applicazione recente e di grande rilevanza è quella dei dispositivi di protezione individuale, come le mascherine facciali, che sfruttano mezzi porosi a struttura multistrato per filtrare particelle, aerosol e patogeni nell'aria, combinando meccanismi di

intercettazione meccanica e diffusione intra-poro. Altre applicazioni tipiche, in ambito ambientale, includono il trattamento delle acque reflue (biofiltri, ad esempio *trickle-bed*), la purificazione di gas industriali e la rimozione selettiva di contaminanti organici ed inorganici;

- **energy storage**: sono largamente utilizzati come mezzi di accumulo energetico. In ambito elettrochimico, ad esempio, in elettrodi di dispositivi quali batterie, supercondensatori, celle a combustibile e sensori. La loro struttura porosa offre un'elevata area superficiale attiva, favorendo il trasferimento ionico e migliorando la cinetica delle reazioni associate. In ambito di accumulo termico (**thermal energy storage**), sono ad esempio impiegati per incapsulare materiali *phase-change* (PCM, Figura 1.3) [4], una classe di composti che ha l'obiettivo di assorbire o rilasciare calore in forma latente, permettendo così un accumulo energetico a temperature costanti - come accade nel caso del *passive cooling*;

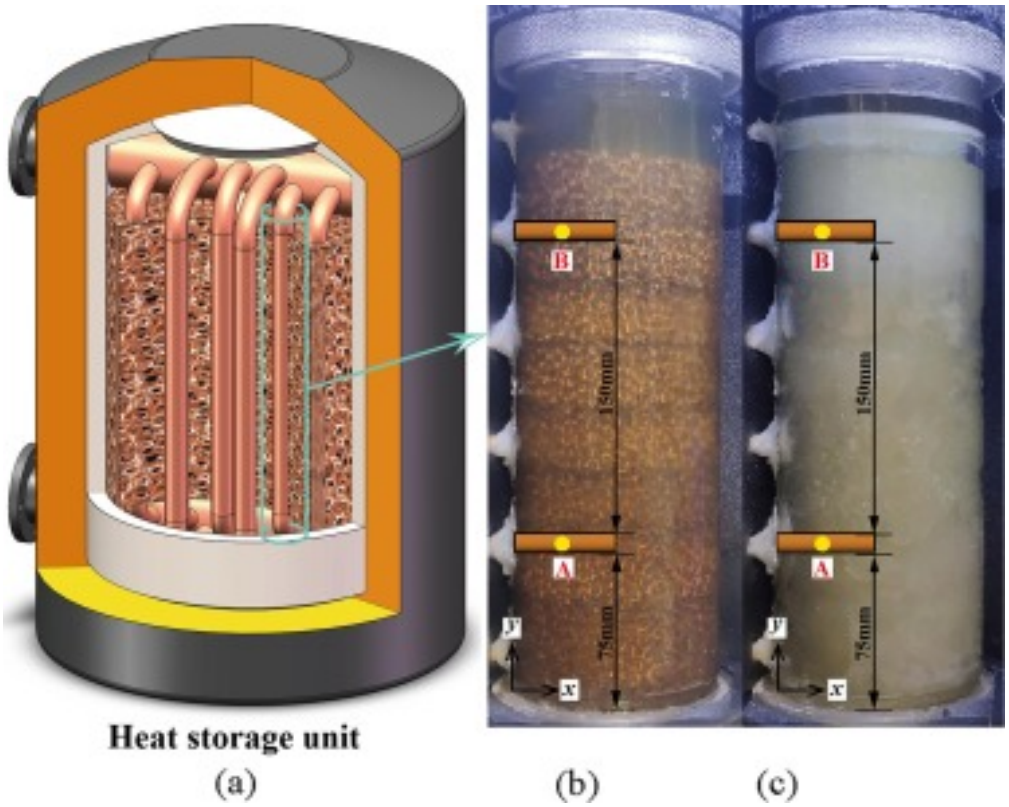


Figura 1.3: Schiuma metallica per applicazione PCM, con schema del reattore (a), reattore reale (b) e reattore reale in operazione con PCM di natura paraffinica (c). Fonte: [4]

- **ingegneria idrogeologica**: la descrizione e caratterizzazione dei mezzi porosi risulta particolarmente utile nell'analisi del suolo e delle sue proprietà meccaniche e idrauliche, consentendo di prevedere il comportamento del flusso di acqua sotterranea, la percolazione dei contaminanti e la stabilità dei terreni.

La necessità di accoppiare la descrizione fluidodinamica di questi meccanismi a modelli surrogati e *data-driven* di tipo *machine learning* è una scelta motivata da ragioni di tipo prettamente pratico: le simulazioni CFD sono molto spesso costose in termini di tempo e di risorse computazionali impiegate, risultando in processi complessi e risultati non attingibili nell'immediato. Ciononostante, la presenza di modelli appositamente allenati per qualificare determinate caratteristiche - di natura fluidodinamica o chimica - rende lo stadio di inferenza delle stesse molto più agevole ed immediato, con una predizione quasi istantanea. Rapportando questa motivazione teorica ad un risvolto prettamente pratico, riscontrato in questo percorso di ricerca, si è osservato che:

- le simulazioni CFD, in esecuzione su un solo *core*, impiegano circa 24 ore ciascuna per essere svolte *in toto*;
- una parallelizzazione di queste simulazioni CFD fino a 4 *core* comporta una diminuzione del tempo di calcolo fino a raggiungere le 7 ore complessive per ciascuna;
- un modello *machine learning* (in questo ambito, **rete neurale**), nel caso di input solamente numerici e struttura FC, impiega al più 5 minuti, se si esclude il processo di *tuning* (che richiede comunque tempi modesti); nel caso in cui si intenda far apprendere visivamente la geometria alla rete (approccio convolutivo), questo step può richiedere fino ad un massimo di 8 ore.

Da questi dati, è possibile notare come un modello surrogato possa sopperire alla complessità computazionale delle simulazioni CFD, offrendo una valida alternativa predittiva una volta completata la fase di addestramento. In particolare, l'impiego di modelli di tipo machine learning consente di ottenere, in tempi estremamente ridotti, stime attendibili di parametri fluidodinamici o chimico-fisici di interesse, rendendo possibile un'esplorazione più ampia dello spazio delle configurazioni geometriche e operative. Questo si traduce in un vantaggio pratico concreto: mentre la CFD resta fondamentale per la generazione di dati ad alta fedeltà e per la validazione dei modelli - che pur è necessaria per avere una raccolta di dati quanto più vasta ed accurata, l'utilizzo di *surrogate models* permette di effettuare analisi parametriche ed ottimizzazioni in modo molto più efficiente. In altre parole, questo modello diventa un'estensione strategica della simulazione numerica classica, in grado di fornire inferenze rapide e ripetibili con un costo computazionale trascurabile rispetto alla simulazione diretta.

Nel contesto applicativo di questa tesi, si è voluto analizzare un caso di filtrazione attraverso un mezzo poroso. Questo processo di deposizione (nel contesto delle assunzioni presentate in seguito, nel Capitolo 2) può essere considerato analogo ad un processo superficiale reattivo con cinetica estremamente veloce, tuttavia durante le simulazioni fluidodinamiche non è stato modellato esplicitamente mediante una legge cinetica di reazione ma tramite opportune condizioni al contorno. In particolare, la specie da rimuovere presente nel fluido alimentato al mezzo è un colloide - casistica tutt'altro che inusuale nel contesto ingegneristico, ad esempio nel trattamento dei reflui o nell'uso di metalli colloidali (ad esempio, ferro zerovalente) per la bonifica di acquiferi contaminati.

Per quanto concerne la strutturazione di questo lavoro, si è pensato ad un flusso che possa partire da nozioni ed enunciati di base della fluidodinamica (Capitolo 2) e del *machine/deep learning* (Capitolo 3), passando poi in rassegna i metodi computazionali implementati nella costruzione del set di simulazioni (Capitolo 4). Il nucleo del lavoro di tesi propriamente inteso è dato dal Capitolo 5 - che descrive gli elementi pratici e sequenziali nell'impostazione e svolgimento delle simulazioni CFD, oltre ad un'analisi rispettivamente ai parametri caratteristici delle geometrie porose impiegate - e del Capitolo 6, che entra nel vivo della descrizione e intende presentare i risultati ottenuti dall'implementazione di strategie *data-based*. *A posteriori*, si è inserito un capitolo di Conclusioni (Capitolo 7) con una visione ed elaborazione globale dei principali risultati ottenuti, assieme a prospettive e possibilità future, e l'Appendice, contenente gli script di lavoro più importanti di questo percorso.

Capitolo 2

Elementi teorici di modellazione fluidodinamica

Per procedere alla descrizione e alla trattazione delle proprietà di trasporto in mezzi porosi, argomento di questo lavoro di tesi, è necessario descrivere le formulazioni fisico-matematiche alla base delle simulazioni stesse, e su cui preliminarmente sono operate assunzioni e semplificazioni. I fenomeni analizzati vertono su meccanismi di **trasporto di quantità di moto** e **trasporto di materia**.

2.1 Assunzioni

Le assunzioni sottese allo sviluppo di questi modelli sono le seguenti, e sono tratte da quanto rilevato da Boccardo *et al.* (2014) [5]:

1. il fluido è assunto **incomprimibile**, con densità costante;

2. il fluido è assunto **continuo**, in quanto il cammino libero medio delle molecole che lo compongono è significativamente più basso dell'ordine di grandezza dei pori del mezzo;

3. le particelle di colloide sono anch'esse assunte come fase continua, con dimensioni caratteristica di decine di nanometri;

4. il mezzo poroso è saturo, ergo è presente una sola fase fluida;

5. il mezzo poroso è non reattivo, ma esclusivamente coinvolto in un processo di deposizione superficiale;

6. la concentrazione di colloide è sufficientemente bassa da poter assumere il sistema diluito e descrivibile mediante la legge diffusiva di Fick:

$$\mathbf{J}_A = -D_{AB}\nabla c_A \tag{2.1}$$

che, nel caso unidimensionale (flusso lungo una sola componente spaziale x), può essere scritto come:

$$J_{Ax} = -D_{AB}\frac{dc_A}{dx} \tag{2.2}$$

dove $\mathbf{J}_A$ è il flusso molare di A [mol m^{-2} s^{-1}], D_{AB} è il coefficiente di diffusione dela specie A nella specie B [m^2 s^{-1}], ∇c_A è il gradiente di concentrazione molare di A [mol m^{-4}] e c_A la concentrazione di A [mol m^{-3}] - A è la specie colloidale, B la fase acquosa.

7. il fluido è considerato **newtoniano**, facendo seguito all'ipotesi di diluizione della specie colloidale;

8. le particelle colloidali seguono il senso di flusso del fluido in cui sono disperse. Tale ipotesi è effettivamente realistica per una dispersione con dimensione caratteristica inferiore alla scala micronica e densità compresa fra 5000 - 10000 kg m^{-3}, in condizioni normali (NTP). In altri termini, le forze viscose prevalgono su quelle inerziali;

9. si trascura ogni effetto dovuto al doppio strato elettrico che possa alterare il trasporto del fluido e dei colloidi stessi mediante sovrapposizione di un campo di forze di natura elettrica. Inoltre, si rende trascurabile anche il contributo di altre forze esterne, come quella gravitazionale.

2.2 Trasporto di quantità di moto

Avendo a modello le seguenti equazioni fondamentali per descrivere il moto del fluido e il trasporto dela fase colloidale, rispettivamente di **continuità** (2.3) e di **Navier-Stokes** (2.4) [[7], [8]]:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(\rho v_i) = 0 \tag{2.3}$$

$$\rho\left(\frac{\partial v_i}{\partial t} + v_j\frac{\partial v_i}{\partial x_j}\right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left[\mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial v_k}{\partial x_k}\right)\right] + \rho g_i \tag{2.4}$$

sotto le ipotesi specificate, esse subiscono notevoli semplificazioni.

Per l'**equazione di continuità**, che esprime la conservazione della massa del fluido, l'assunzione di **incomprimibilità (ipotesi 1)** - ovvero densità (ρ) del

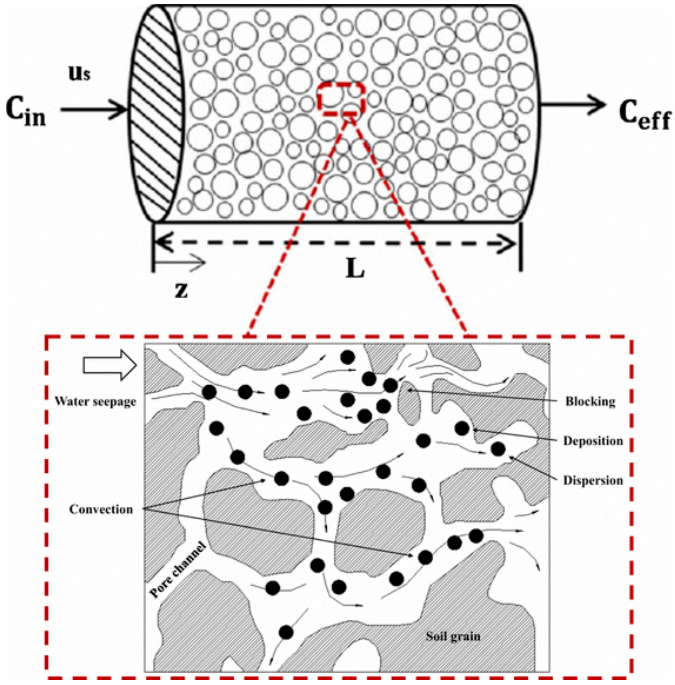


Figura 2.1: Meccanismo di trasporto e deposizione all'interno di un mezzo poroso. Fonte: [6]

fluido costante - implica che la variazione della densità nel tempo e nello spazio è nulla ($\frac{\partial \rho}{\partial t} = 0$). Di conseguenza, l'equazione si riduce a:

$$\frac{\partial v_i}{\partial x_i} = 0 \tag{2.5}$$

dove v_i rappresenta la componente i-esima del vettore velocità del fluido e x_i la coordinata spaziale corrispondente. Questa equazione stabilisce che il volume di un elemento di fluido rimane costante nel tempo.

Per ottenere una forma semplificata dell'**equazione di Navier-Stokes**, che descriva un flusso unidimensionale nella direzione x_j, considerando la variazione della componente i della velocità (v_i) in quella direzione, e assumendo un fluido incomprimibile con viscosità cinematica $\nu = \mu/\rho$ (**ipotesi 1** e **ipotesi 7**), le variazioni della velocità nelle altre direzioni spaziali sono considerate trascurabili. Trascurando inoltre le forze esterne (**ipotesi 9**), l'equazione di Navier-Stokes si riduce a:

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 v_i}{\partial x_j^2} \tag{2.6}$$

dove v_i è la componente della velocità nella direzione i, t è il tempo, v_j è la componente della velocità nella direzione j (che è la direzione principale del flusso).

- $\frac{\partial v_i}{\partial t}$: rappresenta la **variazione locale della velocità** (variazione della velocità in un punto fisso nel tempo).
- $v_j \frac{\partial v_i}{\partial x_j}$: rappresenta il **termine convettivo** (variazione della velocità dovuta al movimento del fluido nello spazio).
- $-\frac{1}{\rho}\frac{\partial p}{\partial x_i}$: rappresenta il **contributo del gradiente di pressione** (forza dovuta alla variazione di pressione nella direzione i).
- $\nu \frac{\partial^2 v_i}{\partial x_j^2}$: rappresenta il **contributo della diffusione viscosa nella direzione del flusso** (forza dovuta all'attrito interno del fluido).

2.3 Trasporto di materia

L'equazione caratteristica che descrive il trasporto della specie colloidale reca come scalare di riferimento la concentrazione di quest'ultima nel mezzo poroso. Essa viene anche definita come **equazione di advezione-diffusione**:

$$\frac{\partial C}{\partial t} + v_j \frac{\partial C}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\mathcal{D}\frac{\partial C}{\partial x_j}\right) \tag{2.7}$$

dove:

- C: rappresenta la **concentrazione del colloide** [mol/m^3];
- v_j: rappresenta la **componente della velocità del fluido** nella direzione j [m/s]. Si assume che le particelle colloidali seguano il senso di flusso del fluido **(ipotesi 8)**;
- $\mathcal{D}$: rappresenta il **coefficiente di diffusione del colloide** [m^2/s], considerato costante nello spazio e indipendente dalla distanza particella - pellet del mezzo poroso.

I contributi all'equazione di trasporto del colloide sono:

- **termine advettivo (o convettivo):** $v_j \frac{\partial C}{\partial x_j}$ rappresenta il trasporto della concentrazione del colloide dovuto al movimento macroscopico del fluido nella direzione x_j;
- **termine diffusivo:** $\frac{\partial}{\partial x_j}\left(\mathcal{D}\frac{\partial C}{\partial x_j}\right)$ rappresenta il trasporto della concentrazione del colloide dovuto al moto browniano delle particelle, che tende a uniformarne la concentrazione spaziale. Le particelle si muovono da regioni ad alta concentrazione verso regioni a bassa concentrazione. Poiché $\mathcal{D}$ è costante, questo termine si semplifica a $\mathcal{D}\frac{\partial^2 C}{\partial x_j^2}$.

In particolare, il coefficiente di diffusione $\mathcal{D}$ può essere stimato mediante la legge di Einstein, secondo cui:

$$\mathcal{D} = \frac{k_B T}{3\pi\mu d_c} \tag{2.8}$$

dove k_B è la costante di Boltzmann (1.38×10^{-23} J/K), T è la temperatura assoluta del fluido e d_c è il diametro della particella colloidale, assunta sferica.

2.4 Proprietà di interesse e altri parametri

Le proprietà che si intende predire attraverso questo lavoro di tesi sono essenzialmente due, e coinvolgono i due ambiti di trasporto poco fa analizzati: la **permeabilità** e il **filtration rate** di un mezzo poroso.

2.4.1 Permeabilità

È una proprietà intrinseca di un mezzo poroso, che quantifica la sua capacità di essere attraversato da un fluido sotto l'azione di un gradiente di pressione. È calcolabile mediante l'**equazione di Darcy generalizzata**, o **legge di Forchheimer**, che descrive il flusso di fluidi attraverso mezzi porosi:

$$-\nabla p = \frac{\mu}{k} q + \beta\rho|q|q \tag{2.9}$$

dove q è la velocità superficiale del fluido (detta di Darcy), k la permeabilità del mezzo (assunta costante nello spazio) [m^2], ∇p è il gradiente di pressione [Pa/m], ρ è la densità del fluido [kg/m^3], e β è il coefficiente di inerzia che rappresenta la resistenza non lineare al flusso [m^{-1}].

In regime di flusso laminare, le forze viscose prevalgono ampiamente sulle forze inerziali, rendendo trascurabile il termine non lineare nell'equazione. L'equazione di Darcy si riduce dunque alla sua forma più semplice:

$$q = -\frac{k}{\mu}\nabla p \tag{2.10}$$

Questa forma lineare descrive un flusso guidato unicamente dalla differenza di pressione attraverso il mezzo. La permeabilità risulta tuttavia dipendente esclusivamente dalla morfologia dell'impaccamento ed indipendente dalle condizioni operative.

2.4.2 Filtration rate

È la velocità caratteristica che descrive la proprietà del mezzo poroso di favorire la deposizione (o adsorbire) la specie colloidale permettendo, al contempo,

il passaggio della fase fluida attraverso le porosità. La sua derivazione completa è effettuata a partire da una media volumica rispetto all'equazione (2.7) - per un ricavo dettagliato della relazione che segue, si rimanda a Boccardo *et al.* (2018) [9]:

$$K_f = \frac{F_{tot}^{in} - F_{tot}^{out}}{\langle C \rangle V}$$

dove:

- K_f: **filtration rate**, anche visualizzabile come velocità caratteristica del processo di filtrazione della specie colloidale attraverso il mezzo poroso, [1/s];
- F_{tot}^{in} e F_{tot}^{out}: **flusso totale** rispettivamente **entrante ed uscente della specie colloidale** (contributo convettivo e diffusivo), [mol/s];.
- $\langle C \rangle$: **concentrazione media volumetrica della specie** nel mezzo, [mol/m^3];
- V: **volume totale del mezzo poroso**, [m^3].

L'equazione suggerisce che K_f è proporzionale alla differenza netta tra il flusso entrante e uscente della specie, normalizzata per la quantità totale di specie presente nel sistema.

Un K_f elevato è sinonimo di una forte variazione di concentrazione fra ingresso ed uscita del fluido, che si traduce in un processo diffusivo e filtrativo molto efficiente, rispetto ad una differenza ingresso - uscita più modesta. Questa considerazione tiene conto del fatto che le simulazioni sono effettuate in stato stazionario e, per l'**ipotesi 5**, non vi sia alcun processo reattivo.

2.4.3 Numeri adimensionali di riferimento

Nel corso della trattazione, verranno introdotti i numeri adimensionali che seguono al fine di quantificare e qualificare la prevalenza di alcuni meccanismi e caratteristiche, di cui è rilevante analizzare l'andamento e l'entità - cfr. Marcato *et al.* (2022) [10]. Questi sono illustrati nella Tabella 2.1, assieme ad una descrizione della loro espressione analitica e relativo significato fisico:

Tabella 2.1: Numeri adimensionali rilevanti per il trasporto in mezzi porosi.

Numero	Simbolo	Definizione	Significato
Reynolds	Re	$\frac{\rho q d_p}{\mu} = \frac{q d_p}{\nu}$	In questo contesto, è basato sulla velocità superficiale - o darcyana (q), una dimensione caratteristica dei grani (d_g), e le proprietà del fluido (densità ρ, viscosità dinamica μ, viscosità cinematica ν), indica il rapporto tra le forze inerziali e le forze viscose nel flusso del fluido attraverso i pori formati dall'impaccamento dei grani.
Schmidt	Sc	$\frac{\nu}{\mathcal{D}}$	Rappresenta il rapporto tra la diffusività cinematica (ν) del fluido e la diffusività di massa ($\mathcal{D}$) della specie colloidale dispersa nel fluido. Un $Sc \gg 1$ indica che la diffusione di massa è lenta rispetto alla diffusione di quantità di moto, mentre un $Sc \ll 1$ indica il contrario. Questo numero è essenziale per comprendere la formazione di strati limite di concentrazione e l'efficacia del trasporto diffusivo.
Péclet	Pe	$\frac{qL}{\mathcal{D}} = Re \cdot Sc$	Esprime il rapporto tra il trasporto convettivo (dovuto al flusso del fluido con velocità superficiale q su una scala di lunghezza L - dimensione del mezzo nel senso di scorrimento del fluido) e il trasporto diffusivo (con coefficiente di diffusione $\mathcal{D}$). È fondamentale per distinguere regimi di trasporto dominati dall'avvezione ($Pe \gg 1$) da regimi dominati dalla diffusione ($Pe \ll 1$). Un alto Pe implica che le particelle di colloide vengono trasportate principalmente dal flusso del fluido, mentre un basso Pe indica che la diffusione gioca un ruolo predominante nella loro distribuzione.
Damköhler	Da	$\frac{K_f L}{q}$	Confronta la velocità di filtrazione (tramite costante K_f, filtration rate) con la velocità di trasporto (caratterizzata da una velocità superficiale q su una scala di lunghezza L). È cruciale per analizzare i processi di deposizione e interazione delle particelle con le superfici del mezzo poroso. Un $Da \gg 1$ indica che la filtrazione è veloce rispetto al tempo di trasporto attraverso il poro, suggerendo che quest'ultimo è lo stadio limitante. Al contrario, un $Da \ll 1$ indica che la filtrazione è lenta rispetto al trasporto, dunque diventa il meccanismo limitante.

Capitolo 3

Machine learning, *deep learning* e reti neurali

Il termine **machine learning** si riferisce al processo attraverso il quale i sistemi informatici apprendono dai dati per eseguire compiti specifici. Diversamente dalla programmazione tradizionale, dove le istruzioni per l'esecuzione di un compito sono definite esplicitamente, il machine learning si basa sull'apprendimento di pattern e relazioni direttamente dai dati. Mediante algoritmi altamente dedicati, le macchine analizzano *dataset* per estrarre caratteristiche (o *features*) significative e identificare andamenti. L'obiettivo principale è costruire modelli predittivi o decisionali efficaci, capaci di generalizzare la conoscenza appresa a nuovi dati - per tale ragione, definiti anche **data-driven models**. Questo apprendimento può avvenire in diverse modalità, come nel caso del *clustering* (*unsupervised learning*: learning da dati etichettati - input e risposta corretta; l'obiettivo è associare queste label a nuovi dati), della *classificazione* e della *regressione* (*supervised learning*: learning da dati non etichettati - inferenza di pattern nascosti e complessi), o tramite regole di *associazione*.

Il **deep learning** è una branca specialistica del machine learning, il cui fine è inferire dei pattern complessi (a partire da immagini, testi, suoni, etc.) mediante l'utilizzo di architetture a struttura profonda (*multi-layer structures*, *deep* per l'appunto), anche note come **reti neurali**, in grado di cogliere la natura gerarchica delle feature instrinseche per mezzo di processi di apprendimento *supervised*, *unsupervised* o *semi-supervised*.

3.1 Introduzione alle reti neurali artificiali (ANN)

Le **reti neurali** [[11], [12]](anche conosciute come **Artificial Neural Networks - ANN**) sono algoritmi di machine learning il cui principio cardine è emulare il funzionamento del sistema nervoso umano, composto da cellule neuronali, deputate alla trasmissione degli impulsi nervosi tramite legami sinaptici. Allo stesso modo, un neurone artificiale sarà soggetto ad un flusso di informazioni tale per cui l'apprendimento - o **training** - avviene mediante un rapporto causa-effetto (in altre parole, un flusso input-output).

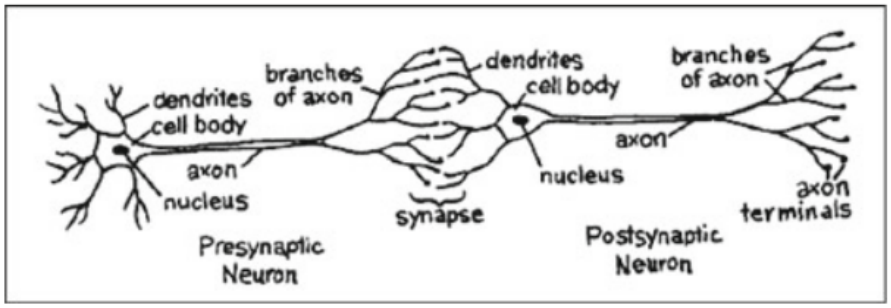


Figura 3.1: Struttura di un neurone umano. Fonte: [11]

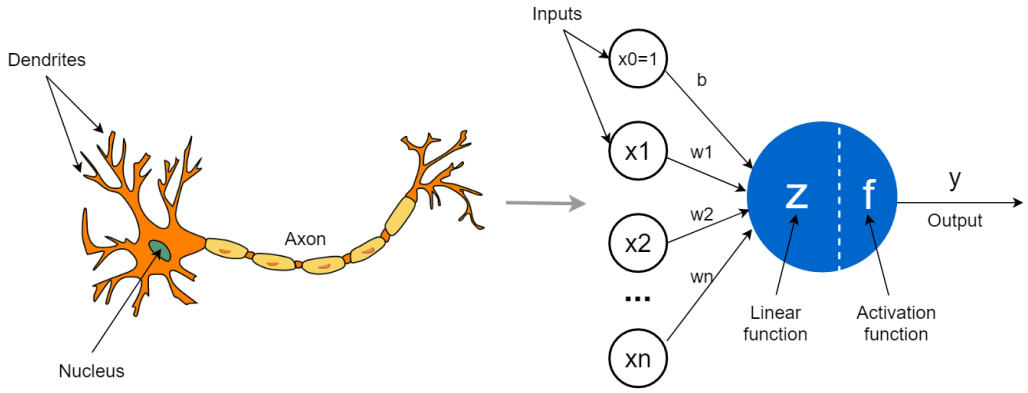


Figura 3.2: Confronto fra neurone umano e neurone artificiale. Fonte: [13]

Questo neurone ha una funzione essenziale di building block della rete, ed è anche riferito come **perceptrone**. Reca un singolo layer di **N** nodi destinati a ricevere il dato in input ($\bar{X} = [\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N]$), che viene immesso in una funzione lineare, data dal prodotto con i pesi associati ai rispettivi nodi ($\bar{W} = [\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_N]$), la

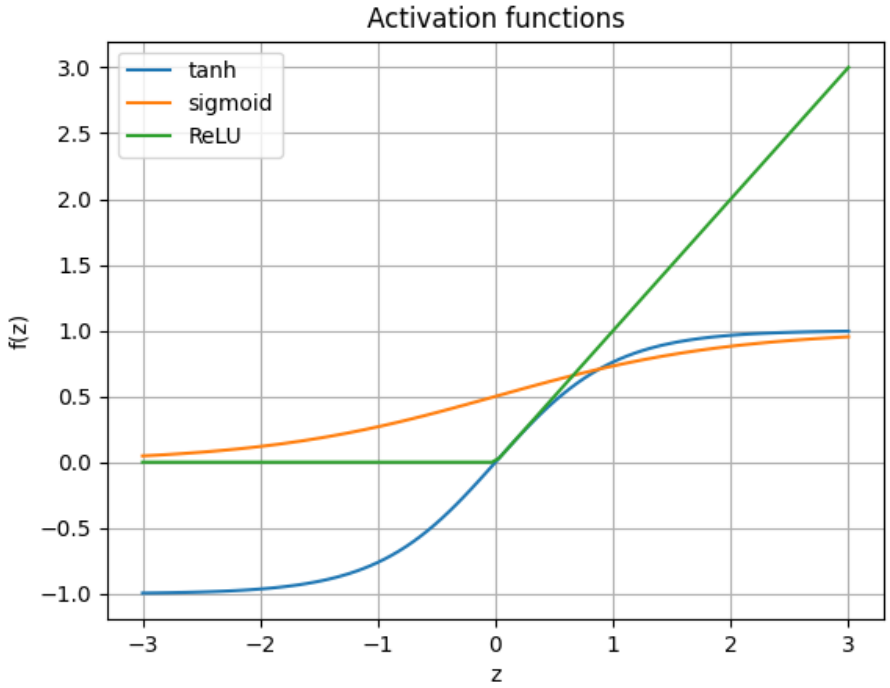


Figura 3.3: Funzioni di attivazione.

quale, dopo applicazione di una funzione di nonlinearità f, risulta nella predizione $\hat{y}$:

$$\hat{y} = f(\sum_{i=1}^{N} w_i x_i + b) \tag{3.1}$$

In particolare:

- la **funzione di attivazione** f (Figura 3.3) ha il ruolo di introdurre una non linearità rispetto alla **combinazione lineare** di somma pesata $z_i = \sum_{i=1}^{N} w_i x_i + b$, tale da permettere alla rete di modellare pattern complessi e spesso non lineari, quali sono quelli dei dati reali, incrementandone la capacità di apprendimento. Tra le funzioni di attivazione più usate, spiccano:
  - la funzione **sigmoide**, anche detta **logistica**: $\sigma(z_i) = \frac{1}{1+e^{-z_i}}$,
  - la funzione **tangente iperbolica**: $tanh(z_i) = \frac{e^{z_i}-e^{-z_i}}{e^{z_i}+e^{-z_i}}$,
  - la **Rectified Linear Unit** $(ReLU)$: $f(z_i) = max(0, z_i)$.

- il **bias** costituisce, al pari dei pesi, un parametro apprendibile della rete. La sua funzione è quella di introdurre una traslazione nell'attivazione, garantendo un output non nullo qualora tutti i valori di input (x_i) siano nulli.

La struttura tipica di una rete neurale ad utilizzo in ambito deep learning è quella mostrata in Figura 3.4, ed è dovuta alla caratteristica profondità (*depth*)

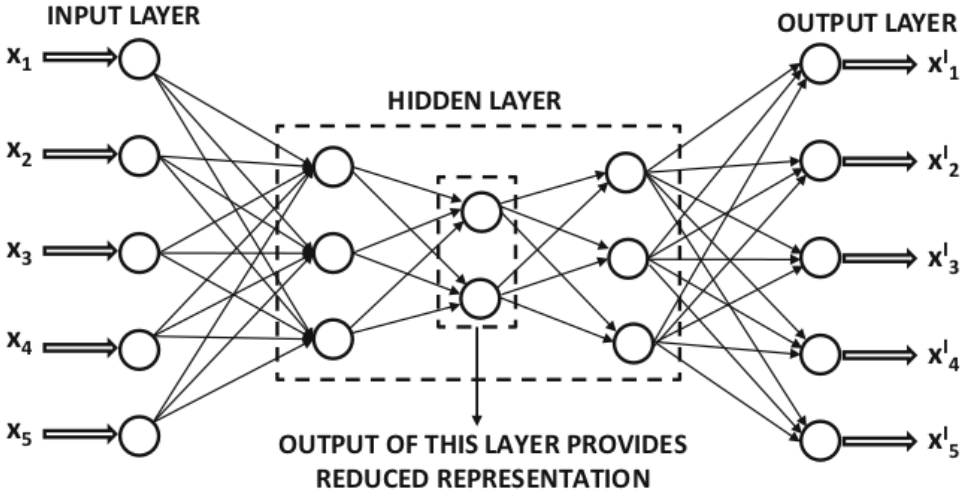


Figura 3.4: Architettura di una rete neurale per deep learning. Fonte: [11]

generata dalla presenza di un numero variabile ed elevato di strati nascosti (*hidden layers*) compresi fra l'input e l'output.

Questa architettura profonda permette alla rete di apprendere rappresentazioni gerarchiche dei dati, dove ogni strato successivo estrae feature di livello sempre più astratto. Ad esempio, nel contesto del riconoscimento di immagini, il primo strato potrebbe imparare a rilevare bordi e angoli, gli strati intermedi forme e texture, e l'ultimo strato oggetti complessi.

L'apprendimento della rete si realizza ottimizzando pesi e bias, ossia i parametri che definiscono la trasformazione lineare in ogni neurone. Questo processo è guidato dalla minimizzazione di una **funzione di perdita** (*loss function*), o **funzione di costo**, che quantifica l'errore tra l'output predetto e quello reale (Figura 3.5). La minimizzazione di questo avviene tramite l'algoritmo di **backward propagation** (o **backpropagation**), che calcola il gradiente della funzione di perdita rispetto ai pesi. Questo gradiente guida l'aggiornamento dei pesi per ridurre l'errore.

L'influenza dell'assegnazione randomica iniziale dei pesi sul processo di apprendimento ed inferenza sarà analizzata nei capitoli successivi.

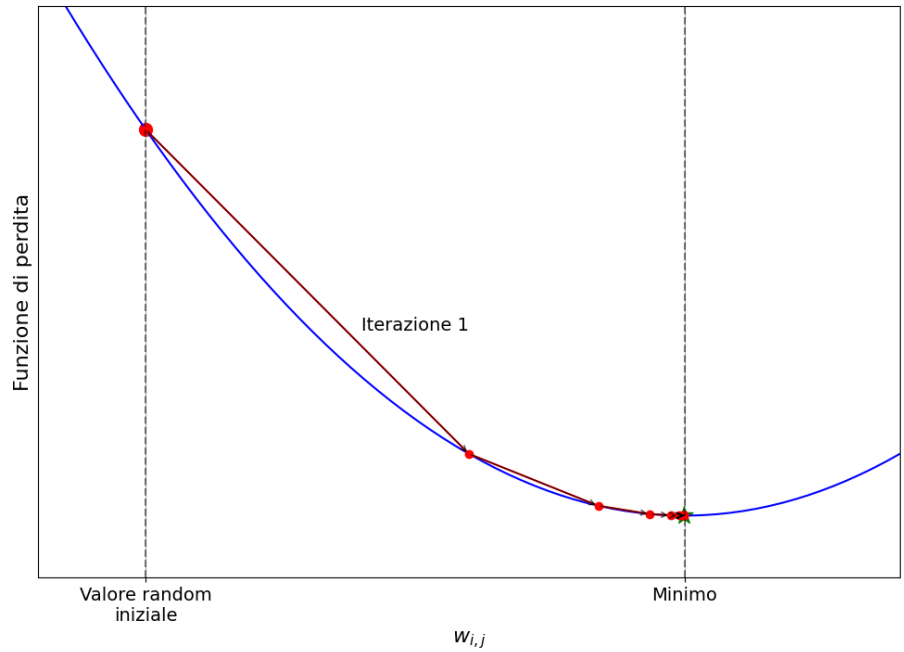


Figura 3.5: Minimizzazione della funzione di perdita mediante aggiornamento dei pesi associati ai neuroni.

3.2 Concetti e parametri di valutazione

La scelta dell'architettura (numero di strati, numero di neuroni per strato, tipo di connessioni) e degli iperparametri (tasso di apprendimento, funzione di attivazione, ottimizzatore) è cruciale per il successo dell'addestramento e dipende dal tipo di problema da risolvere e dalla natura dei dati. Si analizzano in seguito le definizioni più importanti al riguardo:

- **learning rate**: determina quanto velocemente la rete apprende dai dati di training, ossia la frequenza di aggiornamento dei pesi sulla base di un processo di ottimizzazione. Matematicamente, con w_t peso al tempo t, viene definito come $\alpha = \frac{w_t - w_{t+1}}{\nabla L(w_t)}$, con $L(w_t)$ funzione di perdita rispetto al peso w_t;
- **loss function**: è una funzione atta a quantificare l'errore fra il valore predetto $(\hat{y}_i)$ e quello reale (y_i). Fra le più comuni:
- **ottimizzatore**: è la funzione che regola l'aggiornamento dei pesi sulla base della funziona di perdita, con l'obiettivo di trovare configurazioni che massimizzino le prestazioni del modello.

Tabella 3.1: Loss functions comuni (N numero totale di campioni).

Nome	Espressione matematica
Mean Squared Error (MSE)	$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
Mean Absolute Error (MAE)	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i - \hat{y}_i\rvert$
Binary Cross-Entropy	$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$

Tabella 3.2: Ottimizzatori comuni.

Nome	Funzione
Gradient Descent (GD)	$w_{t+1} = w_t - \alpha\nabla L(w_t)$
Stochastic Gradient Descent (SGD)	$w_{t+1} = w_t - \alpha\nabla L(w_t; x_i)$
Momentum	$v_{t+1} = \beta v_t + \alpha\nabla L(w_t)$
	$w_{t+1} = w_t - v_{t+1}$
RMSprop	$v_t = \beta v_{t-1} + (1-\beta)(\nabla L(w_t))^2$
	$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t+\epsilon}}\nabla L(w_t)$
Adam	$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(w_t)$
	$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w_t))^2$
	$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
	$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
	$w_{t+1} = w_t - \frac{\alpha\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$

Il processo di apprendimento ed inferenza si compone di tre fasi, che rispecchiano la suddivisione del dataset alla rete stessa.

1. **training** (o **addestramento**): in questo step, il modello apprende dai dati in input, instaurando dei pattern e delle relazioni nella direzione della riduzione dell'errore di perdita (*training loss*);

2. **validation** (o **validazione**): valuta la performance del modello durante la fase di training, ed è particolarmente utile a prevenire fenomeni di sovradattamento ai dati (overfitting). La *validation loss* è tipicamente il parametro che, più della *training loss*, governa la bontà del modello stesso;

3. **test** (o **inferenza**): è lo stadio ultimo del processo, e valuta le prestazioni del modello appena addestrato su nuovi dati mai analizzati dalla rete stessa, verosimilmente al suo caso di applicazione reale.

Tipicamente, il rapporto rispettivo fra i tre set in termini percentuali risulta essere di 60-70% per il train, seguto dalla validation per il 15-20%, e test 10-15%.

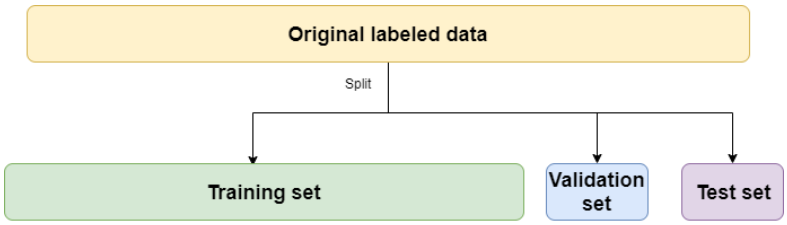


Figura 3.6: Suddivisione del dataset.

3.3 Tipologie di ANN

Nell'ambito machine e deep learning, si sono affermati nel corso del tempo svariati modelli ed intepretazioni di connessioni neurali:

1. **Feedforward Neural Network (FFNN)**: anche detta **Fully Connected Neural Network (FCNN)**, è la rete neurale più immediata, con un flusso di dati che è indirizzato univocamente dall'input all'output, attraverso gli strati nascosti intermedi. Ciascun neurone di ogni layer è collegato con tutti i neuroni degli strati adiacenti. Il loro uso è vantaggioso quando i dati non recano una struttura sequenziale o spaziale specifica.

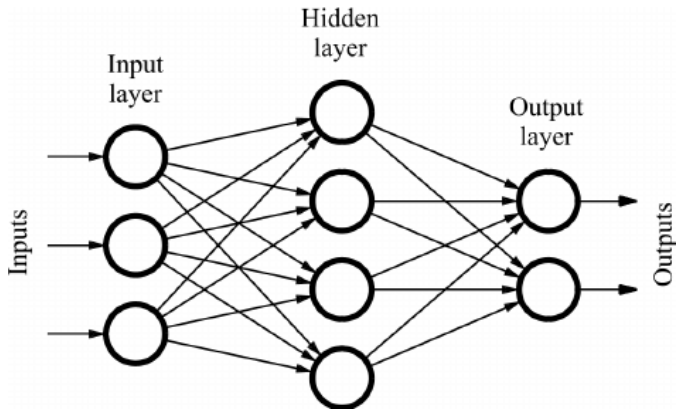


Figura 3.7: Feedforward neural network. Fonte: [14]

2. **Recurrent Neural Network (RNN)**: sono specificamente concepite per elaborare ed analizzare dati strutturati in modo sequenziale, come segmentazione di testi e serie temporali. L'utilizzo di dati precedenti (spazialmente o temporalmente) è finalizzato ad incrementare la performance della rete su dati

presenti e futuri. La peculitarità di questa rete è la presenza di loop interni, che hanno la funzione di immagazzinare le informazioni precedenti.

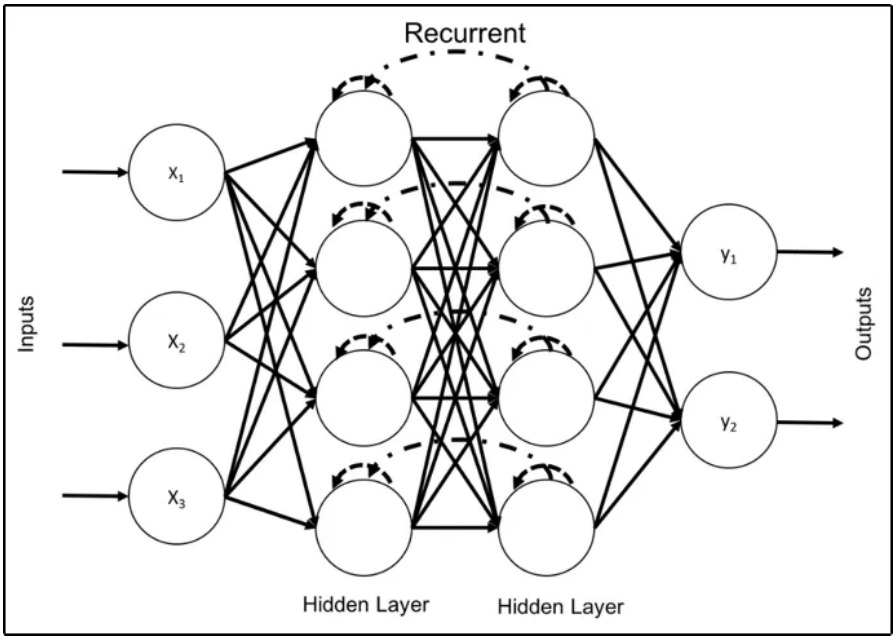


Figura 3.8: Recurrent neural network. Fonte: [15]

3. **Autoencoder**: è un algoritmo neurale di tipo *unsupervised*, il cui fine non è l'inferenza di un trend di dati, ma l'apprendimento della ricostruzione dell'input stesso. Per questa sua funzione, è tipicamente impiegato nella rimozione di rumori che alterano la bontà dei dati raccolti (*denoising autoencoder*).

4. **Convolutional Neural Network (CNN)**: è lo strumento computazionale su cui verte questo progetto di tesi, pertanto verrà trattata in maniera dettagliata nel prossimo paragrafo 3.4.

3.4 *Convolutional Neural Networks* (CNN)

Le **reti neurali convolutive (CNN)** sono classi di architetture ispirate alla corteccia cerebrale visiva. La loro popolarità si impose negli anni Ottanta, supportata da precedenti studi sulla funzionalità visiva dei neuroni, che valse il Premio Nobel a D. H. Hubel e T. Wiesel negli anni Cinquanta. Il concetto base di una rete convolutiva è di imitare la funzionalità di un neurone come entità dotata di *campo ricettivo*, e di visualizzarne l'interazione e la sovrapponibilità con gli altri simili, fino a riprodurre l'intero campo visivo, partendo da un input composto da dati strutturati a griglia.

3.4.1 Architettura di una rete convolutiva

Gli elementi caratterizzanti di una CNN sono essenzialmente **quattro**:

1. **layer convolutivo**
2. **layer di pooling**
3. **layer di flatten e layer di dropout**
4. **layer fully-connected**

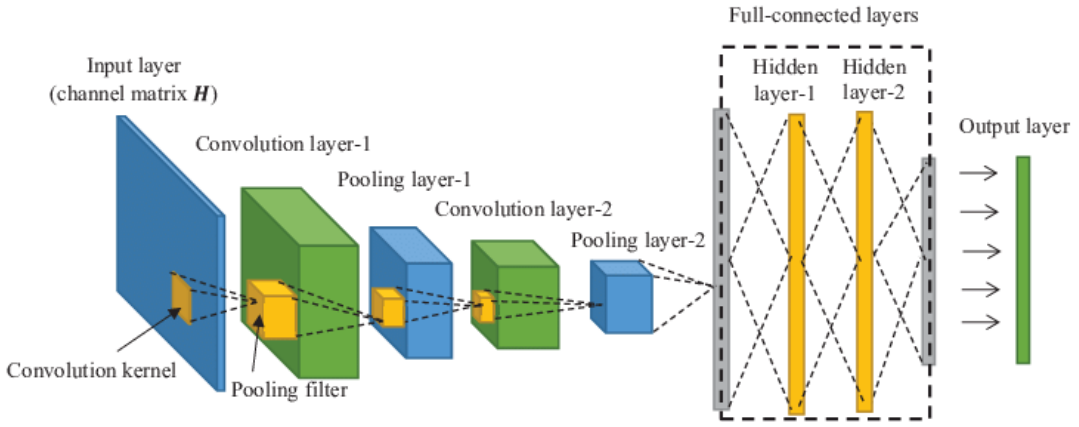


Figura 3.9: Generica struttura di una rete CNN. Fonte: [16]

Di rilevante importanza è la gestione della dimensionalità nell'architettura, che deve essere compatibile con quella dell'input. A differenza di una rete FCNN, in cui i neuroni sono disposti secondo un layer unidimensionale, una CNN sfrutta la tridimensionalità delle connessioni neuronali, risultando in dimensioni caratteristiche di *width* (larghezza), *height* (altezza), *depth* (profondità).

3.4.2 Filtri e kernel

Un **filtro** è una matrice di dimensioni ridotte contenente i pesi, ed è finalizzato a rilevare specifiche caratteristiche nei dati in ingresso. La funzione pratica del filtro è di scorrere sistematicamente lungo il set di dati (nel caso di un'immagine, lungo i pixel), eseguendo un'operazione di convoluzione. Questo è un processo di mappatura dinamica delle caratteristiche (o **feature map**) dell'input, e quantifica a livello numerico la rilevanza di queste: elevati valori implicano una rilevanza nelle feature, mentre, al contrario, bassi valori una certa trascurabilità.

A livello di nomenclatura, un filtro 2D è definito anche **kernel**, concetto non equivalente qualora questo filtro sia in 3D. In quest'ultimo caso, *kernel* è il termine per descrivere solo una *slice* del filtro 3D.

Un aspetto cruciale, qualora l'input sia un'immagine, è il numero di **canali** (**channels**), che indica quante componenti indipendenti costituiscono l'immagine. Ad esempio, un'immagine in scala di grigi possiede un unico canale (uno per ogni pixel, dove il valore indica l'intensità della luce), mentre un'immagine a colori RGB è composta da tre canali, uno per ciascuna delle componenti di colore: rosso (R), verde (G) e blu (B). Quando il filtro viene applicato su un'immagine, ogni canale viene convoluto con il filtro, e i risultati di ciascun canale vengono combinati per generare la **feature map** finale.

A titolo esplicativo, si riportano due immagini recanti rispettivamente un'operazione di filtering e di convoluzione bidimensionale:

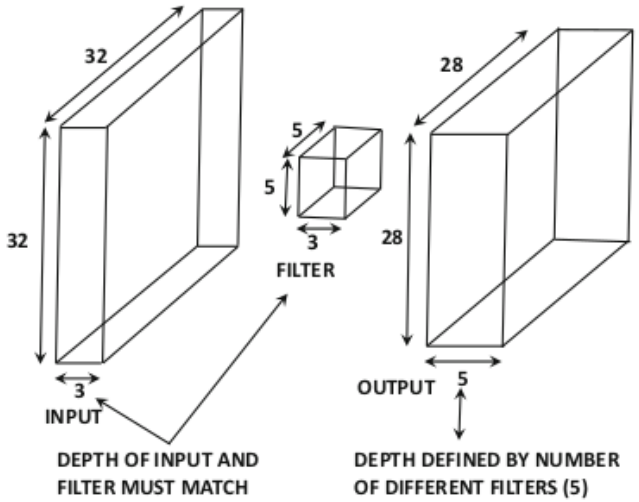


Figura 3.10: Processo di convoluzione fra input di dimensioni $32 \times 32 \times 3$ ed un filtro di dimensioni $5 \times 5 \times 3$. Fonte: [11]

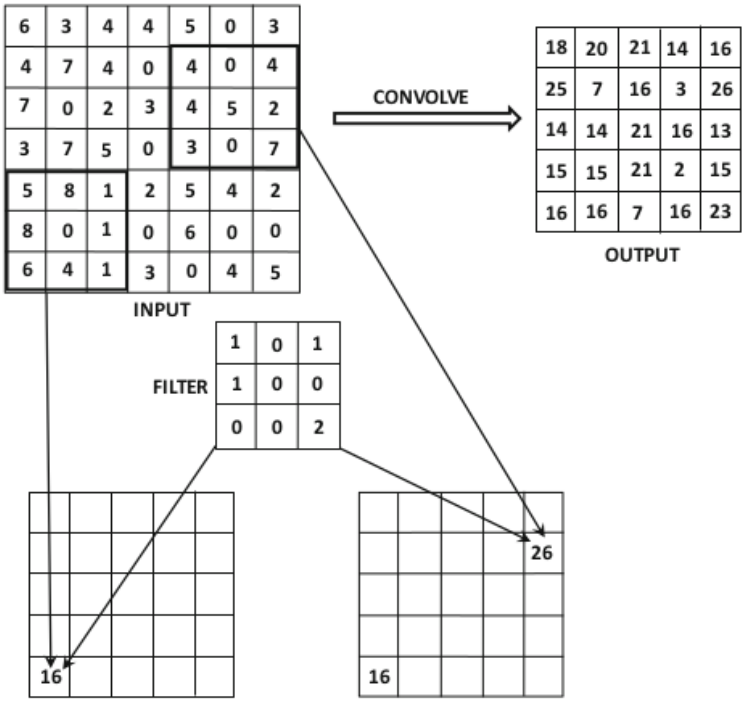


Figura 3.11: Operazione di convoluzione numerica. Fonte: [11]

3.4.3 Layer convolutivi

Rappresentano il *core* della rete neurale stessa, e, come detto poc'anzi, effettuano l'operazione di convoluzione, calcolando il prodotto scalare fra filtro e le componenti di input e producendo una **feature map**, 2D o 3D a seconda della dimensionalità di quest'ultimo.

Le principali caratteristiche di uno strato convolutivo sono le seguenti:

- **disposizione spaziale dell'output**: questa caratteristica definisce la struttura tridimensionale del volume prodotto dallo strato convolutivo. È determinata da tre aspetti principali:
  - **profondità del volume in output**: rappresenta il numero di filtri (o kernel) distinti che vengono applicati all'input. Ogni filtro è progettato per rilevare uno specifico tipo di *feature* (ad esempio, bordi orizzontali, curve, texture). Quindi, la profondità del volume di output corrisponde al numero di diverse *feature map* generate, ognuna come risultato dell'applicazione di un filtro;
  - **stride**: specifica di quanti pixel il filtro si sposta sull'input ad ogni passo durante l'operazione di convoluzione. Uno stride unitario significa che il

filtro si muove di un pixel alla volta (sia orizzontalmente che verticalmente), mentre uno stride maggiore fa sì che il filtro "salti" alcuni pixel. Lo stride influenza direttamente la dimensione spaziale (larghezza e altezza) della *feature map* di output;

- **padding**: è una tecnica utilizzata per aggiungere uno o più strati di valori (solitamente zeri) attorno al bordo dell'input. Questo viene fatto per due motivi principali: **controllo della dimensione spaziale dell'output** (senza padding, l'applicazione ripetuta di filtri convoluzionali tende a ridurre le dimensioni spaziali delle *feature map*); **preservazione delle informazioni ai bordi** (il padding assicura che anche le informazioni ai bordi contribuiscano in modo più significativo al calcolo delle *feature map* di output). La strategia più comune è lo *zero-padding*, dove vengono aggiunti zeri ai bordi;

- **connettività locale**: a differenza delle reti neurali completamente connesse (come le FCNN), dove ogni neurone è collegato a tutti i neuroni dello strato precedente, in uno strato convolutivo ogni neurone della *feature map* di output è connesso solo a una piccola regione spaziale dell'input. Questa regione è chiamata **campo recettivo del neurone**. Questa connettività locale sfrutta la natura spaziale delle *feature* in molte tipologie di dati (come le immagini).

- **condivisione parametrica**: all'interno di una singola *feature map*, tutti i neuroni utilizzano lo stesso identico set di pesi e bias per calcolare il loro output. Questo significa che se un filtro impara a rilevare un certo tipo di bordo in una parte dell'immagine, lo stesso filtro verrà utilizzato per cercare lo stesso tipo di bordo in ogni altra parte dell'immagine. La condivisione dei parametri ha due vantaggi principali: **riduzione del numero di parametri da apprendere** (questo rende il modello più efficiente in termini di memoria e di dati necessari per l'addestramento, riducendo il rischio di overfitting); **invarianza traslazionale** (poichè lo stesso filtro viene applicato su tutto l'input, la rete diventa più robusta alle traslazioni delle *feature* nell'input).

L'operazione di convoluzione numerica bidimensionale tra un'immagine di input discreta $I[x, y]$ (di dimensioni $X \times Y$) e un kernel discreto $K[u, v]$ (di dimensioni $U \times V$) produce un'immagine di output $O[x, y]$, le cui dimensioni dipendono dal padding utilizzato. Se non si utilizza padding, le dimensioni dell'output sarebbero $(X - U + 1) \times (Y - V + 1)$. La formula generale per calcolare l'elemento $O[x, y]$ è la seguente:

$$O[x, y] = \sum_{u=0}^{U-1} \sum_{v=0}^{V-1} I[x - u, y - v] \cdot K[u, v]$$

dove:

- $I[x, y]$: rappresenta l'immagine di input;
- $K[u, v]$: rappresenta il kernel o filtro;
- $O[x, y]$: rappresenta l'immagine di output, ossia il risultato della convoluzione;
- $\sum$: indica l'operazione di sommatoria;
- u e v: sono gli indici che scorrono sugli elementi del kernel;
- x e y: sono gli indici che scorrono sugli elementi dell'immagine di output.

Il termine $I[x-u, y-v]$ accede ai pixel dell'immagine di input, tenendo conto della posizione del kernel. Il termine $I[x-u, y-v] \cdot K[u, v]$, per ogni posizione (x, y) dell'output, moltiplica gli elementi del filtro con la porzione corrispondente dell'immagine di input.

È importante notare che, in pratica, l'indice dell'immagine di input $I[x-u, y-v]$ potrebbe cadere al di fuori dei limiti dell'immagine. In questi casi, si utilizzano tecniche di padding (come il suddetto *zero-padding*) per definire i valori di I al di fuori dei suoi limiti.

3.4.4 Pooling layer

Il layer di *pooling* è un componente fondamentale delle reti neurali convolutive, la cui funzione è di ridurre la dimensionalità delle *feature maps*, diminuendo così il numero di parametri e la complessità computazionale. Questa tecnica aiuta a prevenire l'overfitting e rende la rete più robusta alle piccole variazioni nell'input, migliorando la sua capacità di generalizzare a nuovi dati. Le operazioni di *pooling* più comuni includono il *max pooling*, che seleziona il valore massimo in una regione, e l'*average pooling*, che ne calcola la media. La dimensione della finestra di *pooling* e lo *stride* determinano il grado di riduzione della dimensionalità. È importante notare che il *pooling* comporta una perdita di informazioni spaziali, ma questo è spesso bilanciato dai vantaggi in termini di efficienza e robustezza.

3.4.5 Flatten layer e Dropout layer

- Il layer **dropout** è una tecnica di regolarizzazione per prevenire l'overfitting nelle reti neurali, ed affronta questo problema disattivando casualmente una frazione dei neuroni durante l'addestramento. Questo impedisce ai neuroni di co-adattarsi eccessivamente, migliorando la capacità di generalizzazione. Durante l'addestramento, ogni neurone ha una probabilità p di essere disattivato, dove p è un iperparametro regolabile.

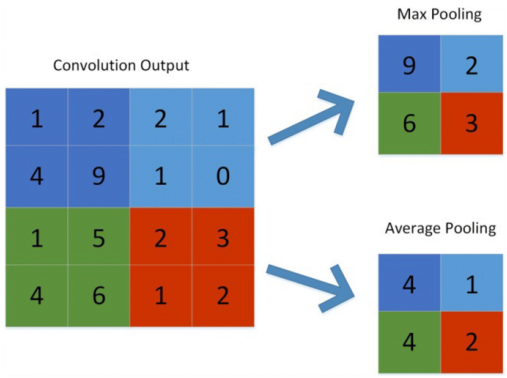


Figura 3.12: Esemplificazione di un processo di pooling secondo *max* ed *average*.

- Il layer **flatten** (di appiattimento) agisce come ponte tra i layer convoluzionali e i layer *fully connected*. Dopo che i layer convoluzionali e di pooling hanno estratto le caratteristiche rilevanti dall'immagine, queste sono rappresentate come tensori multidimensionali. Tuttavia, i layer *fully connected* richiedono input vettoriali unidimensionali. Il layer *flatten* risolve questo problema trasformando il tensore multidimensionale in un vettore 1D: acquisisce la matrice di pixel e la trasforma in una serie di numeri, che poi vengono forniti al successivo strato *fully connected*.

3.4.6 Fully connected layer

Lo strato *fully connected* (FC) agisce come una rete neurale feedforward (FCNN), in cui ogni neurone è connesso a tutti i neuroni degli strati adiacenti. Questo strato è fondamentale per la classificazione finale nelle reti neurali convolutive (CNN), poiché apprende combinazioni non lineari delle caratteristiche estratte dai livelli precedenti.

Capitolo 4

Metodi computazionali

Questo capitolo sarà dedicato alla descrizione e all'analisi dei metodi utili allo svolgimento delle simulazioni di fluidodinamica computazionale, effettuate per la creazione del dataset di lavoro. In particolare, verrà analizzato il **metodo dei volumi finiti (FVM)**, implementato nel software `OpenFOAM` (*Open-source Field Operation and Manipulation*). La versione utilizzata è `OpenFOAM 9`, e la scelta di quest'ultimo per questa ricerca è motivata dalla sua natura *open-source* e gratuita, che garantisce accessibilità e flessibilità, assieme alla sua ampia libreria di solutori e modelli sviluppati per la fluidodinamica.

4.1 Metodo dei volumi finiti

Il metodo dei volumi finiti [[17], [18]] nasce nel corso degli anni Settanta come seguito dei metodi delle differenze finite (FDM) e degli elementi finiti (FEM). La necessità di avere un metodo flessibile, la cui discretizzazione è effettuata direttamente nello spazio fisico senza alcuna necessità di trasformazione intermedia, è ciò che lo rende attualmente uno dei metodi più usati in ambito CFD. Riflette altresì la fisica e le proprietà integrali conservative che modella attraverso le equazioni di governo e dei loro termini matematici. Nell'immagine che segue (Figura 4.1), si illustra un tipico *workflow* nell'applicazione del metodo (in questo caso, applicazione dell'equazione di Fourier per trasporto di calore):

La geometria di interesse (anche definita **volume di controllo**) è dunque discretizzata in domini computazionali elementari definiti *celle*, tramite un processo di *meshing*, su cui verranno singolarmente risolte le equazioni di trasporto del fenomeno che si desidera modellare.

Come visibile da Figure 4.2 e 4.3, le celle sono fra loro interconnesse, con una notazione del tipo **north-south-west-east** e composti, adiacenti ad una cella

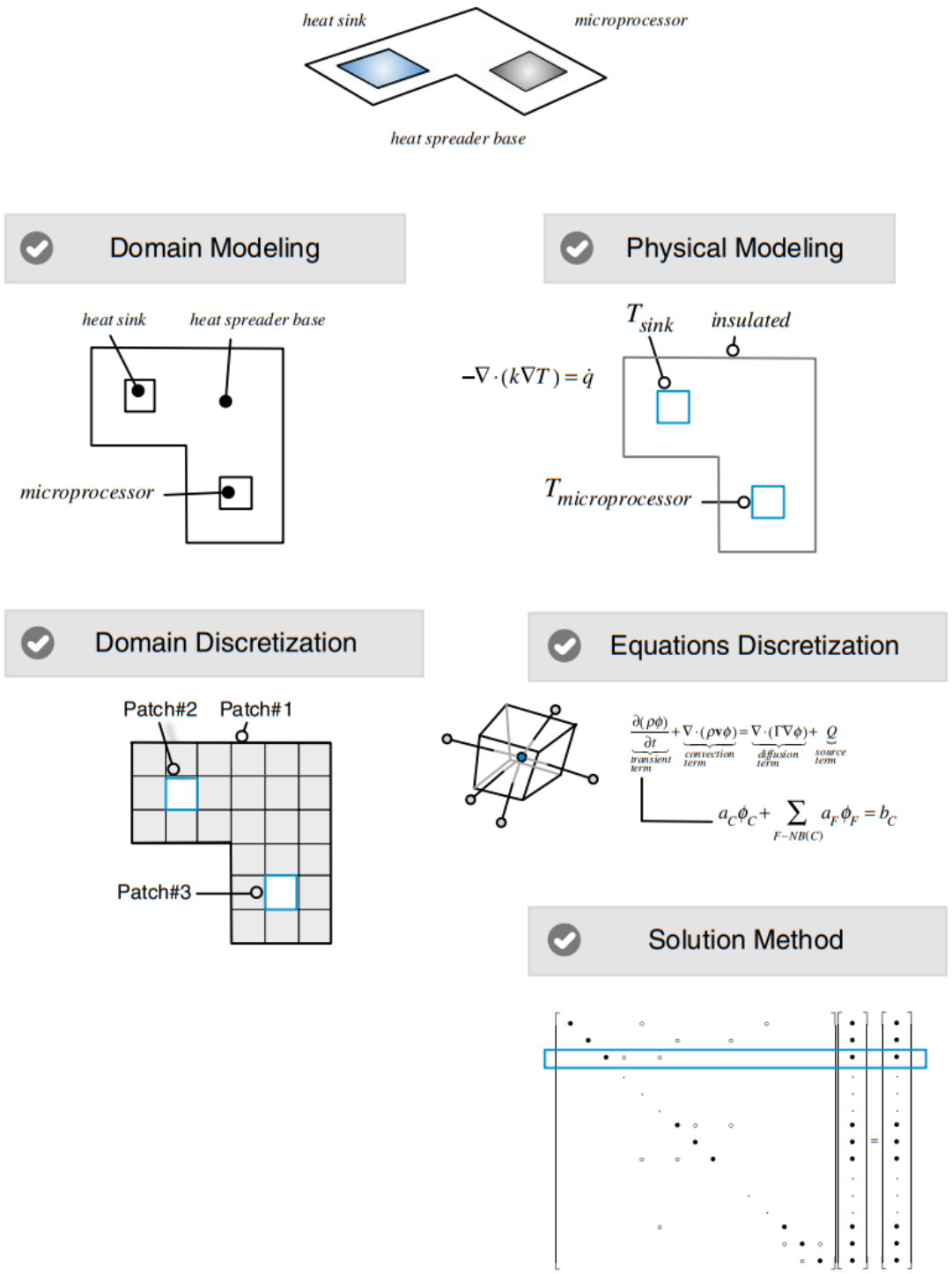


Figura 4.1: Illustrazione di un processo di FVM applicato ad un processore. Fonte: [17]

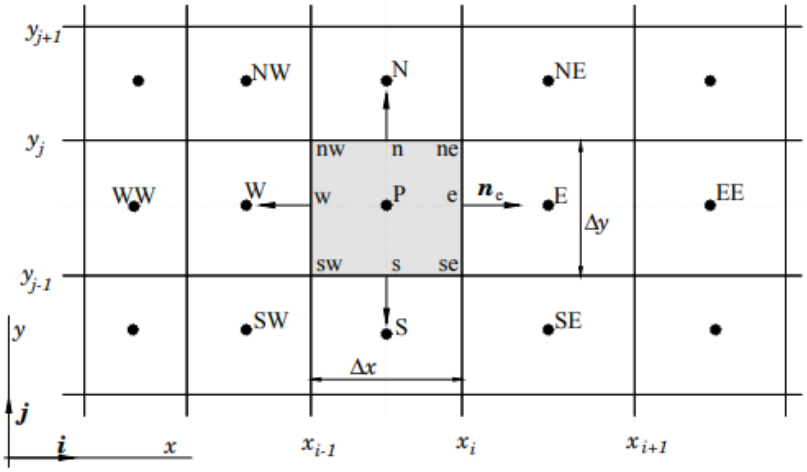


Figura 4.2: Discretizzazione 2D di un volume di controllo tramite FVM. Fonte [18]

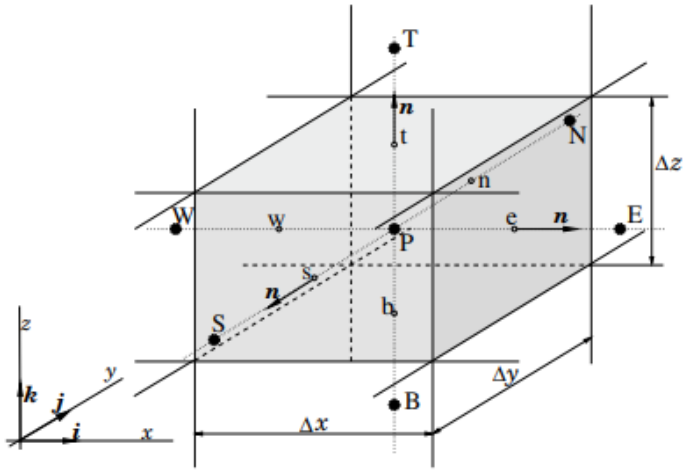


Figura 4.3: Discretizzazione 3D di un volume di controllo tramite FVM. Fonte [18]

centrata nel punto **P**. Le lettere maiuscole indicano i centri delle celle (**N, S, E, W**), le lettere minuscole le facce condivise dalla cella madre centrata in **P** con le adiacenti (**n, s, e, w**). Il flusso tra celle confinanti risponde ad un bilancio di tipo conservativo, che si applica alle quantità da simulare. Queste sono costanti all'interno della cella stessa, il cui centro viene utilizzato come punto di risoluzione. Le condizioni alla frontiera della cella vengono utilizzate per inizializzare, secondo un meccanismo a catena, le condizioni al contorno per le equazioni di trasporto nelle celle adiacenti. La somma di tutte le celle è, in ogni caso, pari al volume di controllo fornito per la simulazione.

4.1.1 Equazione semi-discretizzata

Il primo stadio di applicazione del metodo FVM consiste nell'integrazione dell'equazione di governo sui volumi finiti in cui il volume di partenza è stato suddiviso (le celle, per l'appunto). Tale equazione si presenta in una forma del tipo:

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{termine transitorio}} + \underbrace{\nabla\cdot(\rho\mathbf{u}\phi)}_{\text{termine convettivo}} = \underbrace{\nabla\cdot(\Gamma^{\phi}\nabla\phi)}_{\text{termine diffusivo}} + \underbrace{Q^{\phi}}_{\text{termine sorgente}} \tag{4.1}$$

dove ϕ è la generica variabile scalare conservativa. Nel caso *steady-state* (stazionario), il termine transitorio può essere eliminato, trascurando la dipendenza di ϕ dal tempo:

$$\underbrace{\nabla\cdot(\rho\mathbf{u}\phi)}_{\text{termine convettivo}} = \underbrace{\nabla\cdot(\Gamma^{\phi}\nabla\phi)}_{\text{termine diffusivo}} + \underbrace{Q^{\phi}}_{\text{termine sorgente}} \tag{4.2}$$

Per integrazione sul volume di controllo V_C, si ha:

$$\int_{V_c}\nabla\cdot(\rho\mathbf{u}\phi)dV = \int_{V_c}\nabla\cdot(\Gamma^{\phi}\nabla\phi)dV + \int_{V_c}Q^{\phi}dV \tag{4.3}$$

che per applicazione del teorema di Gauss (o della divergenza), risulta in un'equazione integrale lungo la superficie generata da V_C:

$$\oint_{\partial V_c}(\rho\mathbf{u}\phi)\cdot d\mathbf{S} = \oint_{\partial V_c}(\Gamma^{\phi}\nabla\phi)\cdot d\mathbf{S} + \int_{V_c}Q^{\phi}dV \tag{4.4}$$

In notazione indiciale, ed introducendo la notazione versoriale n_i con direzione e verso normale rispetto alla superficie della cella di superficie delimitante S:

$$\oint_{\partial V_c}(\rho u_i\phi)n_i dS = \oint_{\partial V_c}(\Gamma^{\phi}\frac{\partial\phi}{\partial x_i})n_i dS + \int_{V_c}Q^{\phi}dV \tag{4.5}$$

Per procedere alla valutazione del termine convettivo ($\oint_{\partial V_c}(\rho u_i\phi)n_i dS$), è necessario conoscere il valore della proprietà al centro delle facce limite; tuttavia,

l'unico valore conosciuto è quello al centro del volume di cella **P**, motivo per cui si procede ad una semplificazione dell'integrale mediante un bilancio del tipo:

$$\int_{\partial V_c}(\rho u_i n_i)\phi dS = \rho[(Su\phi)_w - (Su\phi)_e + (Su\phi)_s - (Su\phi)_n + (Su\phi)_t - (Su\phi)_b] \quad (4.6)$$

dove, oltre ai citati indici spaziali *n, s, w, e*, vi sono anche t e b, ossia la faccia superiore (top) e inferiore (bottom) rispetto alla cella in **P**.

Analogamente, si discretizza il termine diffusivo $\oint_{\partial V_c}(\Gamma^\phi \frac{\partial \phi}{\partial x_i})n_i dS$:

$$\begin{aligned}\int_{\partial V_c}(\Gamma\frac{\partial\phi}{\partial x_i})n_i dS = &-\left[\left(S\Gamma\frac{\partial\phi}{\partial x}\right)_w - \left(S\Gamma\frac{\partial\phi}{\partial x}\right)_e\right] + \\ &-\left[\left(S\Gamma\frac{\partial\phi}{\partial y}\right)_s - \left(S\Gamma\frac{\partial\phi}{\partial y}\right)_n\right] + \left[\left(S\Gamma\frac{\partial\phi}{\partial z}\right)_b - \left(S\Gamma\frac{\partial\phi}{\partial z}\right)_t\right] \quad (4.7)\end{aligned}$$

Il termine sorgente, d'altro canto, può essere approssimato rispetto al valore medio $\overline{Q}_\phi$ assunto nel volume di controllo V_c:

$$\int_{V_c} Q_\phi dV \approx \overline{Q}_\phi V_c \quad (4.8)$$

4.2 Schemi di discretizzazione spaziale

In questo paragrafo analizziamo la scelta del metodo di discretizzazione spaziale del termine convettivo, con le varie possibilità ed implicazioni. Infatti, se il valore della proprietà ϕ al centro della cella è noto, non lo è al centro delle facce, ragion per cui è necessario procedere a tecniche di interpolazione *ad hoc*. Nel contesto del FVM, esistono dozzine di metodologie atte a questo scopo [19]. Di seguito, si passeranno in rassegna le principali.

4.2.1 Upwind scheme (UDS)

Consiste nell'approssimazione del valore ϕ_e relativo alla faccia e sulla base del valore che la proprietà ϕ di interesse assume a monte, nella direzione principale del flusso nel sistema. È concettualmente equivalente all'utilizzo all'indietro/in avanti dell'approssimazione della derivata prima (si veda Figura 4.4).

In particolare:

$$\phi_e = \begin{cases}\phi_P & \text{if } (\mathbf{u}\cdot\mathbf{n})_e > 0 \\ \phi_E & \text{if } (\mathbf{u}\cdot\mathbf{n})_e < 0\end{cases} \quad (4.9)$$

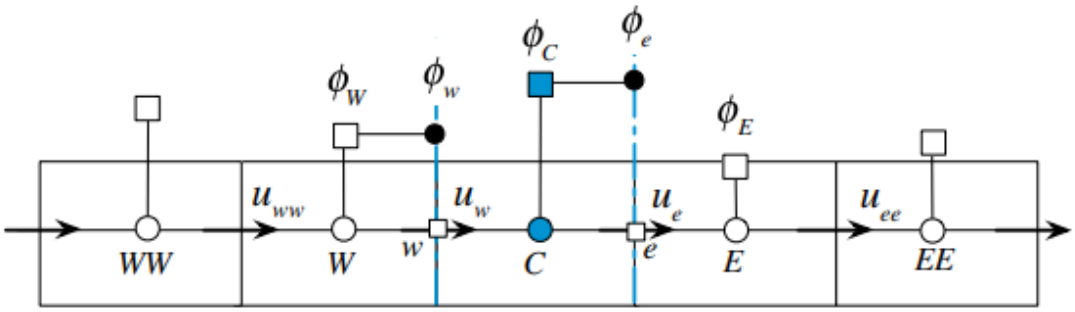


Figura 4.4: Schema upwind. Fonte [17]

Lo svantaggio associato a questo schema è di introdurre un fenomeno definito *diffusività numerica*, ossia un errore di troncamento che agisce in modo simile a un termine diffusivo fisico aggiuntivo. Questo termine di diffusività numerica è proporzionale alla grandezza della velocità del flusso e alla dimensione della cella (o passo della mesh). Infatti, espandendo ϕ_e rispetto al punto **P**:

$$\phi_e = \phi_P + (x_e - x_P)\left(\frac{\partial\phi}{\partial x}\right)_P + \frac{(x_e - x_P)^2}{2}\left(\frac{\partial^2\phi}{\partial x^2}\right)_P + H, \tag{4.10}$$

dove H raggruppa i termini di ordine superiore al secondo. Tuttavia, lo schema upwind tiene in conto solo i termini lineari (ossia è un metodo di prim'ordine), tale da trascurare monomi di ordine superiore. Il termine aggiuntivo diffusivo è dunque dato da:

$$f_e^{diff} = \Gamma_{num}\left(\frac{\partial\phi}{\partial x}\right)_e. \tag{4.11}$$

con $\Gamma_e^{\text{num}} = (\rho v)_e \Delta x/2$ coefficiente di diffusività numerica. Si nota come, per ottenere soluzioni accurate, sia necessario l'utilizzo di griglie molto fini, riducendo il fattore Δx associato.

Un parametro adimensionale fondamentale per valutare l'importanza relativa dei termini convettivi e diffusivi all'interno di una singola cella di calcolo è il **numero di Peclet di cella** (Pe_c). Questo è definito come:

$$Pe_c = \frac{\rho|v|\Delta x}{\Gamma} \tag{4.12}$$

Il valore del numero Pe_c fornisce indicazioni sul comportamento dello schema numerico considerato:

- **Regime a basso Pe_c ($Pe_c \ll 1$):** diffusione dominante. La diffusività numerica può essere *comparabile o maggiore* di Γ, causando eccessiva dissipazione e ridotta accuratezza.
- **Regime ad alto Pe_c ($Pe_c \gg 1$):** convezione dominante. L'errore di troncamento introduce diffusività numerica trasversale e dispersione numerica non fisica, smussando eccessivamente i gradienti e alterando il trasporto convettivo.

4.2.2 Linear Upwind scheme (LUDS)

Per superare le limitazioni dello schema upwind (UDS) di prim'ordine e ridurre l'effetto della diffusività numerica, si introduce lo schema **Linear Upwind Difference Scheme (LUDS)**. Questo schema mira ad ottenere una maggiore accuratezza nell'approssimazione del valore ϕ_e sulla faccia e utilizzando un'interpolazione lineare basata sui valori della proprietà ϕ in due nodi a monte. Invece di considerare solamente il nodo immediatamente a monte, LUDS prende in considerazione anche il nodo precedente nella direzione del flusso, fornendo una stima più precisa del gradiente locale e quindi del valore sulla faccia. In questo modo, si cerca di catturare una parte dell'informazione relativa alla variazione spaziale della proprietà ϕ, che viene trascurata nell'approssimazione di primo ordine di UDS.

La formulazione specifica di LUDS dipende dalla direzione del flusso. Consideriamo un flusso monodimensionale con velocità v.

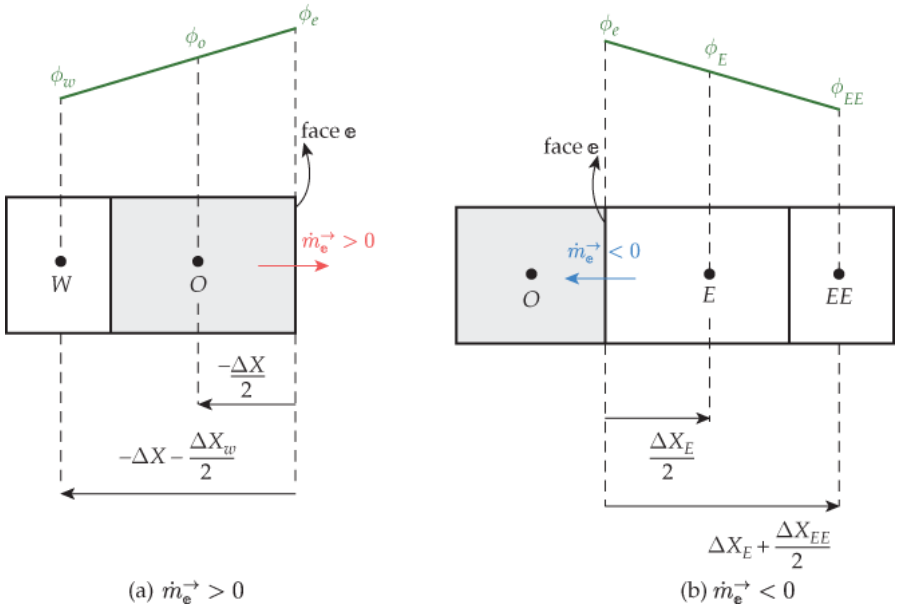


Figura 4.5: Schema linear upwind. Fonte [20]

- **Flusso positivo ($v_e > 0$, Figura 4.5a):** Se la velocità del flusso u sulla faccia e è positiva, il flusso proviene dal nodo P. I due nodi a monte rilevanti sono P (nella Figura identificato da O) e il nodo immediatamente precedente a P nella direzione del flusso (W). L'approssimazione di ϕ_e è data da:

$$\phi_e = \phi_P + (x_e - x_P)\left(\frac{\partial \phi}{\partial x}\right)_P \tag{4.13}$$

  Approssimando la derivata $\left(\frac{\partial \phi}{\partial x}\right)_P$ con una differenza all'indietro:

$$\left(\frac{\partial \phi}{\partial x}\right)_P \approx \frac{\phi_P - \phi_W}{\Delta x} \tag{4.14}$$

  Sostituendo e considerando $x_e - x_P = \Delta x/2$ per una griglia uniforme:

$$\phi_e = \frac{3}{2}\phi_P - \frac{1}{2}\phi_W \tag{4.15}$$

- **Flusso negativo ($v_e < 0$, Figura 4.5b):** Se la velocità del flusso u sulla faccia e è negativa, il flusso proviene dal nodo E. I due nodi a monte rilevanti sono E e il nodo immediatamente precedente a E nella direzione del flusso (EE). L'approssimazione di ϕ_e è data da:

$$\phi_e = \phi_E + (x_e - x_E)\left(\frac{\partial \phi}{\partial x}\right)_E \tag{4.16}$$

  Approssimando la derivata $\left(\frac{\partial \phi}{\partial x}\right)_E$ con una differenza in avanti (considerando la direzione negativa del flusso):

$$\left(\frac{\partial \phi}{\partial x}\right)_E \approx \frac{\phi_E - \phi_{EE}}{-\Delta x} = \frac{\phi_{EE} - \phi_E}{\Delta x} \tag{4.17}$$

  Sostituendo e considerando $x_e - x_E = -\Delta x/2$ per una griglia uniforme:

$$\phi_e = \frac{3}{2}\phi_E - \frac{1}{2}\phi_{EE} \tag{4.18}$$

L'utilizzo di LUDS porta a una riduzione significativa della diffusività numerica rispetto a UDS, in quanto introduce un'approssimazione di ordine superiore per il valore sulla faccia. Questo si traduce in una maggiore accuratezza. specialmente in problemi dove i gradienti della proprietà ϕ sono significativi. Tuttavia, in alcune situazioni di flussi fortemente convettivi o con forti gradienti, può ancora presentare delle limitazioni o introdurre oscillazioni non fisiche se non vengono adottate opportune tecniche di limitazione.

4.3 Architettura delle simulazioni OpenFOAM

Le simulazioni CFD svolte sono state finalizzate alla risoluzione di campo di moto e di concentrazione, rispettivamente risolvendo l'equazione di Navier-Stokes (1.4) e di trasporto di materia (1.7), con le dovute semplificazioni analizzate nel corso del Capitolo 1. Nel caso in analisi, la struttura di un `foamCase` ricalca quella standard comunemente diffusa. [21]

All'interno di un `foamCase`, si trovano le seguenti cartelle e file:

- `0`: è la cartella che contiene i file delle condizioni iniziali (rispettivamente di `p` ed `U` se in risoluzione di campo di moto, e di `C` ed `U` se in risoluzione di concentrazione). Ogni faccia del volume di controllo è associata ad una condizione specifica, che può essere di tipo *fixedValue* (valore specifico e costante al bordo), *symmetry* (condizione di simmetria sul bordo, con flusso e gradienti normali nulli), *zeroGradient* (gradiente nullo della variabile normale al bordo), assieme ad una notazione sulla dimensionalità della variabile, secondo la notazione SI, ossia un vettore di 7 numeri [M L T Θ N I J] che definiscono le potenze delle unità fisiche di una variabile (massa, lunghezza, tempo, temperatura, quantità di sostanza, corrente, intensità luminosa).

- `constant`: contiene i file relativi alla sturttura della mesh (`polyMesh` - *boundary, faces, neighbor, owner, points* e `triSurface` - ospita la geometria 3D da meshare), oltre ai file relativi alle `transportProperties` (modello di trasporto, ad esempio *Newtonian*, assieme alla viscosità cinematica `nu`) e `turbulenceProperties` (definisce il tipo di regime in cui si opera, e il modello di turbolenza RAS - *kEpsilon, kOmega, kOmegaSST*, etc.).

- `system`: è il repository dei file di configurazione essenziali per la definizione e l'esecuzione della simulazione numerica. Il suo contenuto è deputato alla specificazione dei parametri computazionali e degli schemi di discretizzazione necessari per la risoluzione delle equazioni del problema fisico in esame. Tipicamente contiene:

  - `controlDict`: definisce il protocollo temporale e di output della simulazione, e specifica l'intervallo di tempo di integrazione numerica (*deltaT*), i tempi di inizio e fine della simulazione (*startTime, endTime*), e la frequenza con cui i campi di soluzione vengono scritti su disco (*writeInterval, writeControl*);
  - i file `fvSchemes` e `fvSolution`: stabiliscono il framework numerico per la discretizzazione e la risoluzione delle equazioni differenziali parziali. Il primo prescrive gli schemi di approssimazione numerica per i termini derivativi spaziali (e.g., gradienti, divergenze, operatori laplaciani) e per

la derivata temporale: la scelta di tali schemi influenza direttamente l'ordine di accuratezza, la stabilità e le proprietà di conservazione del metodo numerico impiegato. Il secondo configura i solutori iterativi per i sistemi lineari e non lineari risultanti dalla discretizzazione. Esso include la selezione degli algoritmi di risoluzione, le tolleranze di convergenza per i criteri di arresto dei solutori;

- `blockMeshDict`: è il file di input per l'utility blockMesh. Il suo scopo è definire una griglia strutturata specificando il numero di suddivisioni (*blocks*), oltre che *vertices*, *edges* e definizione delle patch caratteristiche del volume di controllo tramite *boundary*;
- `snappyHexMeshDict`: è il file di input per l'utility snappyHexMesh. Definisce in modo parametrico il processo di generazione di una griglia poliedrica non strutturata a partire da una rappresentazione geometrica di superficie in formato STL. La sua struttura è organizzata in sezioni distinte, ognuna deputata al controllo di una specifica fase del processo di meshing, con l'obiettivo di evitare ridondanze nella definizione dei parametri.
- *time directories* per l'archiviazione dei campi di soluzione ad istanti temporali discreti nelle simulazioni transitorie.

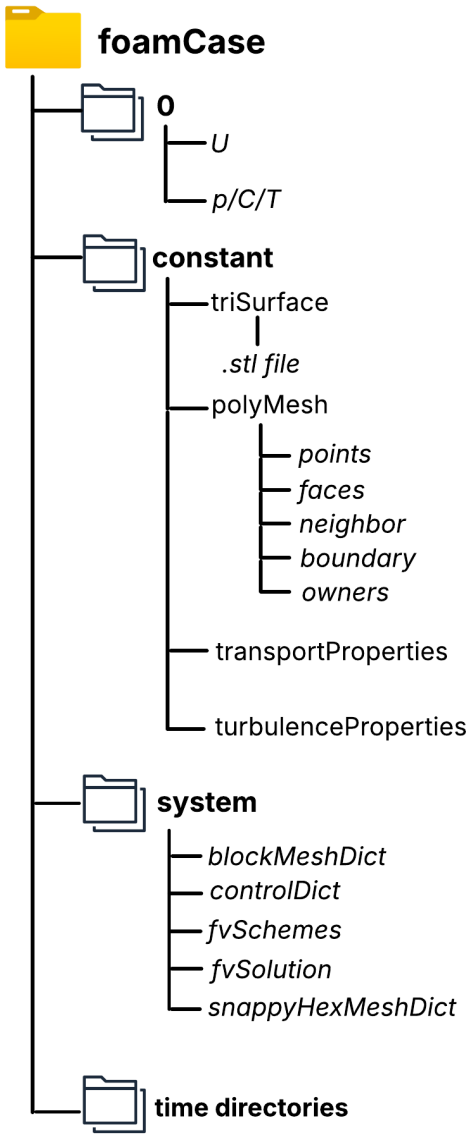


Figura 4.6: Struttura di un foamCase.

Capitolo 5

Implementazione dei casi di lavoro

Questo capitolo sarà dedicato alla descrizione *step-by-step* del processo di messa a punto dei casi di lavoro su cui verranno effettuate le simulazioni CFD, partendo dall'elaborazione delle geometrie fino allo step di *post-processing* alla fine di queste. Nel caso in analisi, questo flusso di lavoro è applicato alle geometrie cilindriche. Per quanto concerne altre geometrie, si è gentilmente attinto da dataset già messi a punto ed esistenti *in toto* (per le sfere, cfr. [10]).

Mentre la geometria sferica offre una semplicità intrinseca che ne facilita la caratterizzazione e la modellazione degli impaccamenti, il passaggio a geometrie più complesse introduce una notevole difficoltà nella loro descrizione univoca. La geometria di un pellet si colloca in questo spettro come un elemento di transizione: essa supera la perfetta regolarità della sfera, introducendo un grado di anisotropia, ma mantiene una forma definita e caratterizzabile, a differenza dell'impaccamento casuale e intrinsecamente variabile di frammenti, schiume e materiali simili, la cui descrizione univoca rappresenta una sfida significativa.

Più dettagliatamente, seguiranno paragrafi volti a giustificare la scelta delle dimensioni delle geometrie e dei parametri operativi. Ad ogni modo, i risultati complessivi dell'integrazione di queste in ambito *machine learning* verranno discusse nei capitoli successivi.

5.1 Generazione delle geometrie

Per la generazione degli impaccamenti presentati in questo lavoro, si è fatto ricorso a `Blender 3.0.1`, un potente software *open source* e gratuito, noto per la sua completa strumentazione dedicata alla creazione di grafica tridimensionale. Dalla modellazione poligonale alle simulazioni fisiche, `Blender` si configura come

una piattaforma estremamente versatile, adatta a svariate applicazioni. Un aspetto fondamentale che ha reso questo software particolarmente adatto a questo studio è la sua elevata estensibilità, resa possibile dal pieno supporto al linguaggio di programmazione `Python`. Questa compatibilità di *scripting* ha consentito una fruizione notevolmente più efficiente del software, grazie all'utilizzo e alla creazione di script `Python` specificamente sviluppati per automatizzare e controllare il processo di generazione degli impaccamenti. Data la loro estensione e specificità, i codici sorgente di tali script sono riportati per completezza in Appendice. Si riporta di seguito la struttura essenziale di una cartella di lavoro:

1. script **contenente i parametri geometrici** (`stmCase.yaml`): è un file in formato .yaml da cui attingerà lo script di generazione vera e propria per ricavare i parametri caratteristici della geometria che si vuole creare; ha una struttura del tipo:

```
additional_options:
  scale_factor: 1.0
  smoothness_level: 0
container:
  diameter: 5.0
  height: 5
  shape: box
  user_def_size: 14
fragmentation:
  flag: false
  noise: 0.2
  number_of_fragments: 20
grain_size_distribution:
  diameter: 1.0
  distr: gaussian
  grains: 2700
  mean: 1.1252979991857939
  pellet: Cylinder
  std_dev: 0.297082205358475
rigid_body:
  friction: 0.3
  refine_level: 3
  restitution: 0
```

   dove:

   - `additional_options` è il *dictionary* che regola parametri come fattore globale di scala della geometria (definito non nullo e positivo - `scale_factor` = 1 implica una non alterazione nelle proporzioni dell'oggetto) e il fattore di levigatura superficiale (*smoothness*);

- `container`: definisce le caratteristiche del contenitore in cui verrà ospitato l'impaccamento, ossia la forma (*shape*, che può essere cubica - *box*, definita da un lato *user_def_size* - o cilindrica - *cylinder*, definita da un'altezza *height* ed un diametro *diameter*);
- `fragmentation`: regola l'eventuale processo di frammentazione, se desiderato (*flag: true*, altrimenti *false*), così come il grado di casualità indotto dalla frammentazione (*noise*) ed il numero di frammenti (*number_of_fragments*);
- `grain_size_distribution`: definisce le caratteristiche dell'impaccamento, in termini di forma (tipo di *pellet*: *Cylinder, Icosphere, Raschig, etc.*), dimensione caratteristica (*diameter*), numero di grani (*grains*), media di altezza (nel caso di cilindri, *mean*), tipo di distribuzione (*distr*, che può essere *gaussian*, *lognormal* o *uniform*) con relativa deviazione standard (*std_dev*);
- `rigid_body`: impostazioni per le proprietà di corpo rigido degli oggetti durante la simulazione di impaccamento - attrito (*friction*), livello di dettaglio superficiale (*refine_level*), coefficiente di restituzione in seguito ad urto (*restitution*, con 0 per urti completamente anelastici).

2. script di **assegnazione di parametri geometrici** (`geometry_generator.py` in questo caso): automatizza e randomizza il processo di assegnazione delle variabili geometriche da manipolare mediante un ciclo `for` sul numero di impaccamenti desiderati (*n_samples*), generando un `stmCase.yaml` e raccogliendo poi il tutto all'interno di un file finale (*dataset.txt*). Datane l'estensione, per la sua consultazione, si richiama all'Appendice.

3. script di **generazione della geometria**: rappresenta il *core* del processo, ed è richiamato dal file `geometry_generator.py`. Effettua le seguenti operazioni:

   - creazione della scena `Blender` con impostazione del contenitore secondo specifiche dell'utente, ed impostazioni di corpo rigido;
   - creazione di array di grani, che verranno rilasciati per gravità all'interno della box;
   - eventuale frammentazione dei grani;
   - fase di pulizia dei grani collocati al di fuori della box e di ricavo del punto di *locationInMesh*, che sarà essenziale nella fase di *pre-processing* in `OpenFOAM`, di cui si parlerà in seguito.
   - esportazione della geometria nel formato **.stl**, leggibile e manipolabile dal software CFD `OpenFOAM`.

4. cartella **stl_files**: raccoglie i file .stl generati sequenzialmente alla fine di ogni ciclo del `geometry_generator.py`.
5. eventuali altre cartelle e script dedicati all'implementazione delle funzioni cui attingerà lo script in **(3)**, ad esempio per la defizione delle classi di distribuzione dei grani del letto impaccato.

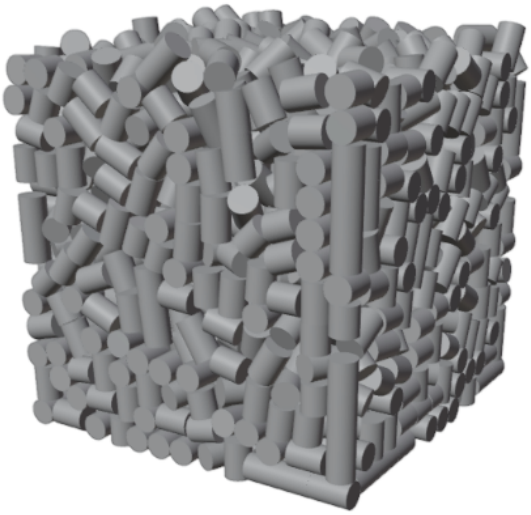


Figura 5.1: Impaccamento di cilindri realizzato tramite Blender.

Figura 5.2: Impaccamento di anelli Raschig realizzato tramite Blender.

5.2 Voxelizzazione delle geometrie

Con **voxelizzazione** si intende il processo di trasformazione di un oggetto continuo nello spazio tridimensionale in una sua rappresentazione discreta, costituita da un insieme finito di elementi volumetrici chiamati *voxel*. Analogamente ad un'immagine bidimensionale composta da pixel, un oggetto 3D voxelizzato è formato da una **griglia tridimensionale regolare** di "piccoli cubi" (*voxel*), dove ogni cubo indica la presenza o l'assenza di materiale all'interno di quella specifica regione dello spazio.

Tutti i voxel hanno la stessa dimensione, definita **pitch**. Questa dimensione determina la **risoluzione** della rappresentazione: voxel più piccoli offrono una maggiore precisione nella rappresentazione della forma, ma comportano anche un aumento significativo della quantità di memoria e di tempo per memorizzare l'oggetto.

Nel caso in analisi, la geometria tridimensionale è fornita sotto forma di file **.stl**, di cui si desidera voxelizzare una regione volumetrica rappresentativa dell'impaccamento (ciò che nel capitolo a seguire definiremo come REV - *representative elementary volume*). Considerando un REV cubico di lato L e un pitch uniforme p:

$$N = \frac{L}{p} \tag{5.1}$$

dove N è il numero di voxel lungo una dimensione. La risoluzione R è data da:

$$R = \frac{N}{L} = \frac{1}{p} \tag{5.2}$$

Il numero totale di voxel (N_{tot}) nel REV è:

$$N_{tot} = N^3 = \left(\frac{L}{p}\right)^3 \tag{5.3}$$

Da cui derivano:

$$p = \frac{L}{\sqrt[3]{N_{tot}}} \tag{5.4}$$

$$R = \frac{\sqrt[3]{N_{tot}}}{L} \tag{5.5}$$

Il processo di voxelizzazione della geometria 3D è stato svolto mediante l'utilizzo di uno script `Python` integrato con la libreria `trimesh` che, avendo come dati in input il centro della geometria, il lato ed il pitch, ha generato un tensore 3D assegnando il valore 0 alla parte solida (ossia i pellet) ed un valore 1 alla parte

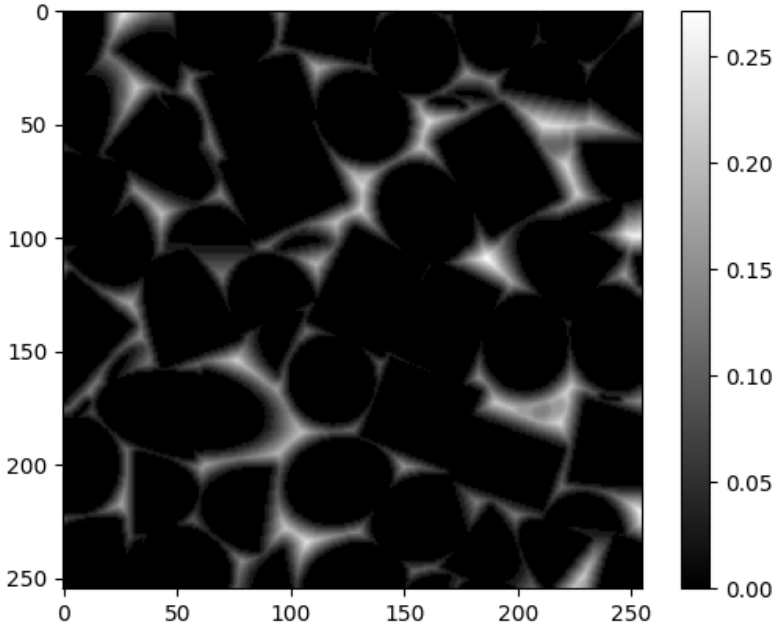


Figura 5.3: Esempio di voxelizzazione di un impaccamento di cilindri (*slice* 2D).

vuota (volume di fluido). Il tutto è stato poi rielaborato in termini di **distanza euleriana**, implementata tramite il comando `distance_trasform_edt` all'interno della libreria `scipy.ndimage`: essa quantifica un campo scalare in cui i voxel della fase solida mantengono valore 0, mentre i voxel della fase fluida assumono valori proporzionali alla loro distanza euclidea minima dalla superficie della fase solida. Ciò arricchisce la rappresentazione spaziale per la successiva analisi *deep learning*. Al fine poi di dare contezza, tramite la geometria, delle reali distanze assunte all'interno dei pori dell'impaccamento, essa è stata scalata di un fattore pari al pitch, riconducendo le distanze ad una scala fra 0 e la massima distanza intra-poro. In altri termini, si passa da una scala adimensionata in **unità voxel** ad una scala dimensionata in **unità di lunghezza**.

La Figura 5.3 esibisce quanto appena descritto, con la *colorbar* adiacente in unità di millimetri ed una risoluzione R pari a 255 voxel.

Per lo scopo del presente lavoro, si è proceduto ad una voxelizzazione delle geometrie con $\mathbf{R = 128}$ **voxel** per ogni lato cubico.

Per ogni dettaglio computazionale, si invita a consultare lo script sorgente contenuto in Appendice.

5.3 Esecuzione delle simulazioni

I tre step cardine nell'esecuzione delle simulazioni, finalizzate alla creazione del dataset, possono essere riassunti in modo sequenziale come segue:

1. **orchestrazione**: questa fase iniziale riguarda la preparazione e la configurazione di tutte le risorse e i parametri per le simulazioni;
2. **lancio**: è la fase di esecuzione vera e propria delle simulazioni. Una volta che l'ambiente è configurato e i file di input sono pronti, si avviano i solutori numerici sui sistemi di calcolo;
3. **pre-processing**: definizione e discretizzazione del volume di controllo in `OpenFOAM`;
4. **processing**: vengono risolti i campi di interesse mediante i solutori implementati nel software CFD;
5. **post-processing**: questa fase è dedicata all'analisi e all'estrazione dei dati rilevanti dai risultati.

Di seguito verranno analizzati con maggior dettaglio.

5.3.1 Orchestrazione

Questa fase consiste nell'assegnazione alla simulazione dei parametri caratteristici, inerenti la geometria (ricavati dopo esecuzione del software grafico `Blender`, secondo quanto descritto nel paragrafo 5.1) e altri parametri fisici randomizzati di cui si intende studiare l'effetto fluidodinamico (ad esempio pressione, diffusione, temperatura, e via discorrendo).

Ad eseguire questo compito è uno script `Python` (di cui a seguire si mostra un frammento) che, clonando iterativamente una cartella di riferimento all'interno di una directory (*simulations*), assegna a ciascuna di esse i suddetti parametri, con l'aggiunta del corrispettivo file geometrico .stl all'interno di *Flow/constant/triSurface*. Si tiene traccia di tutto il processo di assegnazione mediante la creazione a posteriori di un file in formato .txt, essenziale al fine di compiere un'analisi successiva (postProcessing, punto 5).

```
with open(file_path, mode='r', newline='') as file:
    reader = csv.DictReader(file, delimiter='\t')
    print(f"Column names in TXT: {reader.fieldnames}")
    for row in reader:
        if not row['Sample'].strip():
            continue
        try:
            sample_id = int(row['Sample'])
            dg_value = random.uniform(30e-09, 100e-09)
            sample = {
                'Sample': f"{sample_id:03d}",
                'Diameter': row.get('Diameter'),
                'Mean': row.get('Mean'),
                'Std_Dev': row.get('Std_Dev'),
                'Location_X': float(row['Location_X']),
                'Location_Y': float(row['Location_Y']),
                'Location_Z': float(row['Location_Z']),
                'Pressure': random.uniform(6.54235433573448E-07,
   7.771040393909E-06), #pressione in inlet normalizzata sulla
   densità (come richiesto da OpenFOAM)
                'dc': dc_value,
                'DT': float((1.380649e-23 * 298) / (3*
   3.14159265359 * 8.9e-4 * dc_value))
            }
            samples.append(sample)
        except ValueError:
            print(f"Warning: Invalid data in row {row}")
return samples
```

5.3.2 Lancio

La fase di **lancio** rappresenta un momento critico nel flusso di lavoro delle simulazioni, e segna il passaggio dalla preparazione all'esecuzione computazionale vera e propria. Anche in questa sede, il tutto è mediato da un codice `Python`, che qui assume il ruolo di gestione ed esecuzione intelligente, interagendo con il sistema di gestione dei carichi di lavoro del cluster di calcolo (**SLURM**, in questo caso). Per ogni configurazione di simulazione precedentemente predisposta, lo script inoltra una richiesta di esecuzione al sistema, specificando le risorse computazionali necessarie (per mezzo di un file .sbatch), come memoria, nodi e core di calcolo necessari. Il sistema di gestione dei carichi di lavoro si occupa poi di allocare dinamicamente le risorse disponibili sui nodi del cluster e di avviare l'esecuzione del software di simulazione per ogni scenario.

L'importanza di questo step risiede nella sua capacità di sfruttare la parallelizzazione intrinseca all'architettura di calcolo *high performance*. Invece di eseguire

le simulazioni una dopo l'altra, il sistema è in grado di distribuire il carico di lavoro su più processori e nodi, permettendo che molteplici simulazioni vengano eseguite contemporaneamente. Questo parallelismo è fondamentale per ridurre drasticamente i tempi di calcolo complessivi e per poter generare il dataset in un lasso di tempo ragionevole.

5.3.3 Pre-processing

La fase di **pre-processing** è dedicata alla creazione e preparazione della griglia computazionale, ossia la rappresentazione discreta del dominio fisico in cui avverrà la simulazione. La griglia è essenziale per risolvere numericamente le equazioni che governano il comportamento del sistema studiato. In OpenFOAM, questo processo coinvolge diversi strumenti e passaggi:

1. **creazione della griglia cartesiana base**: come step iniziale, viene generata una griglia semplice e strutturata, chiamata **griglia cartesiana**, utilizzando il comando `blockMesh`. Questa griglia servirà come punto di partenza per i passaggi successivi. Per ogni dettaglio circa il contenuto del *dictionary* associato, si rimanda al paragrafo 3.3.

2. **raffinamento e adattamento della griglia alla geometria .stl**: se la simulazione viene eseguita su un singolo processore, si procede direttamente con lo strumento `snappyHexMesh`. Questo permette di adattare la griglia di base alla geometria descritta dal problema - nel caso in analisi, l'impaccamento poroso. Il processo in questione si compone di tre step principali:

   - *castellation*: partendo dalla grigia al punto 1, ed avendo un punto collocato all'interno della parte vuota dell'impaccamento (il punto `locationInMesh` cui si accennava al paragrafo 3.3), effettua un'operazione di rimozione delle celle al di fuori della geometria solida, in modo da avvicinarsi quanto più possibile ad essa;
   - *snapping*: la grigia viene ulteriormente raffinata mediante posizionamento o spostamento dei vertici sulla superficie reale della geometria stessa, così da soddisfare i requisiti di qualità specificati dall'utente;
   - *layer addition*: per studiare bene il comportamento fluidodinamico in corrispondenza delle pareti (il cosiddetto *boundary layer*), opzionalmente si aggiungono strati di celle (*refinement levels*) progressivamente più raffinate tali da catturare in modo accurato lo svolgersi di fenomeni di trasporto in corrispondenza del *boundary*.

3. **parallelizzazione di calcolo (opzionale):** se la simulazione richiede l'uso di più processori per ridurre i tempi di calcolo, il dominio computazionale

viene decomposto in più parti utilizzando lo strumento `decomposePar`. Questo passaggio è fondamentale per distribuire il carico di lavoro tra i processori del sistema di calcolo. Successivamente, `snappyHexMesh` viene eseguito in parallelo, con ogni processore in opera sulla propria porzione di griglia. Una volta completato il raffinamento, le varie sezioni della mesh vengono ricomposte in un unico dominio con lo strumento `reconstructParMesh`.

4. **scalatura della mesh**: la griglia viene scalata per adattarsi alle proporzioni reali del fenomeno da studiare, tramite un fattore globale di scala σ, che può agire in modo selettivo o in tutte le dimensioni. Il comando utilizzato è `transformPoints "scale=(`$\sigma\ \sigma\ \sigma$`)"`.

5. **verifica della qualità della mesh**: questa viene analizzata con l'utility `checkMesh` per riassumerne la topologia (tipologie di celle - esaedriche, prismatiche, tetraedriche, etc.) e per assicurarsi che sia adatta alla simulazione, ossia che verifichi vari criteri, come ad esempio ortogonalità, *skewness*, *aspect ratio*.

5.3.4 Processing

Questo stadio rappresenta il cuore della simulazione CFD, e coinvolge l'utilizzo di `solver` appositamente progettati per adattarsi alle condizioni in cui si intende simulare. Esistono a tale scopo varie opzioni: `simpleFoam` è l'algoritmo che meglio si adatta a simulazioni stazionarie (laminari o turbolenti, di fluidi non comprimibili), `pimpleFoam` (casi transitori), e così via. Per lo studio che si intende effettuare, è stato utilizzato il primo solutore (**SIMPLE**: *SemiImplicit Method for Pressure Linked Equations*), al fine di risolvere la dipendenza pressione-velocità data dal sistema di equazioni differenziali di continuità e di Navier-Stokes (cfr. paragrafo 1.2), con l'assunzione, dunque, di incomprimibilità del fluido. L'algoritmo si basa su un processo iterativo che può essere schematizzato come segue:

1. **predizione del campo di moto**: si assume un campo di pressione e si risolve l'equazione di Navier-Stokes per ottenere una prima stima del campo di velocità **U**.

2. **risoluzione dell'equazione di Poisson per la pressione:** il campo di velocità ottenuto al punto precedente viene impiegato per ricavare un nuovo campo di pressione p attraverso la risoluzione dell'equazione di Poisson:

$$\frac{\partial^2 p}{\partial x_i^2} = -\frac{\partial}{\partial x_i}\left(\frac{\partial \rho v_i v_j}{\partial x_j}\right) \tag{5.6}$$

derivata applicando l'operatore divergenza all'equazione del bilancio di quantità di moto e sfruttando la relazione fornita dall'equazione di continuità.

3. **correzione e convergenza** il campo di pressione calcolato al punto 2 viene confrontato con quello assunto nella predizione iniziale (punto 1). Se la differenza non rientra in una tolleranza prestabilita (specificata in `fvSolution`), si avvia una nuova iterazione. In questa, il campo di moto viene ricalcolato utilizzando il nuovo campo di pressione. La procedura iterativa prosegue fino al raggiungimento della convergenza, ovvero quando la variazione tra i campi di pressione (e di velocità) tra iterazioni successive diviene sufficientemente piccola.

Le osservazioni ed assunzioni che si fanno in questa sede sono:

- **regime di flusso laminare** (valori di Re rilevati nell'ordine 10^{-3} - 10^{-4}), ergo non viene considerato il contributo di turbolenza nel `simpleFoam`;
- **perdita di carico** fra inlet ed outlet normalizzata a 0 rispetto alla *patch* di outlet;
- **no-slip condition** sulla superficie delle particelle solide;
- **condizione di simmetria di flusso** sulle altre facce del dominio computazionale;
- **effetti gravitazionali trascurabili**.

Al termine della risoluzione del campo di moto, si procede alla determinazione del campo di concentrazione attraverso il solver scalare `scalarTransportFoam`, risolvendo l'Equazione 3.1. L'obiettivo è simulare l'iniezione di una soluzione colloidale (avente ruolo di filtrato) all'interno del mezzo poroso. `scalarTransportFoam` si presta efficacemente sia a simulazioni stazionarie che transitorie, essendo specificamente sviluppato per scalari passivi. Con questo termine, si intende una quantità fisica trasportata dal fluido in movimento, il cui comportamento non influenza dinamicamente il moto del fluido stesso. Pertanto, questo rimane determinato univocamente dalle sue proprietà fondamentali, quali velocità e pressione.

Per quanto riguarda i criteri di convergenza per l'arresto delle simulazioni, stabiliti sui residui, questi sono riassunti nella seguente Tabella 5.1:

Tabella 5.1: Criteri di convergenza sui residui.

Variabile	**Criterio di convergenza**
p	10^{-6}
v_i	10^{-5}
C	10^{-6}

Nel setup numerico, la **concentrazione di contaminante** in ingresso è **normalizzata al valore unitario**: questa scelta è supportata dall'assenza di condizioni di turbolenza (che potrebbe alterare in modo rilevante il trasporto del colloide dando vita a moti vorticosi e fenomeni di mixing indesiderati) e dell'effetto di sedimentazione dovuto alla gravità (che, in caso contrario, potrebbe compromettere l'ipotesi di scalare passivo).

5.3.5 Post-processing

La fase di post-processing rappresenta il momento conclusivo dell'elaborazione dei dati CFD. L'obiettivo primario è l'estrazione e l'organizzazione dei risultati generati dalle simulazioni `OpenFOAM`. Questi dati, sia nella loro forma grezza che come base per il calcolo di ulteriori parametri, costituiscono il dataset di *features* fondamentale per l'addestramento della successiva rete neurale. Distinguiamo due approcci di post-processing, specifici per la simulazione di flusso e di filtrazione:

- **flusso**: attraverso uno script dedicato in `Python`, si ottengono le proprietà geometriche della mesh (numero di celle e volume totale), le caratteristiche del volume elementare rappresentativo (REV), la permeabilità e la porosità del mezzo, e la velocità di Darcy (o superficiale). Inoltre, sfruttando la definizione di `functions` nel file `controlDict`, si calcolano le componenti medie della velocità (*volAverage* mediate sul volume del mezzo), le velocità medie areali sulle superfici di ingresso e uscita (*surfaceAverage*), e la tortuosità dell'impaccamento.

- **filtrazione**: Anche in questo caso, si utilizzano `functions` specifiche definite nel `controlDict`. Queste permettono di determinare la concentrazione media della specie filtrata sulla superficie di uscita e nel volume del mezzo, oltre a calcolare i flussi molari (diffusivo e convettivo) in ingresso e in uscita. Questi parametri sono cruciali per la definizione del *filtration rate*, dettagliata nel paragrafo 1.4.2.

La raccolta di tutti questi risultati avviene tramite un consueto script Python (il cui codice completo è riportato in Appendice). Per illustrare i parametri ottenuti,

si presenta un frammento dello script, che include la chiamata di funzioni per la lettura e la scrittura dei dati in un file di testo:

```
if os.path.isdir(microscale_path):
        inlet_area = extract_inlet_area(microscale_path)
        inlet_velocity_z = extract_inlet_velocity_z(
    microscale_path)
        outlet_area = extract_outlet_area(microscale_path)
        outlet_velocity_z = extract_outlet_velocity_z(
    microscale_path)
        volume = extract_medium_volume(microscale_path)
        average_concentration = extract_average_concentration(
    microscale_path)
        porosity, permeability, tortuosity, reynolds,
    darcyan_velocity, mesh_volume, rev_volume =
    extract_results_data(microscale_path)
        ftotin, ftotout, tmeanout = extract_breakthrough_data(
    microscale_path)

        sample['Inlet Area'] = inlet_area
        sample['Inlet velocity'] = inlet_velocity_z
        sample['Outlet Area'] = outlet_area
        sample['Outlet velocity'] = outlet_velocity_z
        sample['Volume'] = volume
        sample['Average Concentration'] = average_concentration
        sample['Porosity'] = porosity
        sample['Permeability'] = permeability
        sample['Tortuosity'] = tortuosity
        sample['Reynolds'] = reynolds
        sample['Ftotin'] = ftotin
        sample['Ftotout'] = ftotout
        sample['Average area concentration out'] = tmeanout
        sample['Darcyan velocity'] = darcyan_velocity

        if rev_volume is not None and mesh_volume is not None:
            sample['Porous medium volume'] = rev_volume -
    mesh_volume
        else:
            print(f"Warning: Volume not calculable for {
    microscale_path} (rev or mesh volume missing)")
            sample['Porous medium volume'] = None

        updated_samples.append(sample)
else:
    print(f"Warning: Folder not found: {microscale_path}")
```

5.4 Caratterizzazione del dataset di pellet

La versatilità d'uso e di dimensioni rende le geometrie cilindriche ampiamente utilizzate all'interno dell'industria chimica e di processo, con applicazioni in:

- **catalisi eterogenea**: pellet cilindrici (ottenuti ad esempio per estrusione) sono comunemente utilizzati come supporti inerti per la deposizione di catalizzatori attivi. Questa configurazione geometrica consente di bilanciare in modo efficace area superficiale disponibile per la reazione e resistenza al flusso, riducendo la caduta di pressione nel letto catalitico e migliorando l'efficienza complessiva del processo;
- **adsorbimento e filtrazione**: materiali porosi come gel di silice, zeoliti o carboni attivi vengono modellati in forma cilindrica per facilitarne l'utilizzo in letti fissi. Trovano applicazione in processi di separazione gas-liquido (ad esempio per la purificazione dei gas esausti industriali o per la deumidificazione), nonché nei trattamenti con resine a scambio ionico (come nella demineralizzazione o addolcimento dell'acqua);
- **bioreattori e trattamenti ambientali**: i mezzi porosi a cilindri possono fungere da substrato per la crescita microbica in bioreattori destinati a processi biochimici o ambientali, come il trattamento biologico delle acque reflue o la degradazione di inquinanti. La loro struttura agevola lo sviluppo di biofilm e favorisce lo scambio di massa tra fase liquida e fase solida.

I parametri da fissare per la caratterizzazione geometrica dell'impaccamento sono i seguenti:

- **diametro:** $\boldsymbol{D}$
- **altezza media:** $\bar{H}$
- ***aspect ratio*:** $\xi = \frac{\bar{H}}{\boldsymbol{D}}$
- **coefficiente di variazione**: $\sigma_{norm} = \frac{\sigma}{\bar{H}}$

Si è effettuata una ricerca e una raccolta di alcuni dati presenti in letteratura in merito a queste dimensioni, i cui risultati sono riassunti nella seguente Tabella 5.2:

Tabella 5.2: Dimensioni caratteristiche di pellet cilindrici secondo letteratura.

D (mm)	$\bar{H}$ (mm)	ξ (-)	**Fonte**
1.5 - 3.2	2 - 7	1.33 - 2.19	Zhao *et al.* [22]
1.5	5	3.33	Kaiser *et al.* [23]
0.29 - 9.22	1.02 - 33.29	3.5	Brunner *et al.* [24]

e si sono formulate le seguenti assunzioni (Tabella 5.3):

Tabella 5.3: Parametri geometrici scelti.

Parametro	**Valore**
Diametro D (mm)	1
Altezza $\bar{H}$ (mm)	1 - 2
Coefficiente di variazione σ_{norm} (-)	0.0 - 0.3
Aspect ratio ξ (-)	1 - 2

In particolare, l'*aspect ratio* è stato regolato in modo da evitare un eccessiva variabilità del REV (che si vedrà a breve) tale che esso possa contenere mediamente un numero quanto più costante di pellet. Se l'*aspect ratio* variasse in un range più ampio, questa condizione sarebbe invalidata.

5.4.1 Analisi dimensionale

In seguito alla creazione del dataset (cfr. paragrafo 5.4), avente una dimensione pari a 280 geometrie ad impaccamento cilindrico, si è voluto caratterizzare lo stesso in termini di distribuzione di **altezze** e di **diametro di Sauter** d_{32} (a parità di diametro di base D). Quest'ultimo è definito come il diametro di una sfera avente lo stesso rapporto volume/area specifica della particella in analisi, o, in termini matematici:

$$d_{32} = \frac{d_v^3}{d_s^2} \tag{5.7}$$

dove:

$$d_s = \sqrt{\frac{A_p}{\pi}} \tag{5.8}$$

è il cosiddetto **diametro di superficie** (con A_p area superficiale della particella), e:

$$d_v = \left(\frac{6V_p}{\pi}\right)^{1/3} \tag{5.9}$$

il **diametro di volume** (con V_p volume della particella).
Ergo la relazione si riduce all'espressione:

$$d_{32} = 6\frac{V_p}{A_p} \tag{5.10}$$

Per una geometria cilindrica, avente altezza media H e diametro costante D, si ha che:

$$V_p = \frac{\pi D^2 H}{4}, \quad A_p = \pi D \left(\frac{D}{2} + H \right) \tag{5.11}$$

e dunque:

$$d_{32} = \frac{3DH}{D + 2H} \tag{5.12}$$

La sfera equivalente rappresenta una descrizione geometrica semplificata, ma altamente utile, che permette di correlare la morfologia reale di una particella ai parametri fluidodinamici che la caratterizzano, come la resistenza al moto o la distribuzione di velocità, fattori cruciali, ad esempio, nella caduta di pressione.

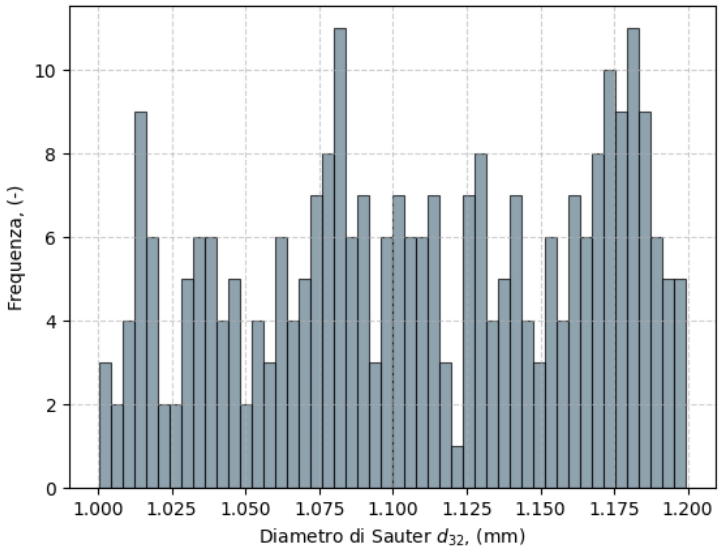


Figura 5.4: Distribuzione dei diametri di Sauter.

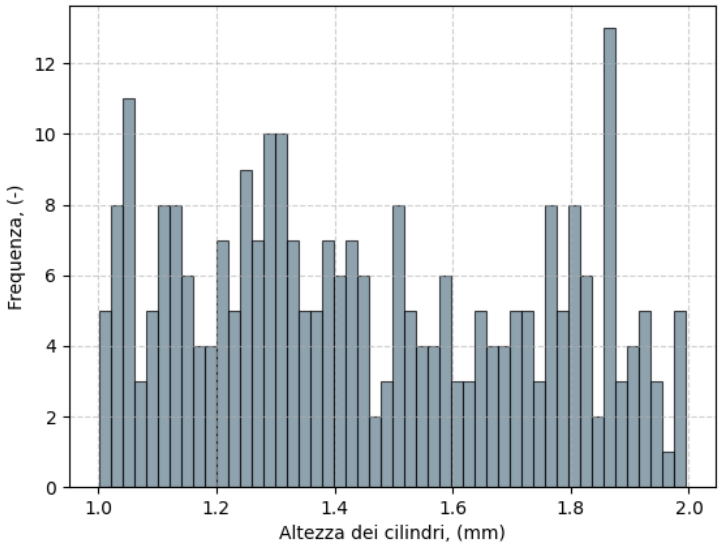


Figura 5.5: Distribuzione delle altezze.

È possibile ora dunque procedere alla caratterizzazione geometrica delle goemetrie che compongono il dataset.

I due strumenti statistici [25] utilizzati per questa descrizione sono:

- **Kernel Density Estimation (KDE)**;
- **Empirical Cumulative Distribution Function (ECDF)**.

La **Kernel Density Estimation (KDE)** è una tecnica non parametrica utilizzata per stimare la **densità di probabilità** di una variabile. A differenza dell'istogramma, che è una rappresentazione discreta della distribuzione dei dati, la KDE fornisce una stima continua della funzione di densità.

La formula per la KDE è la seguente:

$$\hat{f}(h) = \frac{1}{Nw} \sum_{i=1}^{N} K\left(\frac{h - h_i}{w}\right) \tag{5.13}$$

dove:

- $\hat{f}(h)$ è la stima della densità di probabilità all'altezza h;
- N è il numero totale di dati nel campione (nel caso in analisi, il numero di geometrie nel dataset);

- w è la **larghezza della finestra** o **bandwidth**, che determina l'ampiezza del kernel attorno a ciascun dato;
- K è la funzione **kernel**, che può essere una funzione gaussiana, uniforme, o simile;
- h è il punto (di altezza) in cui si stima la densità, mentre h_i è l'i-esimo dato (di altezza) osservato nel campione.

La KDE è particolarmente utile per visualizzare la forma della distribuzione dei dati in modo più fluido rispetto a un istogramma, e viene spesso utilizzata quando si vuole evitare il rumore introdotto dai bin in un istogramma tradizionale. Nel nostro caso, con un kernel di tipo gaussiano, si ottiene la seguente distribuzione di densità:

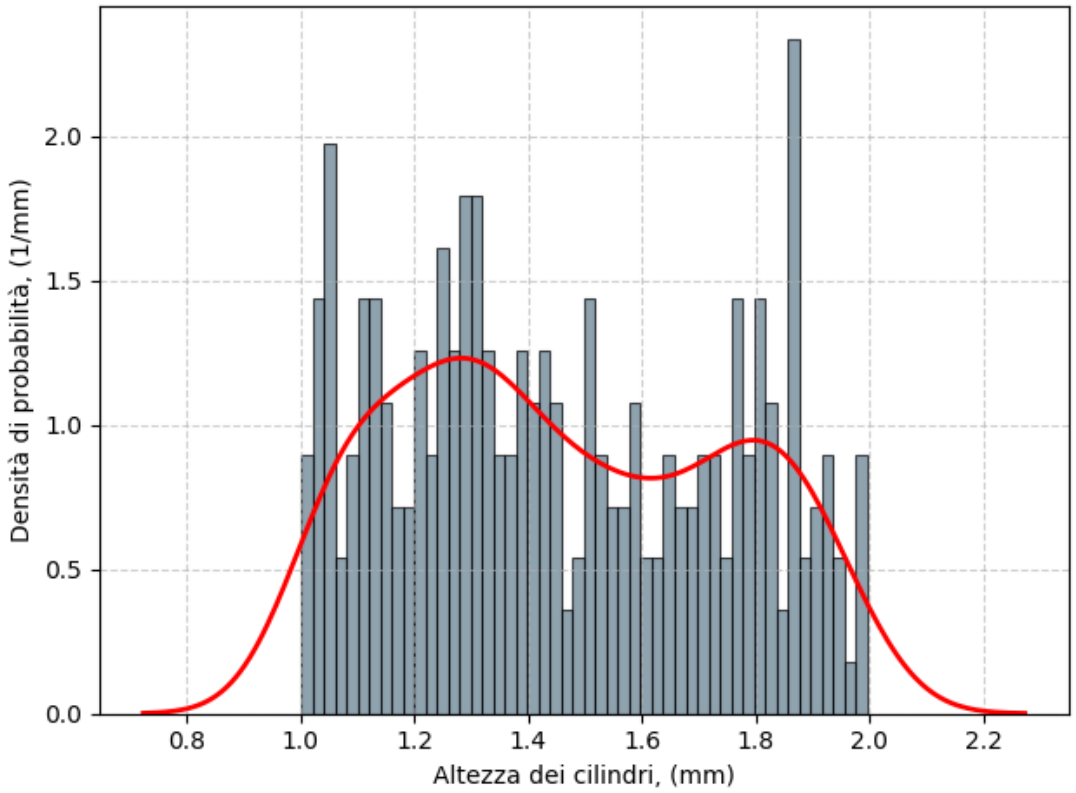


Figura 5.6: Densità di distribuzione delle altezze tramite KDE.

In questo grafico, l'istogramma rappresenta la **densità di probabilità** delle altezze dei cilindri, non la frequenza assoluta, tale che l'area di ogni barra è

proporzionale alla probabilità che l'altezza di un cilindro cada in quell'intervallo. L'area totale di tutte le barre dell'istogramma è pari all'unità.

La Figura 5.6 evidenzia delle regioni di maggiore densità in corrispondenza di un diametro pari a circa 1.3 mm e 1.9 mm, comportamento effettivamente confermato dall'analisi discreta mediante istogramma.

Si passa ora in rassegna l'**Empirical Cumulative Distribution Function (ECDF)**, definita come:

$$\hat{F}(h) = \frac{\text{Numero di altezze } h_i \text{ nel campione tali che } h_i \leq h}{N} = \frac{\sum_{i=1}^{N} I(h_i \leq h)}{N} \tag{5.14}$$

dove:

- $\hat{F}(h)$ è la funzione di distribuzione cumulativa empirica nel punto h (altezza);
- N è il numero totale di dati nel campione (nel caso in analisi, il numero di geometrie nel dataset);
- h_i rappresenta l'i-esima altezza del cilindro nel campione.
- $I(h_i \leq h)$ è la funzione indicatrice, definita come:

$$I(h_i \leq h) = \begin{cases} 1 & \text{se } h_i \leq h \\ 0 & \text{se } h_i > h \end{cases}$$

La ECDF rappresenta, per ciascun valore x, la frazione di osservazioni minori o uguali a x. A differenza della densità (come nel caso della KDE), la ECDF è una funzione *non decrescente* e a tratti costante. Applicata al dataset di interesse, si ha che:

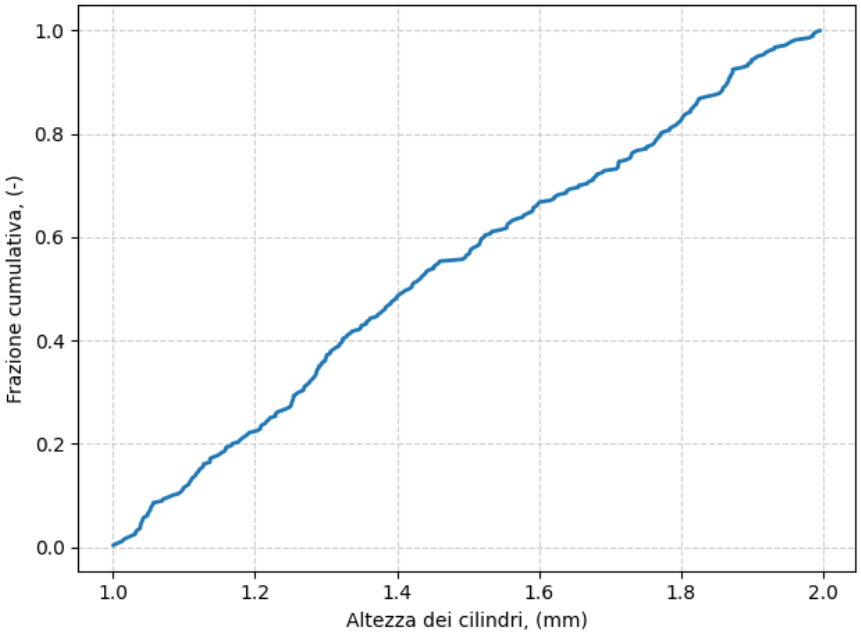


Figura 5.7: Distribuzione ECDF delle altezze.

Nonostante la presenza di picchi nella stima della densità tramite KDE a 1.3 mm e 1.9 mm, l'ECDF presenta un andamento quasi monotonicamente crescente. Tuttavia, in corrispondenza di questi valori, si osserva una distinguibile variazione nella pendenza dell'ECDF, con una risalita mediamente più ripida. Questo indica chiaramente come la maggiore concentrazione di dati, evidenziata dalla KDE in quelle specifiche regioni, si traduca in un incremento più rapido della funzione di distribuzione cumulativa empirica, confermando un significativo raggruppamento di osservazioni.

5.5 Caratterizzazione del dataset di sfere

Pur esulando dal contesto delle geometrie cilindriche, è di essenziale importanza procedere ad un'analisi sommaria del preesistente dataset di particelle sferiche [10], complementare a quello cilindrico, che verrà utilizzato successivamente per la messa a punto di reti neurali convoluzionali (CNN).

Anche in questo caso, come accade per i pellet, si dispone di **280 geometrie sferiche** con annessi dati fluidodinamici. Hanno un diametro che varia fra 0.11 - 0.21 mm, con un REV selezionato pari a circa 0.88 mm (per una definizione rigorosa del concetto di REV, si rimanda al paragrafo 6.1 del capitolo successivo).

Di seguito, la distribuzione del diametro di queste, con relativa frequenza:

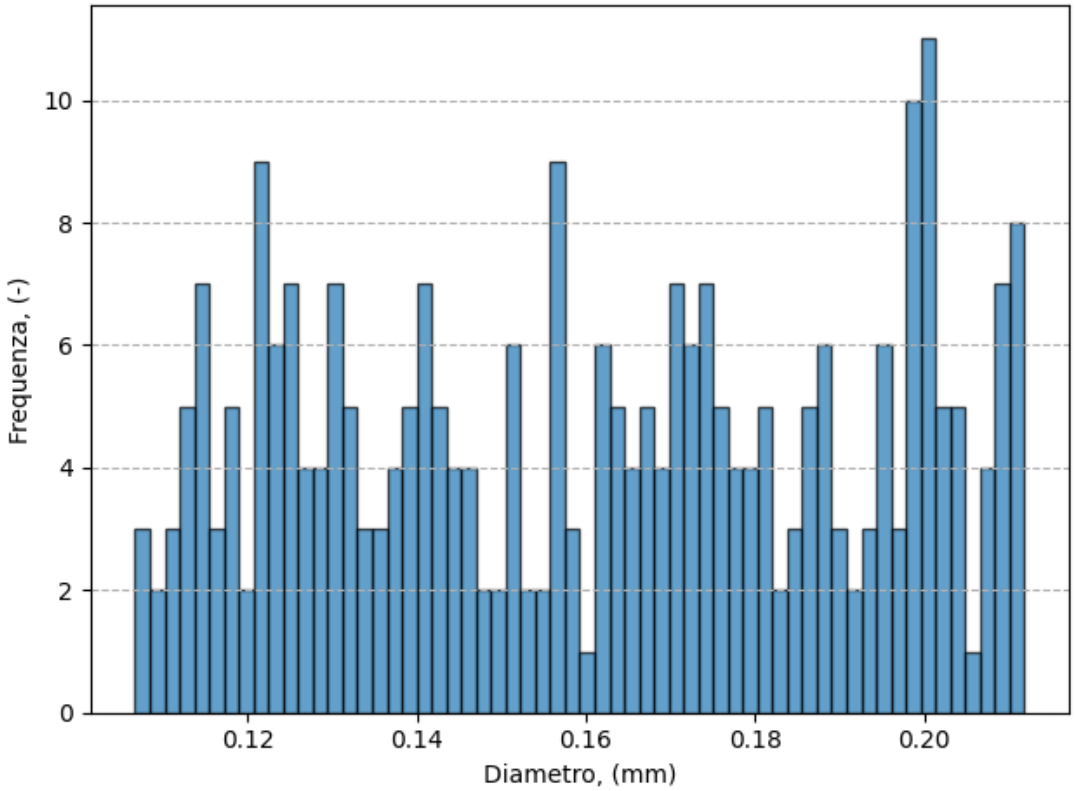


Figura 5.8: Distribuzione dei diametri (sfere).

Passando alle *features* caratteristiche da indagare tramite algoritmi neurali, si hanno:

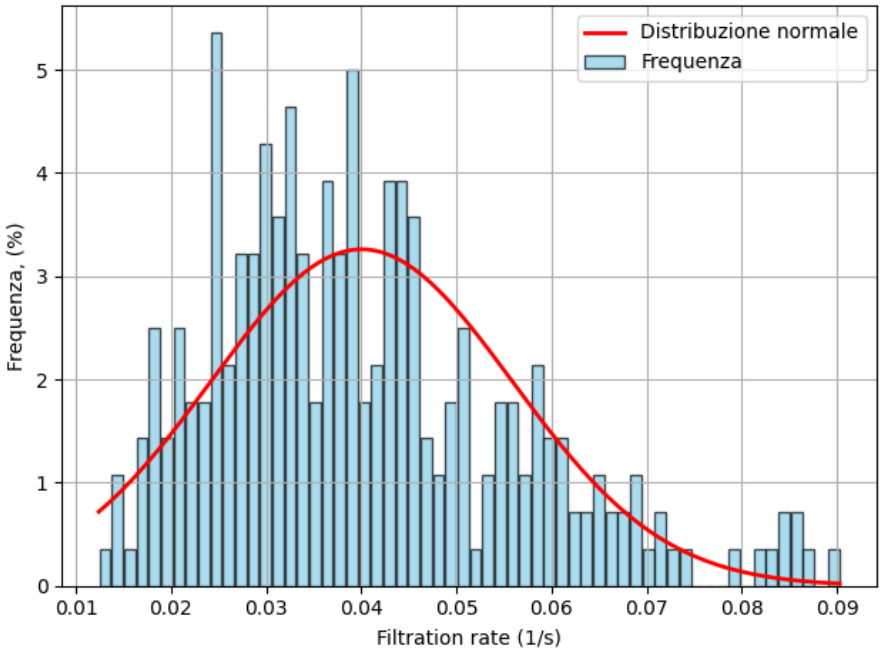


Figura 5.9: Distribuzione di filtration rate (sfere).

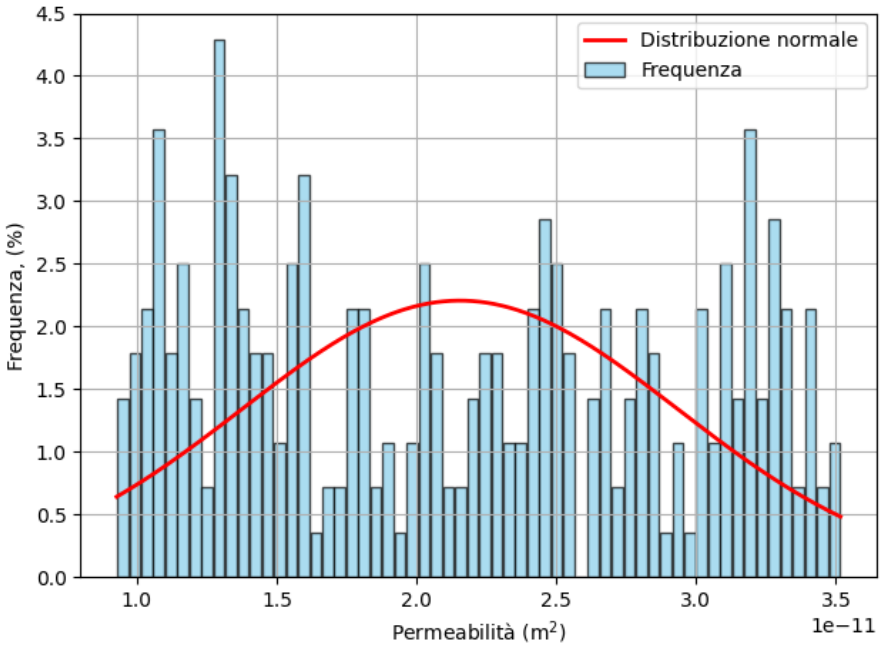


Figura 5.10: Distribuzione di permeabilità (sfere).

Parametro	**Valore**
Media $\bar{K_f}$	4.003×10^{-2} s^{-1}
Deviazione standard σ	1.587×10^{-2} s^{-1}
Coefficiente di variazione σ_{norm}	0.396

Tabella 5.4: Parametri di distribuzione gaussiana di filtration rate (sfere).

Parametro	**Valore**
Media $\bar{k}$	2.155×10^{-11} m^2
Deviazione standard σ	7.799×10^{-12} m^2
Coefficiente di variazione σ_{norm}	0.362

Tabella 5.5: Parametri di distribuzione gaussiana di permeabilità (sfere).

Da notare è la conformazione non propriamente *normale* della distribuzione dei valori di permeabilità. Tuttavia, si è ritenuto opportuno porla in termini gaussiani per alcune osservazioni che verranno svolte nella sezione a seguire.

Per quanto concerne la voxelizzazione delle sfere, anche in questo caso, come per i pellet, la risoluzione è stata settata su **128 voxel** per ciascun lato del REV, il tutto in funzione della distanza euclidea (espressa in *mm* - Figura 5.11).

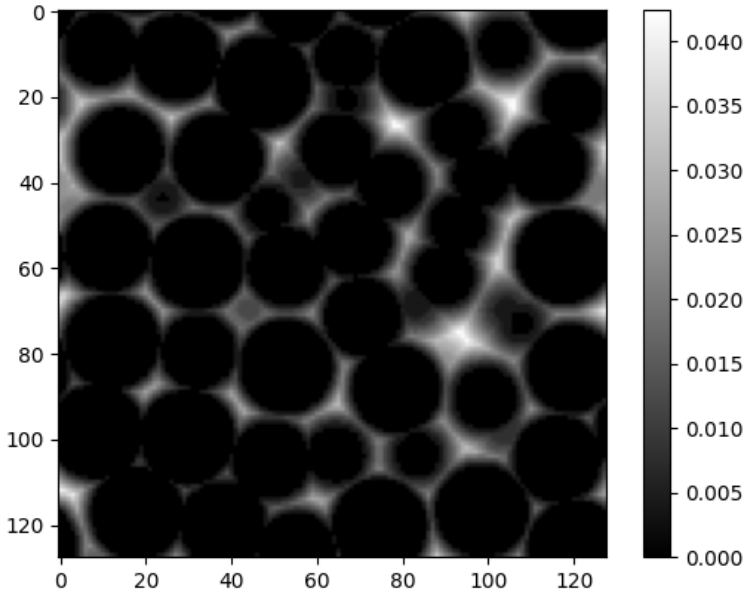


Figura 5.11: Esempio di voxelizzazione di un impaccamento di sfere (slice 2D).

Nota. La distanza euclidea calcolata per gli impaccamenti di pellet **non** è stata normalizzata rispetto alla distanza euclidea massima del preesistente dataset di sfere. Questa strategia, sebbene precluda una diretta comparabilità di scala con le geometrie del dataset di riferimento, è stata adottata per preservare l'informazione intrinseca relativa alle grandezze spazialmente dipendenti (diffusività, permeabilità, ecc.). Una normalizzazione avrebbe infatti introdotto un artefatto di riscalamento, potenzialmente compromettendo la capacità della rete neurale di discriminare impaccamenti morfologicamente analoghi ma caratterizzati da differenti scale di distanza.

5.6 Parametri di processo

Per quanto invece concerne i **parametri di processo** scelti, questi ricadono essenzialmente sulla **pressione in ingresso** al mezzo poroso e al **diametro caratteristico delle particelle colloidali**, che regola il processo diffusivo mediante il coefficiente di diffusione $\mathcal{D}$.

- **pressione in ingresso** p_{in}: il valore della pressione in inlet è stato calcolato sulla base di quanto assunto in [10], e correlato al caso in esame mediante un gradiente minimo e massimo lungo il mezzo, tale da avere delle feature caratteristiche (quali permeabilità e filtration rate) quanto più vicine dimensionalmente al range del precedente dataset. In particolare, avendo il volume elementare una forma cubica di lato 7 mm (cfr. paragrafo 6.1), e con un gradiente pari a 9.31818×10^{-5} - 1.10682×10^{-3} Pa/mm, si è ricavata una pressione in ingresso nel range pari a 6.523×10^{-4} - 7.748×10^{-3} Pa;
- **diametro dei colloidi** d_c: sulle base delle ipotesi formulate in [10], anche in questo caso si ripropone un diametro di elementi colloidali pari a 30 - 100 nm, con un coefficiente di diffusione $\mathcal{D}$ dell'ordine di $10^{-12} - 10^{-11}$ m^2/s.

Tabella 5.6: Parametri di processo scelti.

Parametro	**Valore**
p_{in} (Pa)	6.523×10^{-4} - 7.748×10^{-3}
d_c (nm)	30 - 100

Si ricorda che, essendo la pressione in uscita costante e definita pari a 0 (da condizioni al contorno dettate nel file `0/p`), la p_{in} non è altro che la caduta di pressione globale lungo il volume elementare.

Capitolo 6

Integrazione CFD - *machine learning*

L'obiettivo principale di questo capitolo è presentare e analizzare i risultati derivanti dalla creazione di un nuovo dataset contenente **geometrie cilindriche** (che, per brevità, di qui in poi definiremo anche **pellet**). Questa aggiunta mira a introdurre una maggiore variabilità negli impaccamenti porosi rispetto al dataset di sfere esistente (cfr. Marcato *et al.* (2022), [10]), con lo scopo di generalizzare le architetture neurali a dataset di natura eterogenea. Seguiranno paragrafi volti a giustificare la scelta del REV, dei risultati fluidodinamici e di accoppiamento CFD - *machine learning*, sia in ambito FCNN che CNN.

6.1 Grid independence e analisi del REV

La **grid independence** è la condizione in cui i risultati di una simulazione CFD convergono a un valore stabile all'aumentare della risoluzione della griglia computazionale (*mesh*). A livello pratico, si verifica quando un'ulteriore raffinazione della *mesh* non produce variazioni significative nelle grandezze fisiche di interesse, indicando che la soluzione diviene indipendente, per l'appunto, dalla strategia di discretizzazione spaziale. Nel caso in analisi, i valori di interesse coincidono con le *features* che si desidera inferire mediante modelli machine learning, ergo permeabilità e filtration rate. Per lo scopo di questa tesi, non si è ritenuto necessario svolgere un ulteriore studio al riguardo, e le ipotesi si basano su quanto rilevato dall'indagine di Marcato *et al.* [10], con i seguenti parametri caratteristici:

CPD*	18
Livello di raffinamento	2

*CPD: celle per diametro (cilindrico).

Tabella 6.1: Parametri di meshing.

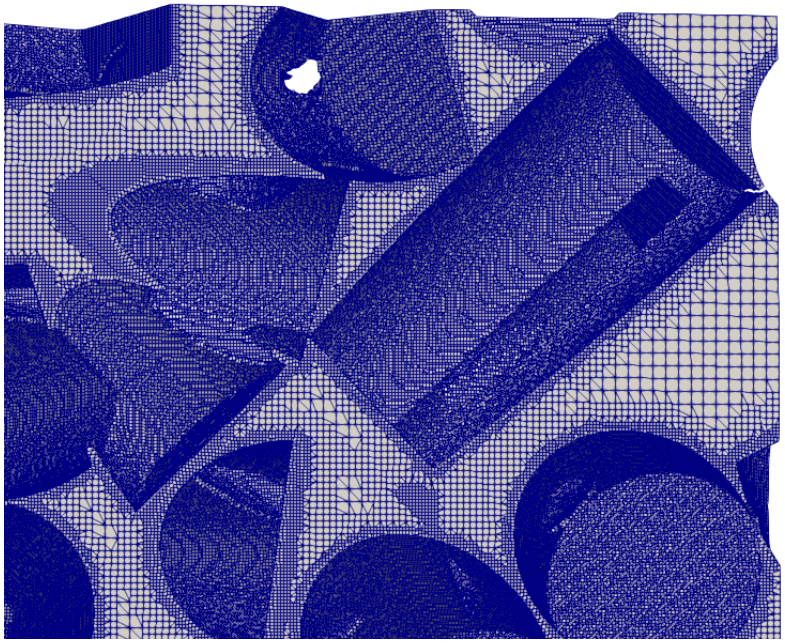


Figura 6.1: Strategia di meshing.

Per quanto concerne la scelta del **REV** (*representative elementary volume*), si è fatto riferimento alla definizione fornita da J. Bear nel suo "*Dynamics of Fluids in Porous Media*" [26]: esso è definito come il più piccolo volume di controllo all'interno del mezzo poroso in cui le proprietà strutturali — come la porosità e la distribuzione solido-vuoto — possono essere mediate in modo da risultare statisticamente rappresentative dell'intero sistema. In tale volume, le grandezze macroscopiche risultano spazialmente uniformi, rendendo possibile l'applicazione di modelli di tipo continuo.

Dal punto di vista statistico, il REV è sufficientemente grande da contenere una configurazione eterogenea del mezzo, ma abbastanza modesto da poter essere considerato puntuale su scala macroscopica. Esso funge da connettore tra la trattazione microscopica (scala dei pori) e quella macroscopica (scala dell'impaccamento), consentendo una modellazione efficace e generalizzabile delle proprietà del mezzo poroso.

Al fine di tenere conto della variabilità delle geometrie nel dataset, si sono selezionati tre campioni di impaccamento, con cilindri aventi altezza media $\bar{H}$ rispettivamente 1 mm, 1.5 mm e 2 mm (con coefficienti di variazione σ_{norm} casuali).

I risultati sono quelli che seguono nelle Figure 6.2 - 4:

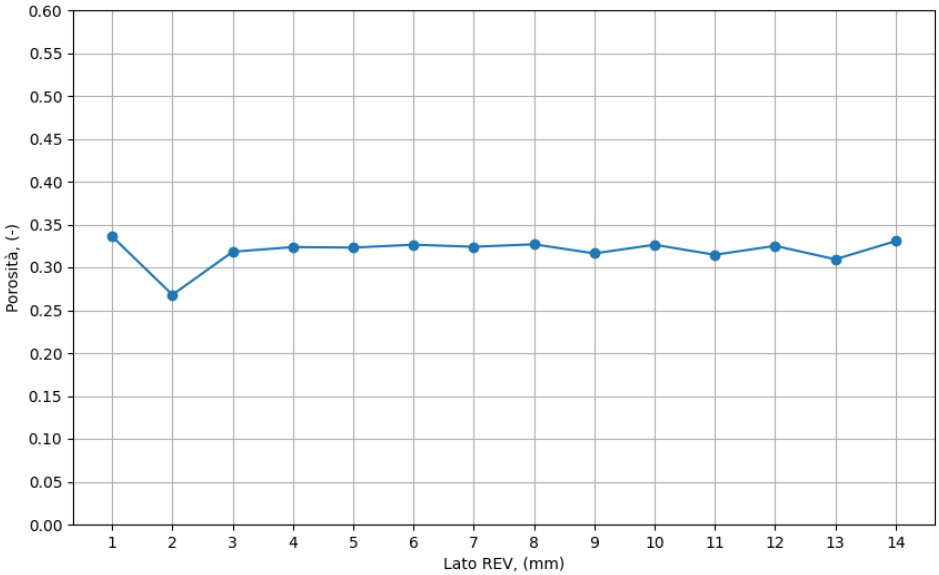


Figura 6.2: Studio REV per $\bar{H} = 1$ mm.

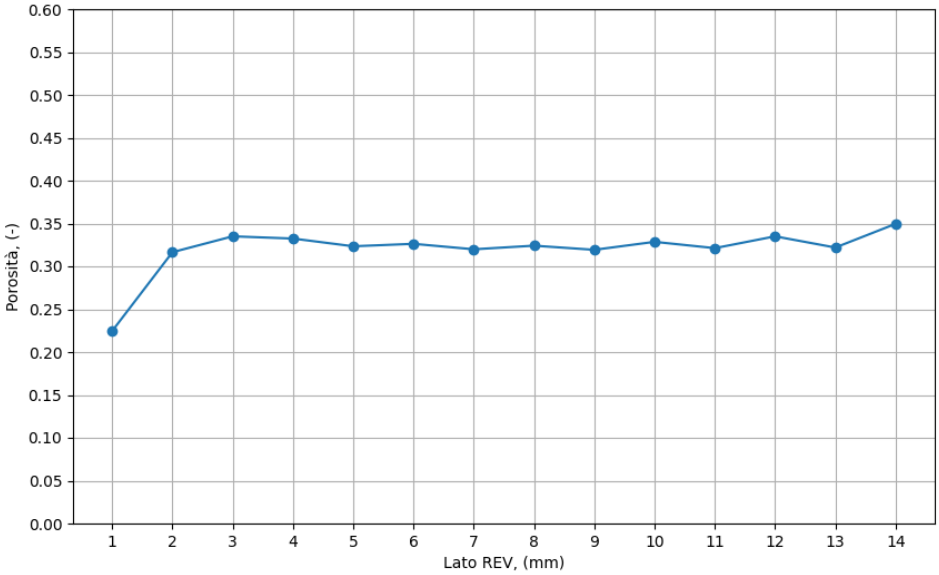


Figura 6.3: Studio REV per $\bar{H} = 1.5$ mm.

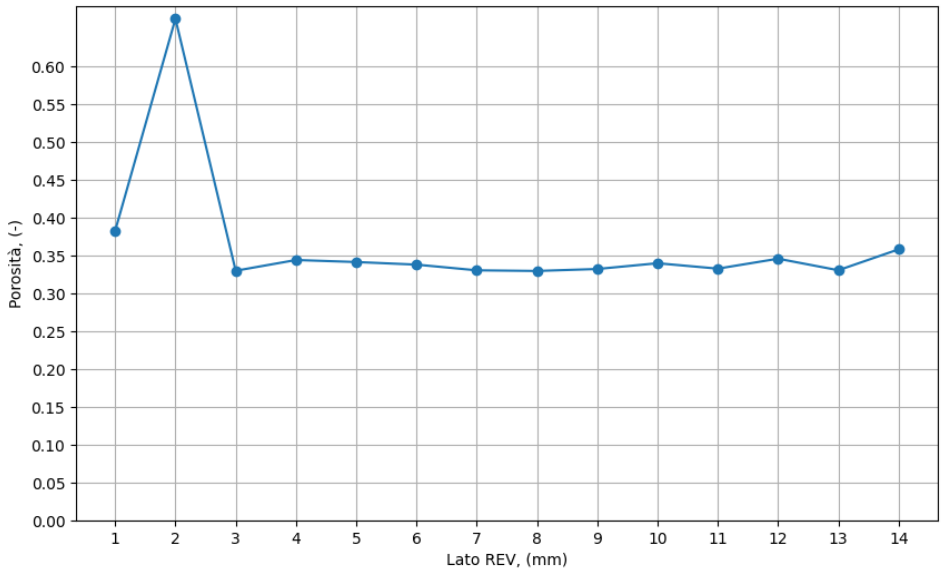


Figura 6.4: Studio REV per $\bar{H} = 2$ mm.

Valori più elevati di L mostrano una netta influenza degli effetti di bordo, dovuti alla *bounding box* contenente l'impaccamento, che risulta in un fenomeno oscillatorio ad ampiezza crescente. Ciò ricalca quanto riscontrato da Boccardo *et al.* [27] per varie geometrie d'impaccamento.

Valori più modesti, d'altro canto, non sarebbero statisticamente sufficienti ad inglobare un volume di controllo che possa essere una rappresentazione generalizzata di un impaccamento più esteso, pertanto sono esclusi anch'essi dal processo di scelta.

In linea generale, l'analisi delle tre casistiche mostra un sostanziale assestamento del valore di porosità ϵ in corrispondenza di un REV cubico di lato $L = 7 - 8$ mm. Il discriminante nella scelta fra questi due valori è stato dettato da una ragione prettamente computazionale, di cui in seguito si dà dettaglio:

- $L = 7$ mm, CPD = 18: 126^3 celle = 2.000.376 celle*
- $L = 8$ mm, CPD = 18: 144^3 celle = 2.985.984 celle*

(*identificate tramite la function `blockMesh`).

Nel caso con lato $L = 7$ mm, valore scelto, si avrebbe una riduzione di circa il 33% di celle rispetto al caso in cui $L = 8$ mm: considerando che, per ogni 1.000.000 di celle, sono necessari circa 10 GB di memoria, questo valore permette un risparmio di memoria allocata sulle risorse computazionali HPC di circa 10 GB.

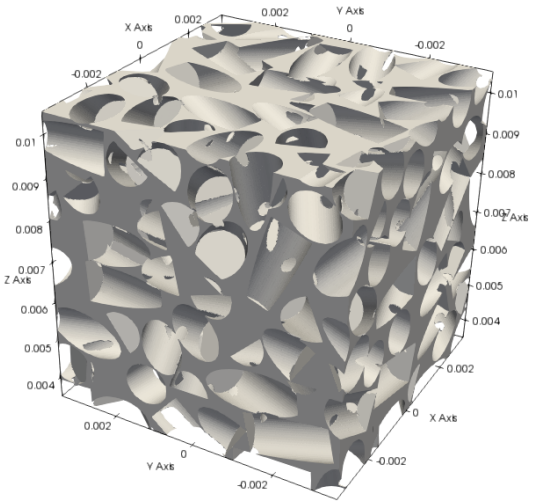


Figura 6.5: REV selezionato.

6.2 Analisi delle *feature*

In seguito all'esecuzione delle simulazioni CFD, si sono raccolti tutti i dati inerenti le *feature* di interesse (filtration rate e permeabilità).

Data la distribuzione dei valori ottenuti, ne è stato studiato l'andamento mediante un'interpolazione gaussiana, di cui si riportano i valori (Figure 6.6-7, Tabelle 6.2-3):

Parametro	**Valore**
Media $\bar{k}$	5.997×10^{-10} m^2
Deviazione standard σ	5.914×10^{-11} m^2
Coefficiente di variazione σ_{norm}	0.099

Tabella 6.2: Parametri di distribuzione gaussiana di permeabilità.

Parametro	**Valore**
Media $\bar{K_f}$	1.124×10^{-3} s^{-1}
Deviazione standard σ	3.467×10^{-4} s^{-1}
Coefficiente di variazione σ_{norm}	0.308

Tabella 6.3: Parametri di distribuzione gaussiana di filtration rate.

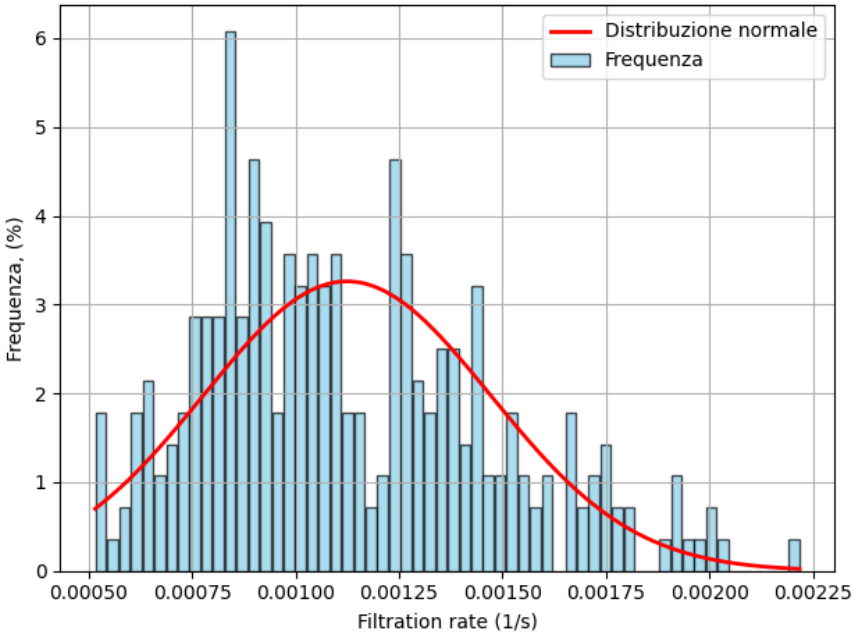


Figura 6.6: Distribuzione di filtration rate.

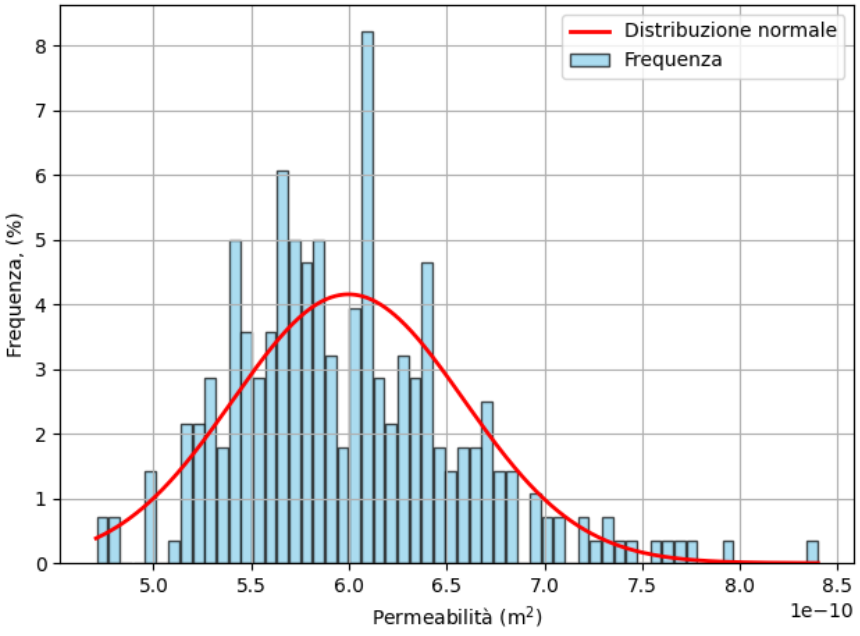


Figura 6.7: Distribuzione di permeabilità.

Stando ai valori del **coefficiente di variazione**, e valutando l'istogramma di distribuzione in sé, si percepisce una maggiore disomogeneità nella distribuzione dei dati di permeabilità, con una concentrazione più fitta di valori in corrispondenza del valore medio ed una riduzione più netta nella frequenza dei valori di coda. In altri termini, il dataset ottenuto rispetto alla velocità di filtrazione è più variabile rispetto alla sua media se comparato a quello di permeabilità.

Queste considerazioni sono essenziali se contestualizzate all'interno di un'analisi numerica (come ad esempio quella di *machine learning*), in cui l'andamento dei dati è di essenziale importanza. Un coefficiente di variazione basso suggerisce una scarsa variabilità nei dati, il che potrebbe rendere più difficile per i modelli apprendere pattern significativi, soprattutto in presenza di valori molto simili, nonostante lo step di *scaling* adottato nella fase di *pre-processing* dei dati.

6.3 Correlazioni fluidodinamiche

6.3.1 Confronto $\Delta P - Re$

Dai risultati CFD, è possibile svolgere alcune considerazioni circa il comportamento del sistema simulato, come ad esempio ricercare una correlazione numerica che possa esprimere una dipendenza fra la caduta di pressione nel mezzo poroso ed il regime di moto che si esplica al suo interno (in altri termini, il numero di Reynolds Re, già definito al paragrafo 1.4.3).

Questa indagine può essere effettuata facendo uso della **legge di Ergun**, una correlazione semi-empirica ampiamente riconosciuta e validata per descrivere la perdita di carico attraverso letti impaccati di grani solidi. La sua formulazione generale è espressa come:

$$\frac{\Delta P_p}{G_0^2}\frac{d_{32}^3}{L}\frac{\epsilon^3}{(1-\epsilon)} = 150\frac{(1-\epsilon)}{(d_{32}G_0)/\mu} + 1.75 \tag{6.1}$$

dove i parametri coinvolti hanno il seguente significato fisico e unità di misura:

- G_0 rappresenta la **velocità di flusso massica** [kg s^{-1}], definita come il prodotto tra la velocità di Darcy q (velocità apparente del fluido attraverso il mezzo poroso) e la densità del fluido ρ. Essa quantifica la massa di fluido che attraversa un'unità di sezione trasversale del letto poroso nell'unità di tempo;
- d_{32} [m] è il **diametro medio superficiale (di Sauter)** delle particelle che compongono il mezzo poroso;
- ϵ [-] è la **porosità** del mezzo. Essa influenza direttamente il percorso e la velocità del fluido attraverso il mezzo;

- μ [Pa s^{-1}] è la **viscosità dinamica** del fluido.

Al fine di semplificare l'analisi e rendere la correlazione più generale, è possibile definire i termini in forma adimensionale. Introducendo la caduta di pressione adimensionale ΔP^* e il numero di Reynolds Re^*, si perviene alla **legge di Ergun ridotta**:

$$\Delta P^* = \frac{150}{Re^*} + 1.75 \tag{6.2}$$

A livello grafico, si ottiene quanto visibile in Figure 6.8-9:

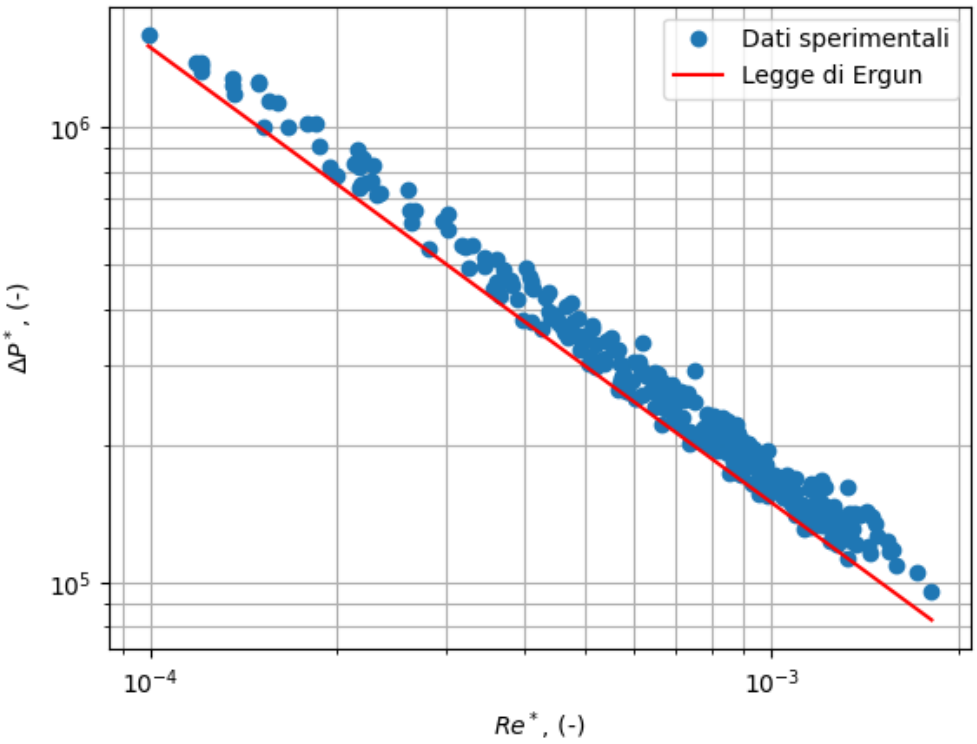


Figura 6.8: Legge di Ergun.

L'analisi del *fit* ottenuto rivela una tendenza che ricalca in modo molto simile i dati sperimentali, pur presentando un lieve scostamento rispetto alla loro dispersione. È interessante notare che questo tipo di comportamento non è inedito nella letteratura scientifica del settore. Ad esempio, Boccardo *et al.* [5] avevano già osservato un fenomeno simile e, per spiegarlo, avevano introdotto il concetto di *effective grain size* (dimensione effettiva del grano), denotato come d_{32}^* (anche riferito come d_g effettivo).

In particolare, la dimensione effettiva del grano d_{32}^* emerge dalla risoluzione analitica dell'Equazione 6.10 rispetto alla variabile diametro d_{32}. Essa rappresenta il valore reale che il grano dovrebbe possedere in termini di rapporto volume/area superficiale per minimizzare la discrepanza tra il *fit* e i dati sperimentali, rendendo la loro concordanza la più elevata possibile. Effettivamente, ciò trova conferma nel seguente risultato:

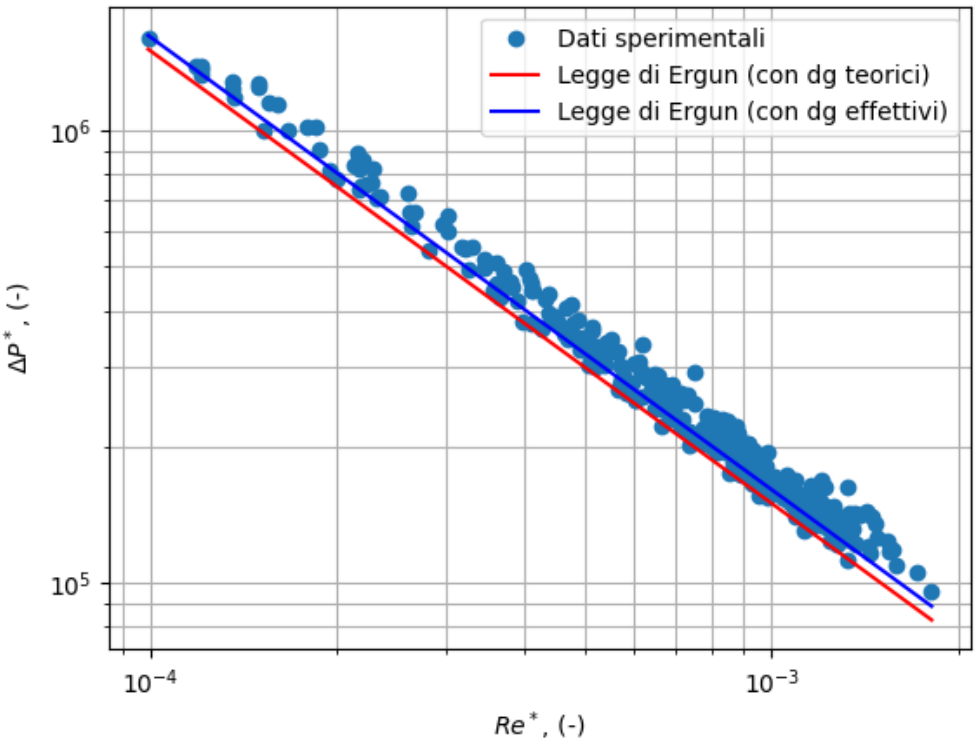


Figura 6.9: Legge di Ergun: caso teorico e caso effettivo.

con delle metriche di errore che sono riassunte nella Tabella 6.4.

Rapporto medio (d_{32}^*/d_{32})	0.93
Errore relativo medio	6.63%
Errore relativo massimo	17.38%
Deviazione standard	3.48%

Tabella 6.4: Confronto tra diametro del grano effettivo e misurato

In particolare, un'analisi più oculata dello scostamento dei diametri piò essere

svolta considerando il rapporto medio di diametro effettivo rispetto al teorico, e visualizzando poi il tutto su un diagramma di parità (rispettivamente Figura 6.10 e Figura 6.11).

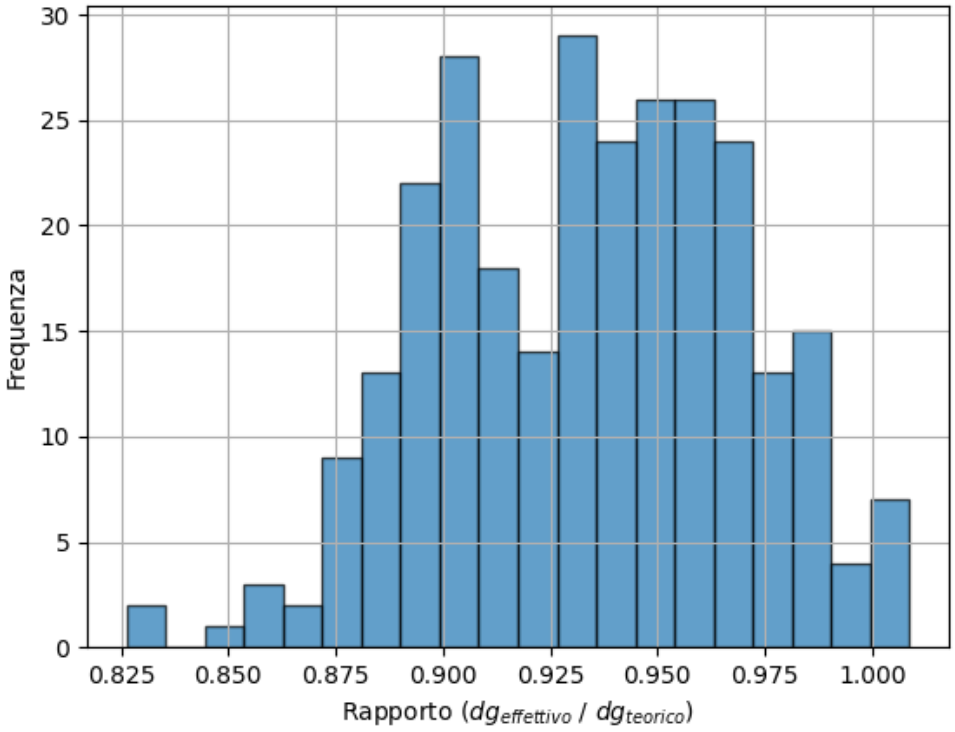


Figura 6.10: Distribuzione del rapporto di diametro effettivo - teorico (in termini di frequenza assoluta).

È ben visibile come, in circa il 93% dei casi analizzati, il diametro effettivo d_{32}^*, riferito alla legge di Ergun, tenda sistematicamente a sottostimare il diametro teorico d_{32} proprio dell'impaccamento stesso. Questa diffusa sottostima suggerisce che, per generare una caduta di pressione specifica (che in questo scenario risulta essere maggiore rispetto a quella prevista dal *fit* iniziale basato su d_{32}), sia generalmente richiesta una dimensione caratteristica del grano inferiore.

Tale osservazione implica che la microstruttura reale del mezzo poroso, con la sua complessa morfologia e l'eterogeneità nella dimensione e forma delle particelle, offre una maggiore resistenza al flusso rispetto a quella idealizzata da un letto avente grani con diametro d_{32}. Di conseguenza, un diametro effettivo d_{32}^* inferiore è necessario nel modello di Ergun per compensare queste discrepanze e predire con maggiore accuratezza la perdita di carico osservata.

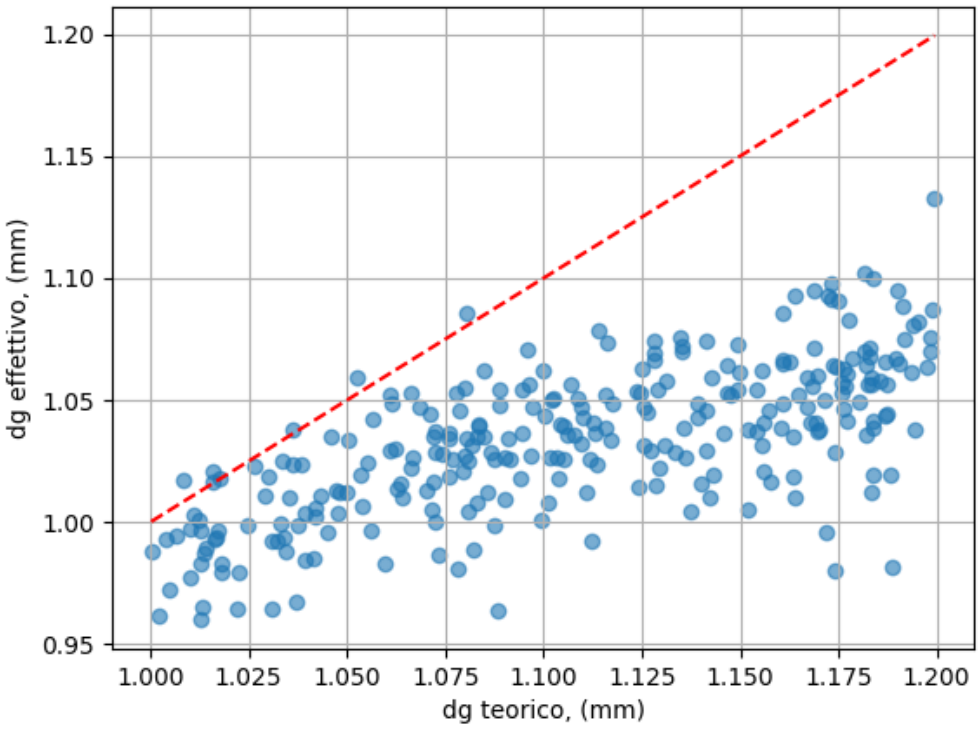


Figura 6.11: Diagramma di parità per i diametri.

6.3.2 Confronto $Pe - Da$

Un altro significativo confronto che è possibile compiere è quello relativo all'andamento del numero di Péclet (Pe) e del Damköhler (Da), anch'essi definiti in 1.4.3.

In un processo di filtrazione, essi sono concorrono alla sua efficacia complessiva. Un alto Pe indica una dominanza del trasporto convettivo sulla diffusione, mentre il Da paragona la velocità di filtrazione alla velocità di trasporto. In generale, per una filtrazione efficiente, è necessario bilanciare questi due numeri: un Da elevato è auspicabile per una rimozione rapida, ma un Pe non eccessivamente alto assicura un tempo di contatto sufficiente tra il fluido e il materiale filtrante per permettere alle interazioni chimico-fisiche di avvenire efficacemente, essendo il processo diffusivo prevalente su quello convettivo.

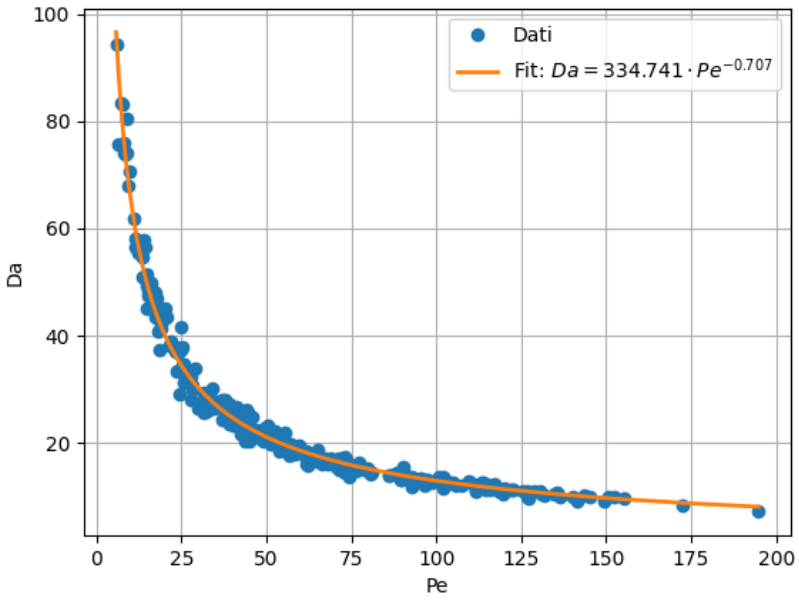


Figura 6.12: Andamento $Pe - Da$ (scala logaritmica).

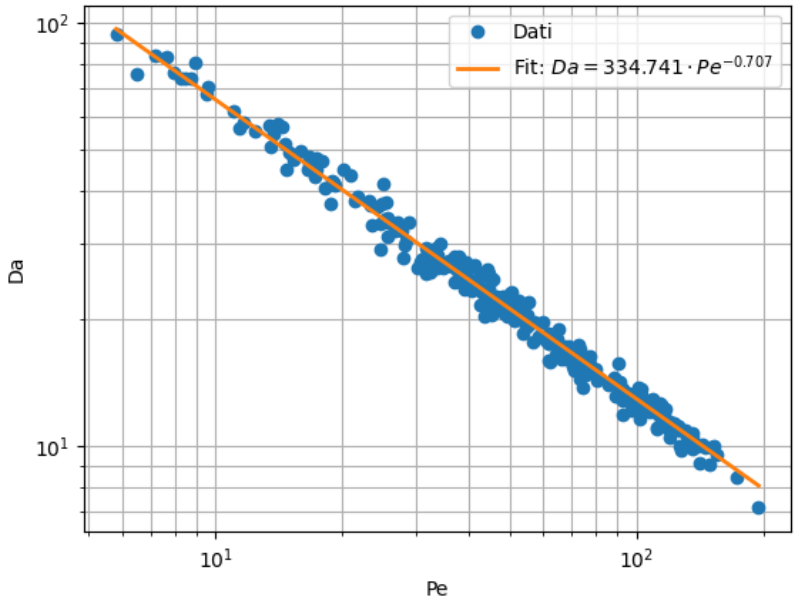


Figura 6.13: Andamento $Pe - Da$ (scala lineare).

I dati sperimentali, in scala logaritmica, sono stati interpolati tramite una relazione del tipo *power-law*, avente una forma del tipo:

$$y = Ax^b \tag{6.3}$$

dove y rappresenta la variabile dipendente (in questo caso, il numero Da), x è la variabile indipendente (numero di Pe), e A e b sono i parametri di fitting ottenuti dall'interpolazione dei dati. Ergo:

$$Da = APe^b \tag{6.4}$$

Il processo di interpolazione è stato effettuato utilizzando la libreria `Python scipy.optimize`, tramite la funzione `curve_fit`: questa minimizza la somma dei residui dei quadrati (RSS) per identificare il miglior accordo fra dati sperimentali e valori di fitting A e b.

I parametri ricavati sono:

$$A = 334.741$$
$$b = -0.707$$

con le seguenti metriche di errore (Tabella 6.5):

Metrica	**Valore**
Mean Squared Error (MSE)	3.57
Root Mean Squared Error (RMSE)	1.89
Errore medio	4.70%
Errore massimo	19.19%
Deviazione standard	3.84%

Tabella 6.5: Errori *fitting* $Pe - Da$.

6.4 Rete neurale *fully-connected* (FCNN)

Il primo step per attuare un'integrazione fra i risultati CFD ed il campo *machine learning* è stato quello di sviluppare una **rete neurale fully-connected** che possa predire il comportamento e il valore delle grandezze caratteristiche sulla base di soli parametri numerici, relativi a queste e alle rispettive geometrie cilindriche del mezzo poroso (qui ancora definibili data la regolarità dei grani d'impaccamento). Il linguaggio utilizzato in questa sede è `Pytorch`.

Sia per il filtration rate che per la permeabilità, si è adottata la seguente strategia operativa:

1. **pre-processing** dei dati: come da prassi consolidata nel *machine learning*, si è reso necessario applicare una fase di *scaling* dei dati. In particolare, si è optato per l'approccio *min-max* al fine di normalizzare i valori delle diverse feature all'interno di un intervallo predefinito (-1 e 1). La formula utilizzata per il *min-max scaling* è la seguente:

$$x_i' = \frac{x_i - \min(x)}{\max(x) - \min(x)}(max_{range} - min_{range}) + min_{range}$$

   dove:

   - x_i' rappresenta il valore scalato della feature x per l'i-esimo campione.
   - x_i è il valore originale della feature x per l'i-esimo campione.
   - $\min(x)$ è il valore minimo osservato per la feature x nell'intero dataset.
   - $\max(x)$ è il valore massimo osservato per la feature x nell'intero dataset.
   - min_{range} è il valore minimo desiderato per l'intervallo di scaling (in questo caso, -1).
   - max_{range} è il valore massimo desiderato per l'intervallo di scaling (in questo caso, 1).

   In questo modo, ogni valore della feature viene trasformato linearmente all'interno del nuovo intervallo specificato. Questa tecnica si rivela cruciale per garantire che feature con scale diverse non influenzino sproporzionatamente l'apprendimento del modello e per migliorare la convergenza degli algoritmi di ottimizzazione. Essendo poi i dati ricavati da modellazioni CFD, non si è proceduto ad una ricerca e rimozione degli *outliers* (che è un altro step standard nell'analisi preliminare dei dati);

2. **hyperparameters tuning** (ricerca dei migliori iperparametri): si è adottata una tecnica di analisi dei dati molto diffusa, denominata **k-fold cross validation** (Figura 6.14), che consiste nel suddividere il dataset di training in k

fold (o sottoinsiemi) disgiunti. Il modello viene addestrato per k iterazioni, dove in ciascuna iterazione un fold viene utilizzato come set di validazione per valutare le prestazioni del modello, addestrato sugli altri $k-1$ fold. Questo approccio permette di ottenere una stima più robusta delle performance del modello rispetto a una singola divisione train-validation, riducendo il rischio di overfitting e fornendo un'indicazione della generalizzabilità del modello su dati non visti, particolarmente utile nel caso di dataset non particolarmente estesi (come quello impiegato). Nel caso in esame, $k = 5$. Le varie combinazioni iperparametriche analizzate, pari a $N = 130$ e vertono sul numero di unità nascoste in cui la rete è strutturata (`hidden_units_list`) e il learning rate (`learning_rate`) che regola il processo di minimizzazione della funzione di perdita rispetto al set di validazione (*validation loss*).

```
param_grid = {
'hidden_units_list': [
    [32], [64], [128], [256],
    [32, 64], [32, 128], [32, 256], [64, 128],
[64, 256], [128, 256],
    [64, 32], [128, 32], [256, 32], [128, 64],
[256, 64], [256, 128],
    [32, 64, 128], [32, 64, 256], [32, 128, 256],
 [64, 128, 256],
    [128, 64, 32], [256, 64, 32], [256, 128, 32],
 [256, 128, 64],
    [32, 64, 128, 256],
    [256, 128, 64, 32],
],
'learning_rate': [1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
}
```

Pertanto, vengono allenate $N \times k$ (650) combinazioni in totale.

3. **post-processing**: per ogni combinazione di iperparametri, valutata tramite le k iterazioni della k-fold cross validation, vengono calcolati gli errori di predizione. In particolare, per ogni fold, si ottengono le predizioni e si calcola l'errore. Successivamente, per ogni metrica di errore (minimo, medio, massimo) e per la deviazione standard percentuale, si calcola la media dei valori ottenuti sui k fold. Analogamente, vengono registrati i valori medi della **validation loss** migliore e finale, anch'essi ottenuti mediando i rispettivi valori su tutte le k iterazioni della cross-validation.

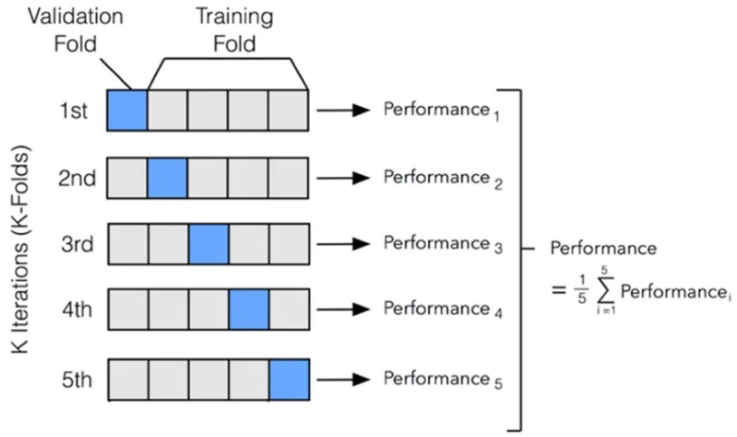


Figura 6.14: Processo di k-fold cross validation. [28]

Alla fine del processo, si ottiene un file testuale che sintetizza le prestazioni di ciascun modello sulla base dei parametri al punto 3. Per discriminare quale modello possa essere più idoneo al caso, è stato preso in considerazione, oltre che l'errore medio e massimo con relativa dispersione dei dati, anche la stabilità della funzione di perdita, intesa come minimizzazione della differenza fra valore dell'ultima *epoch* e miglior valore ottenuto nel corso del *training*.

Una volta identificato il caso ottimale, tale modello viene allenato al fine di generare un'inferenza sul dataset totale, randomicamente suddiviso nella canonica partizione *train* - *validation* - *test*, con proporzione 70 : 20 : 10.

6.4.1 Filtration rate

Le **input features** fornite alla rete, al fine di addestrarla e predire l'**output feature** in oggetto, sono le seguenti:

- **altezza media** $\bar{H}$;
- **coefficiente di variazione** σ_{norm} della distribuzione dei grani relativa a ciascun impaccamento;
- **pressione in ingresso** p_{in} (ovvero caduta di pressione);
- **coefficiente di diffusività di materia** $\mathcal{D}$;
- **porosità dell'impaccamento** ϵ.

Com'è possibile evincere, $\bar{H}$, σ_{norm} ed ϵ sono atte alla caratterizzazione del mezzo poroso, mentre $\mathcal{D}$ e p_{in} alle condizioni operative che regolano il processo di filtrazione e ritenzione del tracciante o inquinante in ingresso al letto impaccato.

La rete selezionata, identificata come *best case* per il range e le condizioni indagate è data da una struttura a *layers* crescenti [64, 128], avente un *learning rate* di 10^{-5} (Figura 6.15).

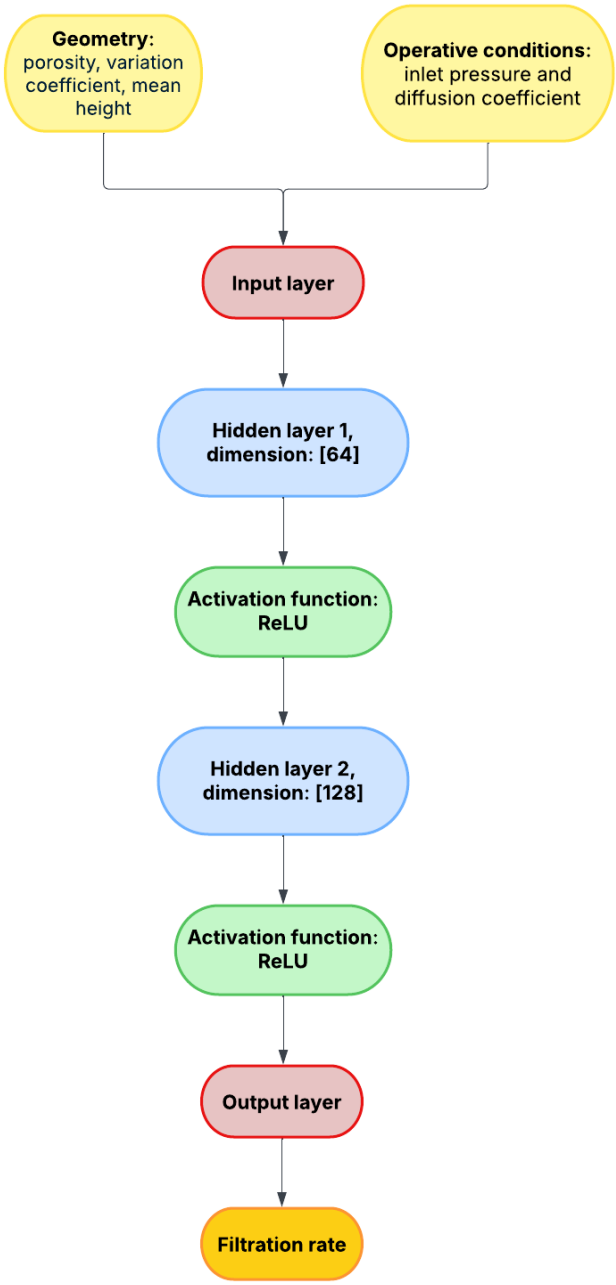


Figura 6.15: Rete FCNN per filtration rate.

Nella seguente Tabella 6.6, sono riassunti i valori associati alla valutazione finale **k-fold cross validation** del modello scelto:

Metrica	**Valore**
Best validation loss (MSE)	0.004019
Final validation loss (MSE)	0.003995
Errore medio	3.64%
Errore massimo	15.69%
Deviazione standard	3.30%

Tabella 6.6: Prestazione *k-fold cross validation* della rete (filtration rate).

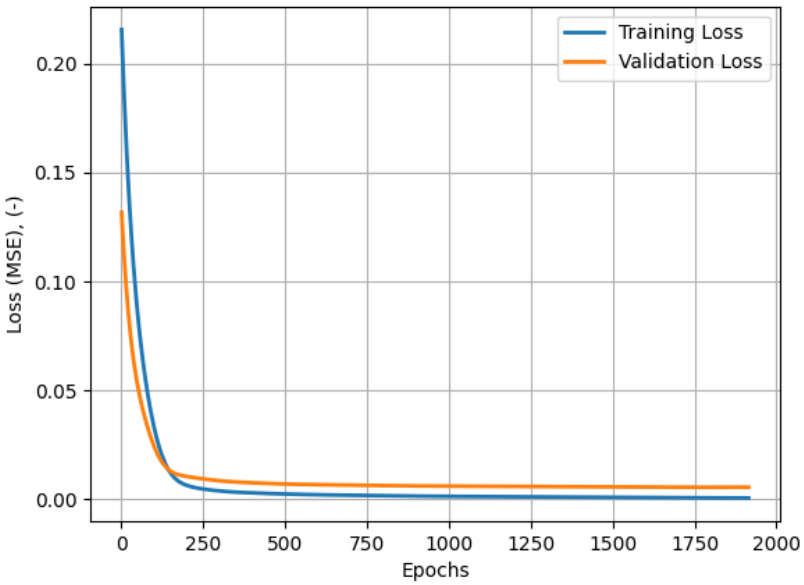
e quelle sul modello riallenato con lo splitting definito in precedenza (Tabella 6.7):

Metrica	**Valore**
Best validation loss (MSE)	0.005551
Final validation loss (MSE)	0.005602
Mean error	3.66%
Maximum error	11.75%
Standard deviation	3.19%

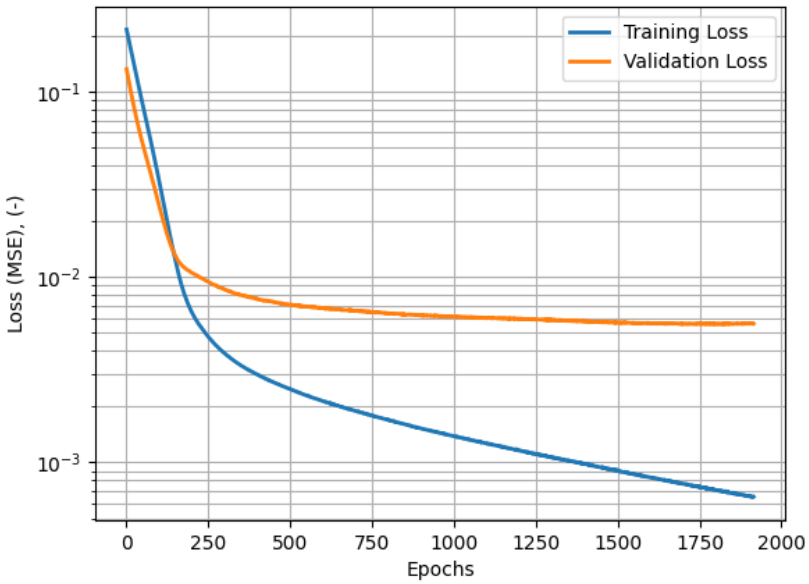
Tabella 6.7: Metriche sul *training* finale (filtration rate).

Le statistiche riportate mostrano una consistente coerenza tra il modello selezionato tramite *k-fold cross validation* e lo stesso riallenato sullo split rispetto al dataset generale. La *cross validation*, grazie alla sua natura più robusta, ha permesso di stimare in modo più oculato le prestazioni del modello, riducendo la dipendenza dalla particolare suddivisione dei dati. Le lievi differenze osservate (ad esempio nella *validation loss* finale o nell'errore massimo) sono plausibilmente attribuibili alla specifica composizione dei dati nei rispettivi set di validazione. Nel complesso, i risultati confermano la stabilità e l'efficacia del modello scelto.

Di seguito, l'andamento della *training* e *validation loss* con le *epochs* (Figura 6.16) ed il *parity diagram* risultante dall'inferenza finale (Figura 6.17):



(a) Andamento delle loss con le epoche (scala lineare).



(b) Andamento delle loss con le epoche (scala semi-logaritmica).

Figura 6.16: Andamento delle loss (filtration rate).

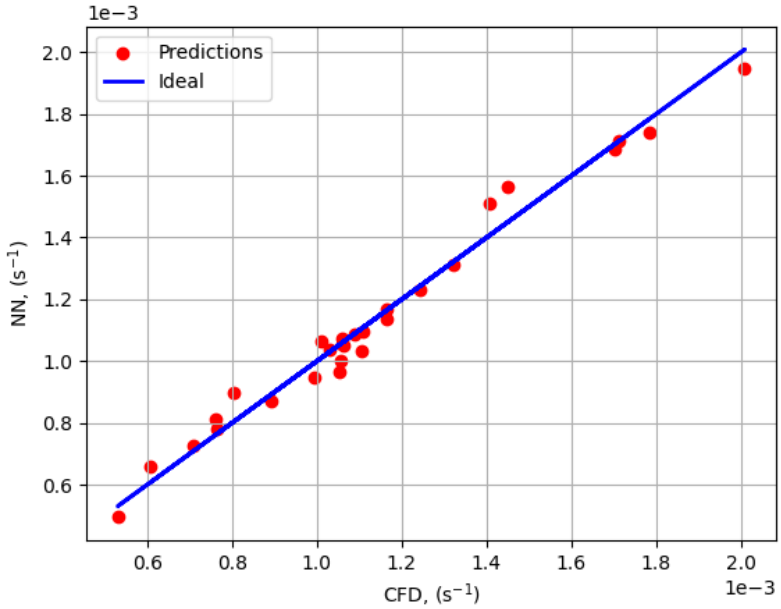


Figura 6.17: *Parity diagram* per filtration rate - FCNN.

6.4.2 Permeabilità

Le **input features** fornite alla rete sono le seguenti:

- **altezza media** $\bar{H}$;
- **coefficiente di variazione** σ_{norm} della distribuzione dei grani relativa a ciascun impaccamento;
- **porosità dell'impaccamento** ϵ.

A differenza del caso precedente, qui i parametri visti dalla rete sono inerenti *esclusivamente* alla geometria. Questo è dovuto al fatto che la permeabilità è indipendente dalle condizioni operative del sistema, essendo una grandezza caratteristica intrinseca del mezzo poroso stesso.

La rete selezionata, identificata come *best case* per il range e le condizioni indagate è data da una struttura a *layers* crescenti [32, 64], avente un *learning rate* di 10^{-5} (Figura 6.18).

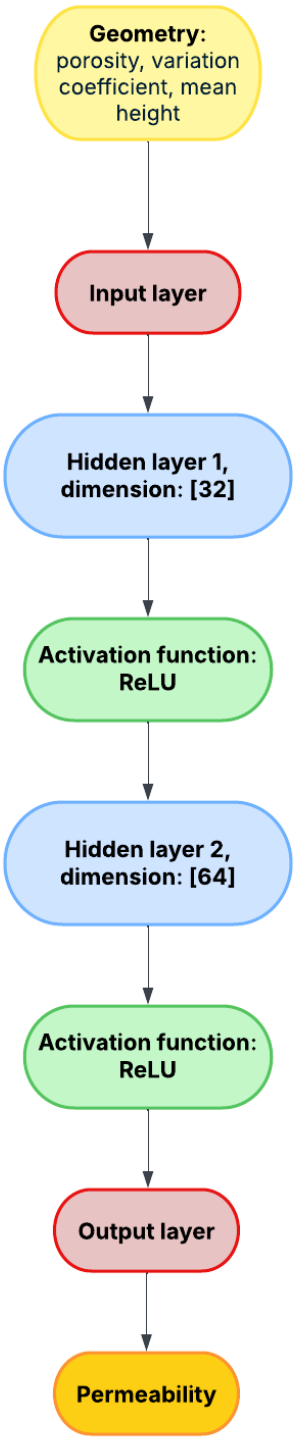


Figura 6.18: Rete FCNN per permeabilità.

Rispetto al filtration rate, dove la rete presentava un numero doppio di neuroni intermedi, in questo caso si è optato per una struttura più snella. Tale riduzione nella capacità della rete potrebbe essere motivata dalla semplicità intrinseca del problema, caratterizzato dall'utilizzo di sole **tre variabili predittive contro le cinque del modello precedente**.

Nella seguente Tabella 6.8, sono riassunti i valori associati alla valutazione finale **k-fold cross validation** del modello scelto:

Metrica	**Valore**
Best validation loss (MSE)	0.020608
Final validation loss (MSE)	0.020651
Errore medio	3.43%
Errore massimo	12.03%
Deviazione standard	2.73%

Tabella 6.8: Prestazione *k-fold cross validation* della rete (permeabilità).

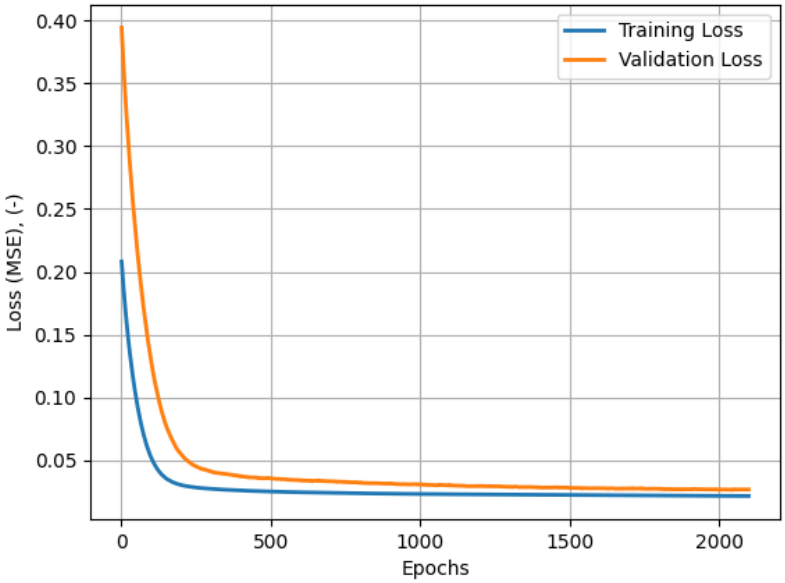
e quelle sul modello riallenato con lo splitting definito in precedenza (Tabella 6.9):

Metrica	**Valore**
Best validation loss (MSE)	0.026719
Final validation loss (MSE)	0.026952
Errore medio	3.00%
Errore massimo	9.40%
Deviazione standard	2.66%

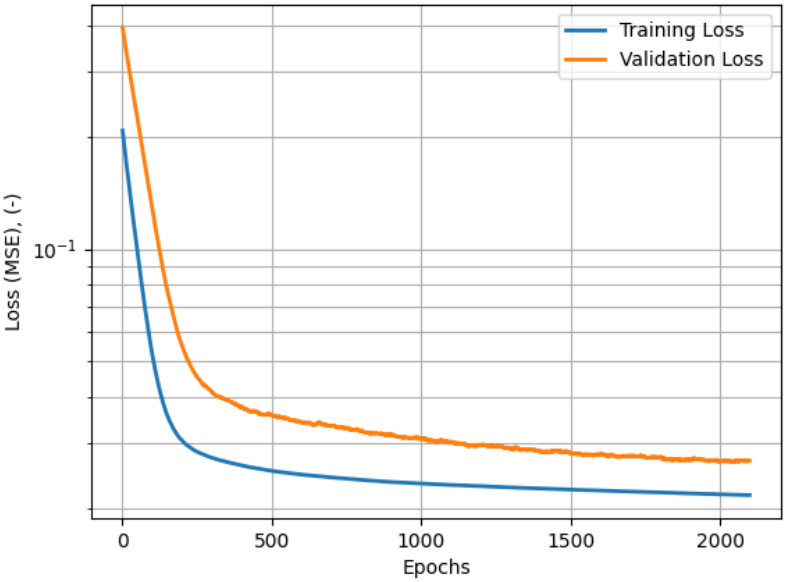
Tabella 6.9: Metriche sul *training* finale (permeabilità).

I dati mostrati evidenziano un accordo coerente tra il modello scelto tramite *k-fold cross validation* e lo stesso riallenato sullo split rispetto al dataset generale. Anche in questo caso, eventuali variazioni nelle metriche sono imputabili alla differente suddivisione del dataset in termini di *validation set*, che ciononostante non pregiudicano la bontà del modello stesso.

Di seguito, l'andamento della *training* e *validation loss* con le *epochs* (Figura 6.19) ed il *parity diagram* risultante dall'inferenza finale (Figura 6.20):



(a) Andamento delle loss con le epoche (scala lineare).



(b) Andamento delle loss con le epoche (scala semi-logaritmica).

Figura 6.19: Andamento delle loss (permeabilità).

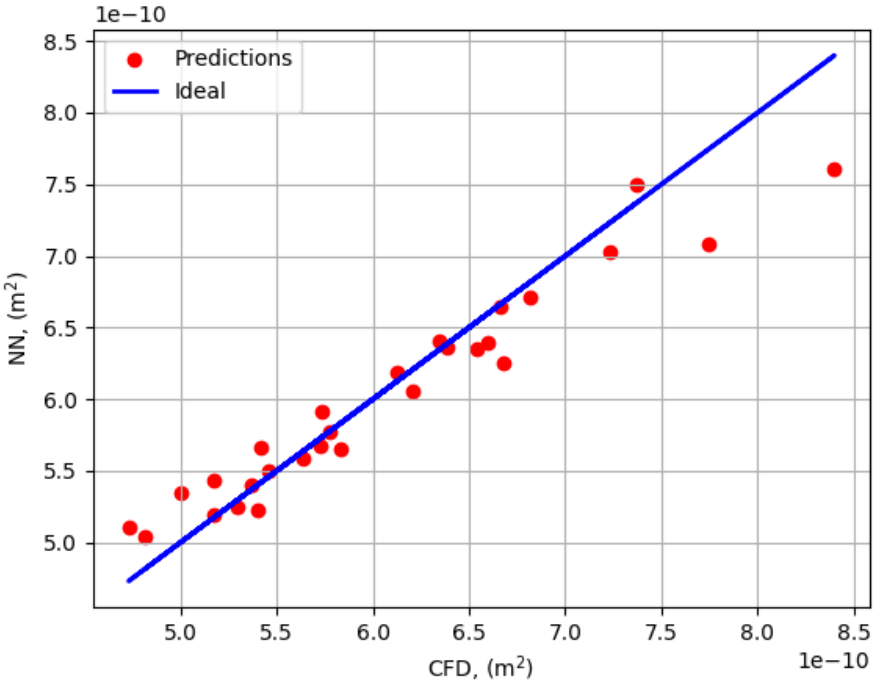


Figura 6.20: *Parity diagram* per permeabilità - FCNN.

6.5 Rete neurale convolutiva (CNN)

6.5.1 Dataset di geometrie sferiche

Lo step successivo consiste nell'elaborazione di due reti neurali convolutive, ciascuna dedicata a una delle due *features* di interesse. L'obiettivo finale è offrire una visione complessiva e generalizzata della tematica affrontata in questa tesi, superando i limiti della mera descrivibilità numerica della geometria. A tal fine, si adotta un approccio basato sull'integrazione di layer convolutivi (Paragrafo 3.4.3), capaci di ricevere la geometria in forma di tensore numerico voxelizzato (Paragrafo 5.2). In una fase iniziale del lavoro, è stato utilizzato `TensorFlow` per familiarizzare con l'ambiente di sviluppo e le dinamiche della modellazione neurale, mettendo a punto due architetture basate sul solo dataset di sfere [10]. Tuttavia, in una fase successiva, si è scelto di tradurre il tutto in linguaggio `PyTorch`, per sfruttarne l'approccio più flessibile e intuitivo e la maggiore diffusione in ambito *machine/deep learning* al giorno d'oggi, prima di procedere all'ampliamento del dataset.

Le reti identificate sono quelle raffigurate nelle Figure 6.21-22, con rispettivi risultati nelle Tabelle 6.10-6.11:

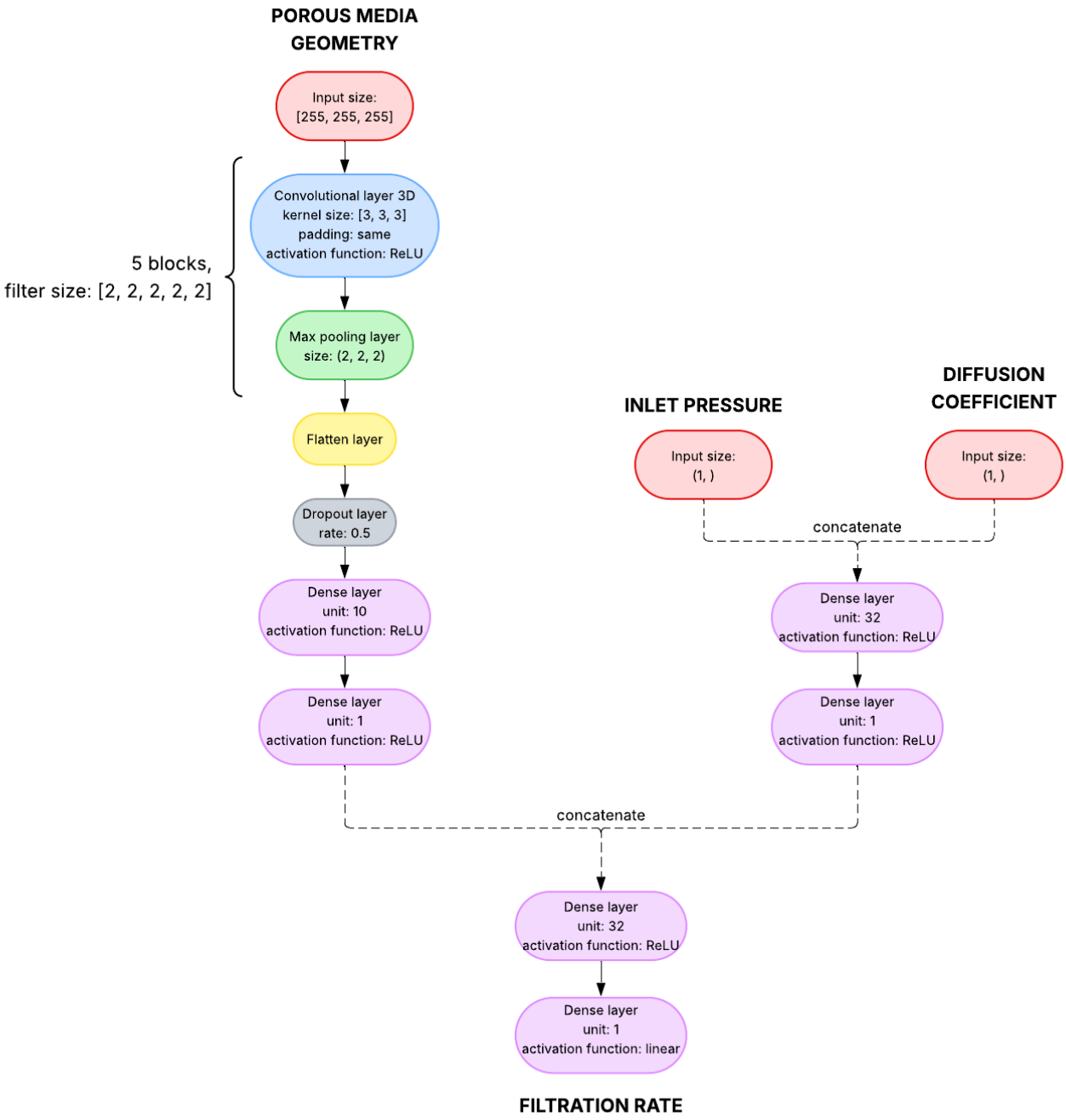


Figura 6.21: Rete neurale convolutiva per filtration rate.

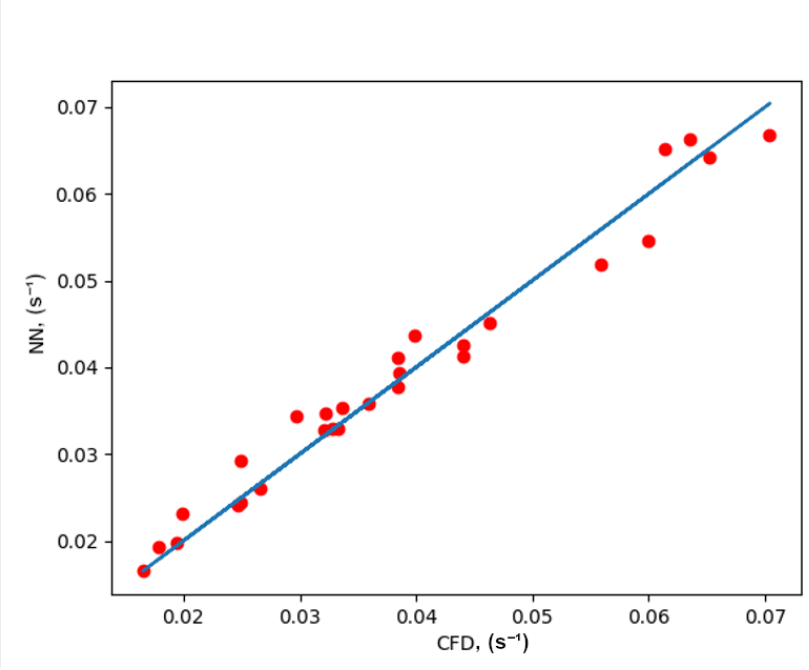


Figura 6.22: *Parity diagram* per filtration rate in TensorFlow - CNN.

Metrica	**Valore**
Errore medio	5.10%
Errore massimo	14.79%
Deviazione standard	4.09%

Tabella 6.10: Metriche di valutazione per filtration rate in TensorFlow - CNN.

Durante la fase di ottimizzazione dell'architettura della rete neurale proposta, è stato osservato che la riduzione del numero di filtri nei 5 blocchi convolutivi 3D (pari a 2, in questo caso) del ramo finalizzato all'elaborazione della geometria, ha indotto un incremento nelle performance del modello. Questo risultato suggerisce che alcune configurazioni iniziali testate, caratterizzate da un maggiore numero di filtri in tali blocchi, eccedevano la capacità rappresentativa necessaria, portando con molta probabilità a una condizione di *overfitting*. In questo contesto, il modello ha appreso in modo eccessivamente specifico i pattern presenti nel set di *training*, compromettendone la capacità di generalizzazione. Viceversa, un'architettura più snella e meno profonda, e quindi con un numero ridotto di parametri addestrabili nel ramo convolutivo, ha dimostrato maggiore efficacia nel bilanciare la complessità del modello con l'apporto degli input scalari di pressione e diffusività.

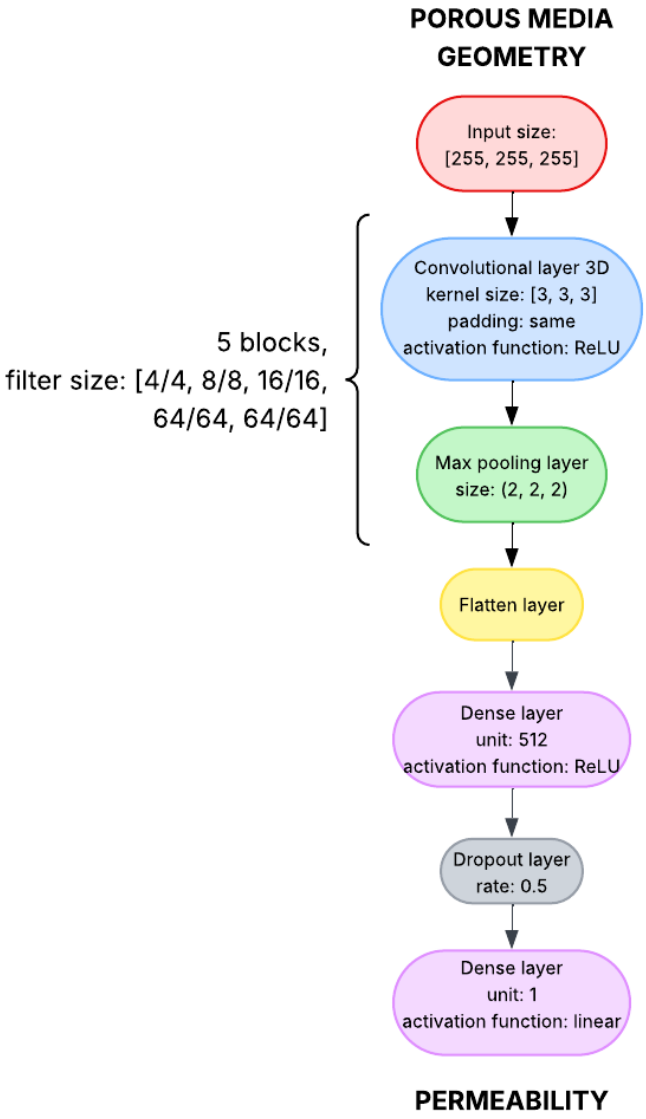


Figura 6.23: Rete neurale convolutiva per permeabilità.

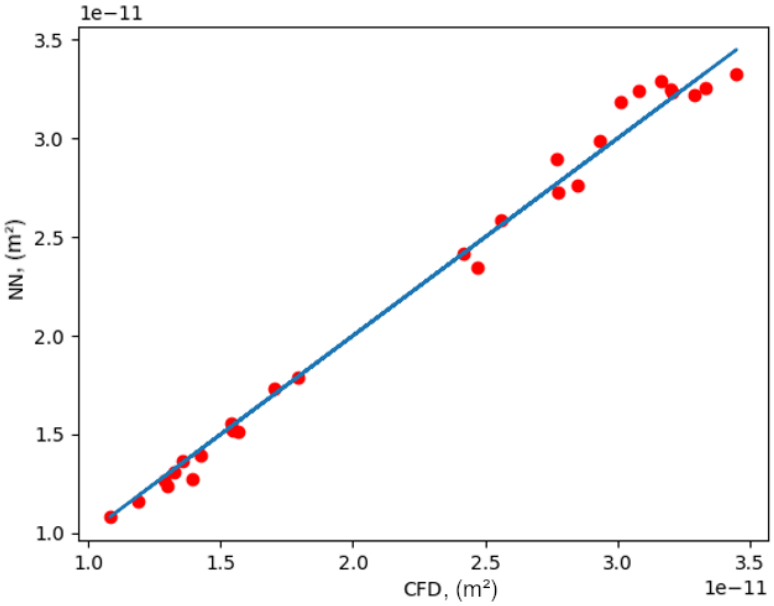


Figura 6.24: *Parity diagram* per permeabilità in TensorFlow - CNN.

Metrica	**Valore**
Errore medio	2.56%
Errore massimo	8.59%
Deviazione standard	1.98%

Tabella 6.11: Metriche di valutazione per permeabilità in TensorFlow - CNN.

Nel caso della permeabilità, la presenza di un solo ramo puramente convolutivo, e di un solo input che è quello geometrico, rende necessaria la presenza di un'architettura più complessa, data da cinque blocchi convolutivi con due filtri per ognuno, a dimensioni crescenti del tipo [4, 8, 16, 64, 64]. Questa incrementata capacità rappresentativa lungo la profondità della rete è essenziale per catturare pattern complessi che influenzano direttamente la permeabilità, quali la connettività dei pori, la tortuosità dei percorsi e la distribuzione spaziale di questi ultimi.

Successivamente, si è tradotta la rete in origine progettata in `Tensorflow` nel linguaggio `PyTorch`. Come indagine iniziale, si è analizzata l'influenza nella scelta dell'inizializzazione dei pesi delle reti CNN sulle performance in termini di inferenza finale. Infatti, come si vedrà in seguito, `Tensorflow` e `PyTorch` hanno

due diverse inizializzazioni di default per questi. Contestualmente, i *bias* della rete sono stati impostati al valore nullo.

Un **peso** (anche definito *weight*) è definito come parametro allenabile dalla rete durante il processo di apprendimento. Esso quantifica l'intensità e la direzione dell'influenza che un determinato input esercita sull'output di un neurone o di un filtro (Figura 6.25), e la sua ottimizzazione è fondamentale per guidare la rete verso la ricognizione quanto più precisa del pattern insito nei dati di input. Nelle formulazioni per inizializzare i pesi, l'influenza dell'input è quantificata dal numero di connessioni che il neurone ha con quelli del layer precedente (fan_{in}); analogamente, l'influenza sull'output è rappresentata dal numero di neuroni successivi connessi a quello d'interesse (fan_{out}).

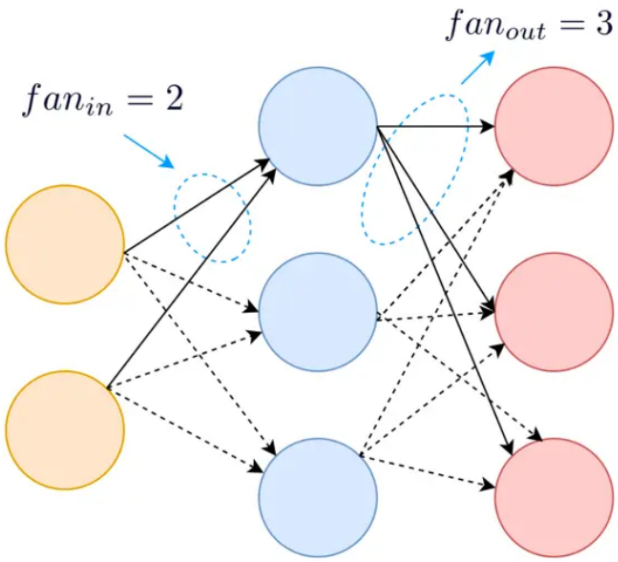


Figura 6.25: Concetto di *weight*. Fonte [29]

Per quanto concerne i pesi indagati, questi sono raccolti nella Tabella 6.12. Le inizializzazioni di default per i due linguaggi sono:

- **Xavier uniforme** (anche detto **Xavier-Glorot uniforme**) in `TensorFlow` per i layer convolutivi 3D [30] e per i layer *Dense* [31];

- in `PyTorch`, le distribuzioni di default non sono classificate sotto alcuna nomenclatura e sono quelle mostrate nella prima riga della Tabella 6.12 (per *Conv3D* [32], per layer *Dense* - qui riferiti come *Linear* [33]);

Tabella 6.12: Tecniche di inizializzazione dei pesi.

Nome	**Distribuzione**	**Formula di inizializzazione**
Default (`PyTorch`)	Uniforme	$w \sim \mathcal{U}\left(-\sqrt{k}, \sqrt{k}\right)$, dove k è:
		- *Linear*: $\frac{1}{fan_{in}}$
		- *Conv3D*: $\frac{\text{channels\_in}}{\text{groups}} \times \prod_{d=0}^{\text{dim}-1} \text{kernel}_{\text{size}}[\text{d}]$
Xavier Uniforme	Uniforme	$w \sim \mathcal{U}\left(-\sqrt{\frac{6}{fan_{in}+fan_{out}}}, \sqrt{\frac{6}{fan_{in}+fan_{out}}}\right)$
Kaiming Uniforme	Uniforme	$w \sim \mathcal{U}\left(-\sqrt{\frac{6}{fan_{in}}}, \sqrt{\frac{6}{fan_{in}}}\right)$
Kaiming Normale	Normale	$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{fan_{in}}}\right)$

Si ottengono i seguenti risultati, sia per filtration rate (Tabella 6.13) che per permeabilità (Tabella 6.14), con un'evidenziazione dell'inizializzazione scelta per ciascuno dei due casi:

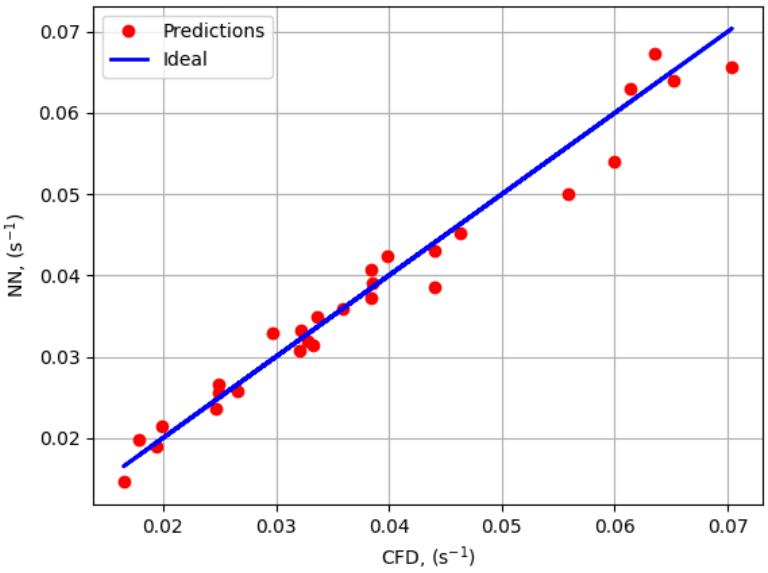
Tabella 6.13: Performance della rete CNN per inferenza del filtration rate.

Inizializzazione	**Errore medio**	**Errore massimo**	**Deviazione standard**	**Epoche**
Xavier uniform	5.20 %	13.57 %	3.77 %	811
Kaiming uniform	5.10 %	15.24 %	3.86 %	2,492
Kaiming normal	5.63 %	15.58 %	4.76 %	1,293
Kaiming normal (*Conv3D*) Xavier uniform (*Linear*)	**5.44 %**	**12.68 %**	**3.49 %**	**778**
Default	28.16 %	68.50 %	16.44 %	598

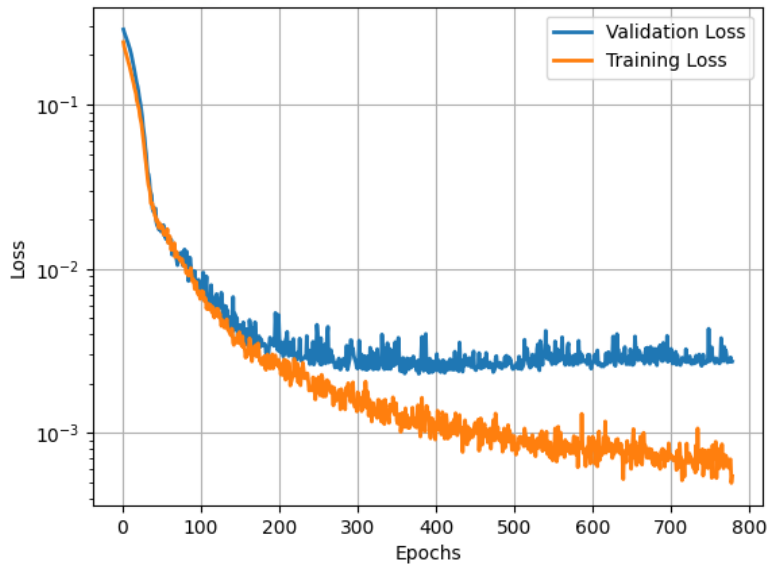
Tabella 6.14: Performance della rete CNN per inferenza della permeabilità.

Inizializzazione	**Errore medio**	**Errore massimo**	**Deviazione standard**	**Epoche**
Xavier uniform	**1.96 %**	**5.59 %**	**1.41 %**	**433**
Kaiming uniform	2.55 %	7.11 %	2.06 %	485
Kaiming normal	2.48 %	5.20 %	1.35 %	488
Kaiming normal (*Conv3D*) Xavier uniform (*Linear*)	2.62 %	6.78 %	2.11 %	485
Default	2.33 %	5.93 %	1.69 %	488

con i seguenti risultati (Figure 6.26-27):

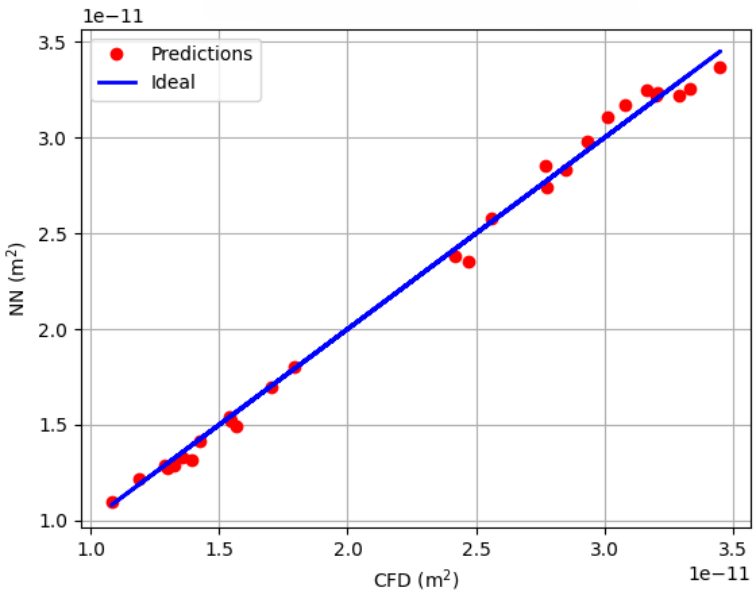


(a) *Parity diagram* per filtration rate.

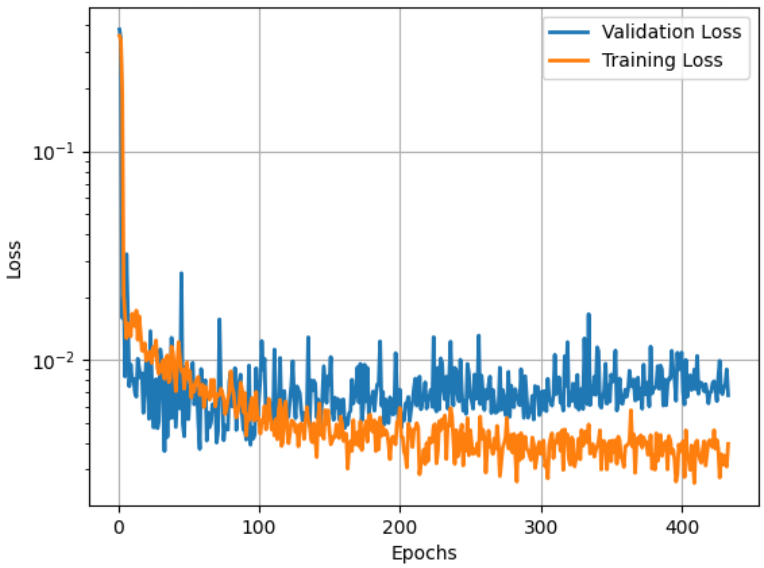


(b) Andamento delle loss con le epoche (scala semi-logaritmica).

Figura 6.26: Modello CNN relativo al filtration rate (dataset di sfere).



(a) *Parity diagram* per permeabilità.



(b) Andamento delle loss con le epoche (scala semi-logaritmica).

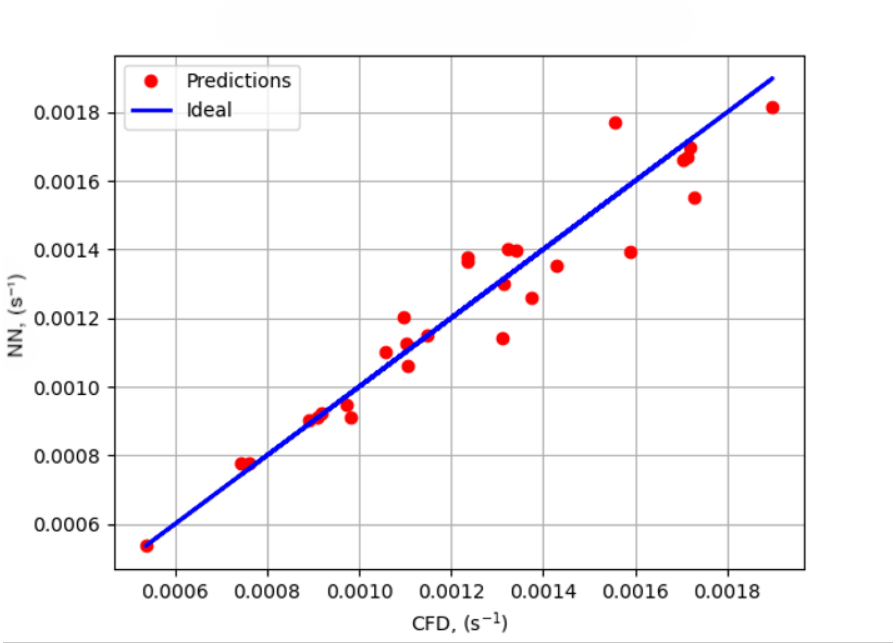
Figura 6.27: Inferenza tramite CNN relativa alla permeabilità (dataset di sfere).

6.5.2 Dataset di pellet

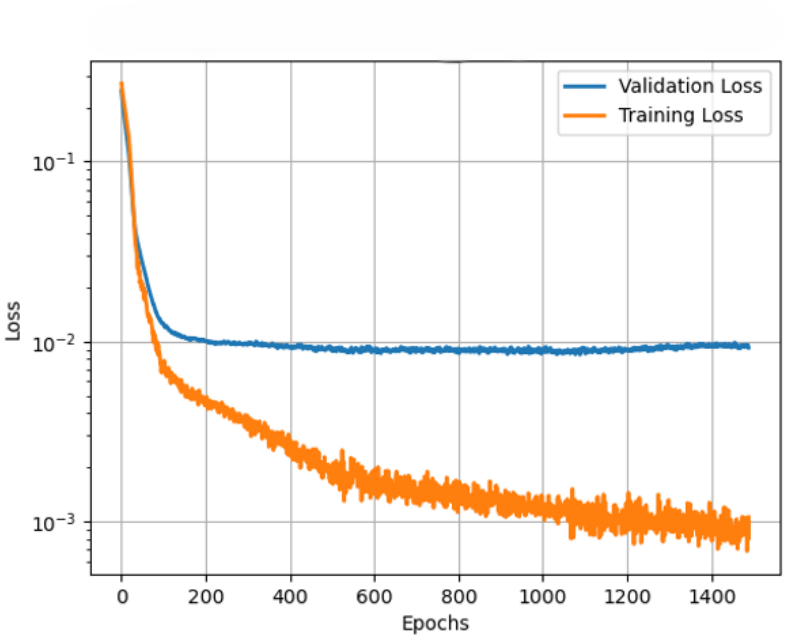
Una volta messe a punto delle strutture neurali efficaci e correttamente funzionanti, si è proceduto ad allenare e testarne l'efficienza su un dataset composto di sole geometrie cilindriche. Questo passo è di cruciale importanza verso una progressiva generalizzazione del metodo a geometrie di varia natura, obiettivo di questo percorso. Le metriche, rispettivamente per filtration rate e permeabilità, sono riassunte in Tabelle 6.15-16, mentre i dati sull'inferenza (diagrammi di parità) e dell'addestramento (*loss plot*) nelle Figure 6.27-28:

Metrica	**Valore**
Errore medio	5.22%
Errore massimo	13.85%
Deviazione standard	4.24%

Tabella 6.15: Prestazioni del modello CNN su dataset di soli cilindri - Filtration rate



(a) *Parity diagram* per filtration rate.



(b) Andamento delle loss con le epoche (scala semi-logaritmica).

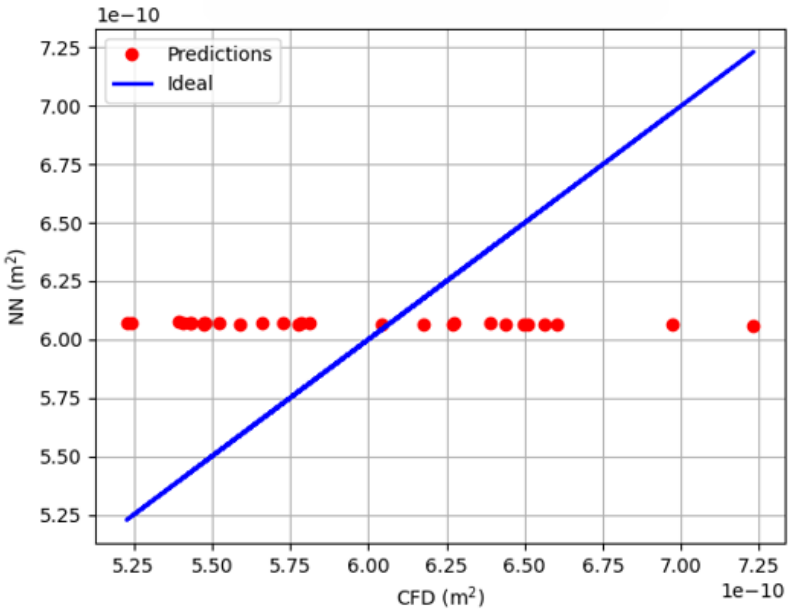
Figura 6.28: Modello CNN relativo al filtration rate (dataset di cilindri).

A vista d'occhio, si nota come le metriche di errore siano sostanzialmente coerenti con quanto riscontrato sull'analisi del dataset sferico, con circa un punto percentuale in più relativamente all'errore massimo.

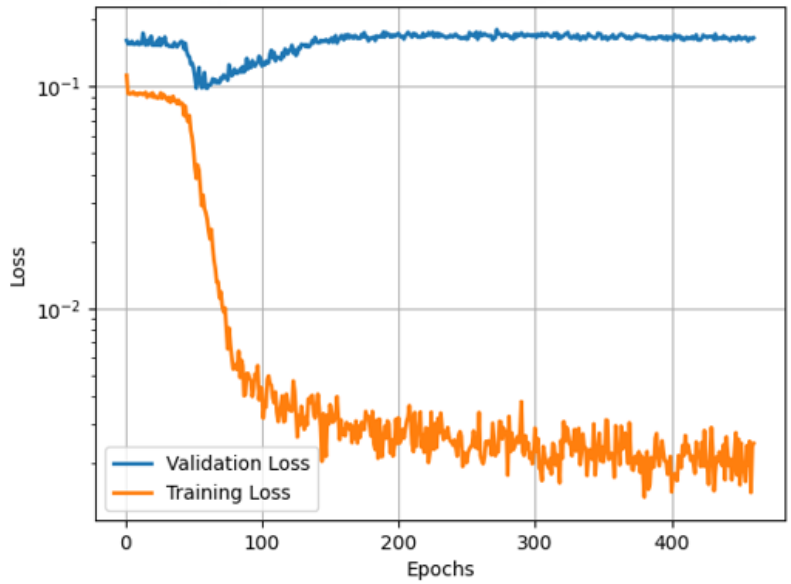
Tuttavia, ciò che funge da spunto di analisi è un sostanziale assestamento asintotico del valore della *loss* di validazione su valori compresi fra 9 $\times 10^{-3}$ - 1 $\times 10^{-2}$, circa 0.006 punti in più rispetto a quanto visibile in Figura 6.26b. Questo scostamento nel processo di apprendimento è sintomo di una minore capacità di generalizzazione della rete neurale sul secondo set di dati, a parità di struttura ed iperparametri.

Eppure, tale comportamento non è del tutto inaspettato: differenze numeriche nei dati che compongono i dataset ed i diversi pattern con cui si correlano fra loro possono inficiare in modo non trascurabile le prestazioni del modello, che è molto sensibile a caratteristiche di questo tipo.

Un'ulteriore conferma di questa sensibilità è data dai risultati ottenuti rispettivamente alla *feature* di permeabilità:



(a) *Parity diagram* per permeabilità.



(b) Andamento delle loss con le epoche (scala semi-logaritmica).

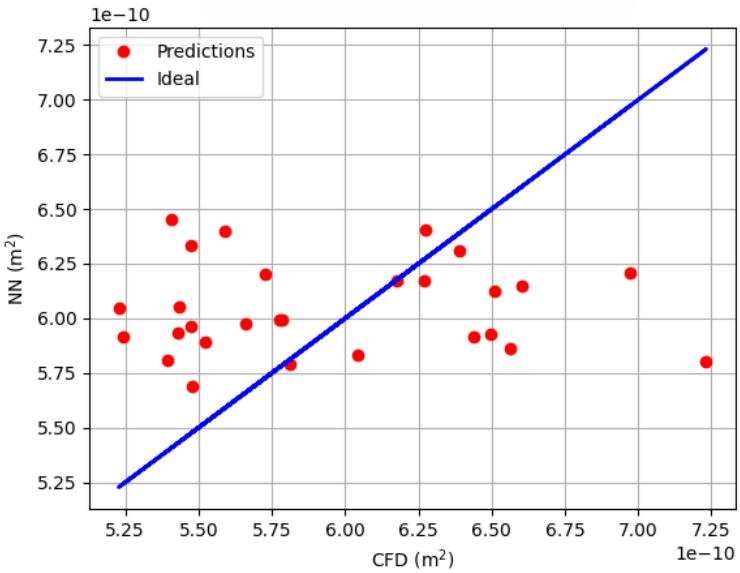
Figura 6.29: Inferenza tramite CNN relativa alla permeabilità (dataset di cilindri).

In questo caso, è evidente un fenomeno di sovradattameto della rete ai dati (il già noto *overfitting*), che ne satura la capacità di apprendimento dando luogo ad una predizione costante (Figura 6.29a), il cui valore coincide con quello della classe di distribuzione di permeabilità più rappresentata in termini statistici (si veda Figura 6.7) - ergo attorno a circa 6.10 $\times 10^{-10}$ m^2.

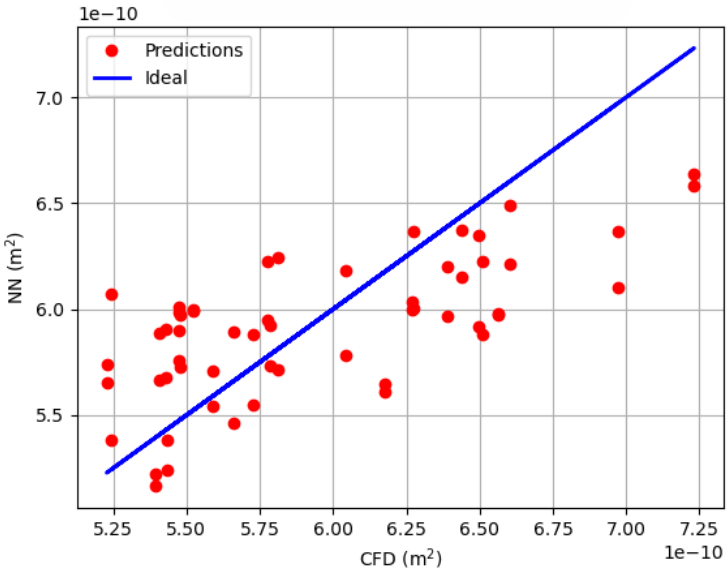
La conferma di *overfitting* deriva da una significativa differenza fra il valore di *training loss*, molto più bassa rispetto a quella di validazione (Figura 6.29b). In questo caso, l'algoritmo perde l'abilità di generalizzare a nuovi dati, con una conseguente inabilità di raggiungere un minimo di validazione - in termini di perdita - sufficientemente modesto da garantire una predizione soddisfacente.

Per valutare la possibilità di attivare correttamente la rete in modo da ottenere un'inferenza non costante, è stata considerata l'applicazione di tecniche di *machine learning* note come **data augmentation**, ovvero processi di pre-elaborazione che consentono la generazione di nuovi dati sintetici a partire da un dataset esistente. In particolare, tali dati derivano da modificazioni controllate degli esempi originali.

Nel caso della rete CNN oggetto di analisi, l'*augmentation* del dataset ha previsto operazioni di rotazione e *flipping* randomizzate delle geometrie voxelizzate (come suggerito da *Goodfellow et al.* nel Capitolo 7.4 del testo *Deep Learning*, [34]), mantenendo comunque l'associazione con le feature numeriche iniziali. Si è investigato sia il caso di una mera modificazione (Figura 6.30a) sia di una duplicazione della geometria con conseguente modificazione (Figura 6.30b):



(a) *Parity diagram* per caso *data augmented* senza duplicazione.



(b) *Parity diagram* per caso *data augmented* con duplicazione.

Figura 6.30: Risultati di *data augmentation*.

Come riscontrabile da entrambi i diagrammi, tuttavia, non si è ottenuto alcun significativo miglioramento rispetto alla situazione di partenza. Questo comportamento supporta l'ipotesi di una causa insita nel dataset stesso e nella distribuzione comunque ravvicinata dei valori di permeabilità. Inoltre, a ben vedere, l'applicazione di un processo di *data augmentation* può introdurre delle variazioni, seppur non statisticamente significative, nel valore di permeabilità associato alla geometria. Questo effetto è dovuto al fatto che la permeabilità risente della direzione del flusso rispetto agli assi e ai versi del volume elementare (non è spazialmente invariante), e dunque può risultare sensibile a trasformazioni spaziali come rotazioni o riflessioni.

6.5.3 Dataset misto

L'ultimo stadio di questo elaborato ha coinvolto il mescolamento del dataset di grani sferici e pellet cilindrici, utilizzando le due reti inizialmente messe a punto per le sole sfere. Tuttavia, in presenza di una variazione sia nell'ampiezza che nella caratterizzazione del dataset, eterogeneo rispetto a quello inizialmente usato per le sfere, si è resa necessaria una revisione degli iperparametri quali *batch size* (numero di campioni processati assieme prima che la rete aggiorni i pesi) e *learning rate*, secondo un processo definito **hyperparameters tuning**. I risultati di questo sono

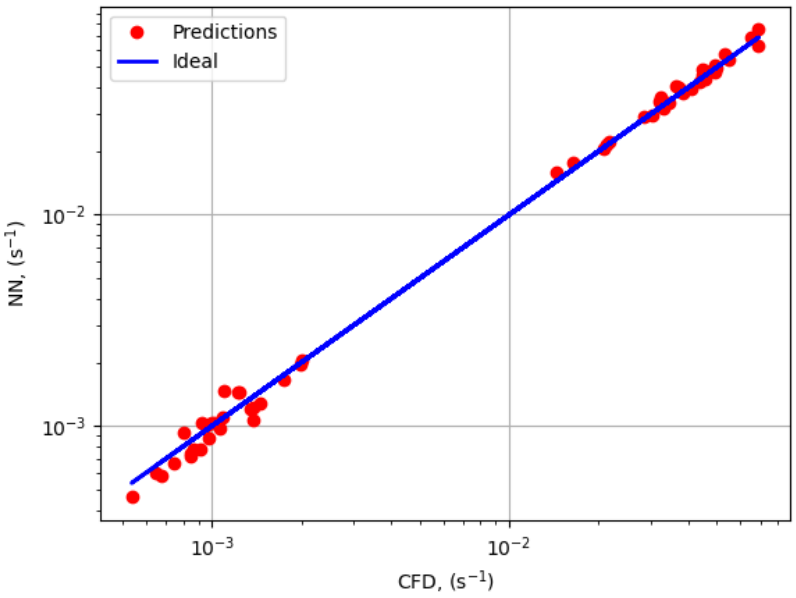
stati sinteticamente raccolti nelle due Tabelle seguenti (6.16-17), con evidenziati i casi che meglio accordano tutti i parametri presi in considerazione:

Tabella 6.16: *Tuning* degli iperaparametri - Filtration rate

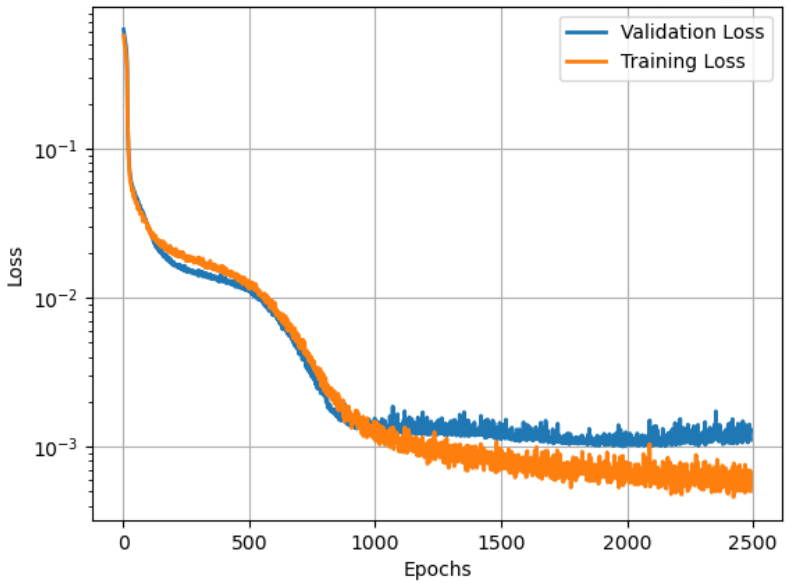
Learning rate	**Metriche**	**Batch size**			
		5	**10**	**15**	**18**
10^{-4}	Errore medio	6.93%	9.51%	8.42%	**7.78%**
	Errore massimo	38.68%	48.12%	60.89%	**32.96%**
	Deviazione standard	7.57%	10.30%	9.99%	**6.27%**
	Epoche	1651	1964	2550	**2491**
10^{-5}	Errore medio	13.80%	7.81%	8.50%	14.82%
	Errore massimo	111.45%	39.72%	44.47%	94.28%
	Deviazione standard	20.00%	7.75%	8.56%	20.67%
	Epoche	6701	9247	11984	13119
10^{-3}	Errore medio	9.30%	11.27%	8.85%	10.62%
	Errore massimo	45.04%	52.77%	50.30%	69.39%
	Deviazione standard	9.65%	12.11%	9.57%	12.43%
	Epoche	611	627	952	598

Tabella 6.17: *Tuning* degli iperaparametri - Permeabilità

Learning rate	**Metriche**	**Batch size**			
		5	**10**	**15**	**18**
10^{-5}	Errore medio	10.32%	13.12%	6.50%	7.55%
	Errore massimo	36.98%	56.90%	19.87%	30.79%
	Deviazione standard	8.65%	14.47%	5.24%	7.33%
	Epoche	700	960	1602	1426
10^{-6}	Errore medio	5.22%	**3.78%**	5.76%	3.68%
	Errore massimo	14.77%	**12.58%**	19.29%	12.87%
	Deviazione standard	3.74%	**2.48%**	4.53%	3.02%
	Epoche	5614	**8133**	9978	9697
10^{-4}	Errore medio	22.01%	31.55%	26.59%	19.53%
	Errore massimo	65.87%	139.54%	115.48%	58.70%
	Deviazione standard	21.90%	38.44%	30.93%	16.93%
	Epoche	432	426	457	469

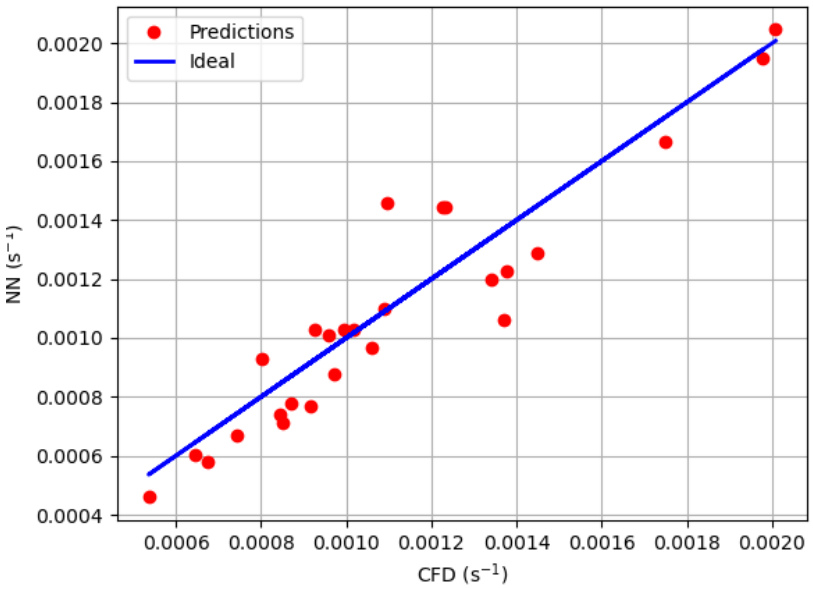


(a) *Parity diagram* generale in scala logaritmica.

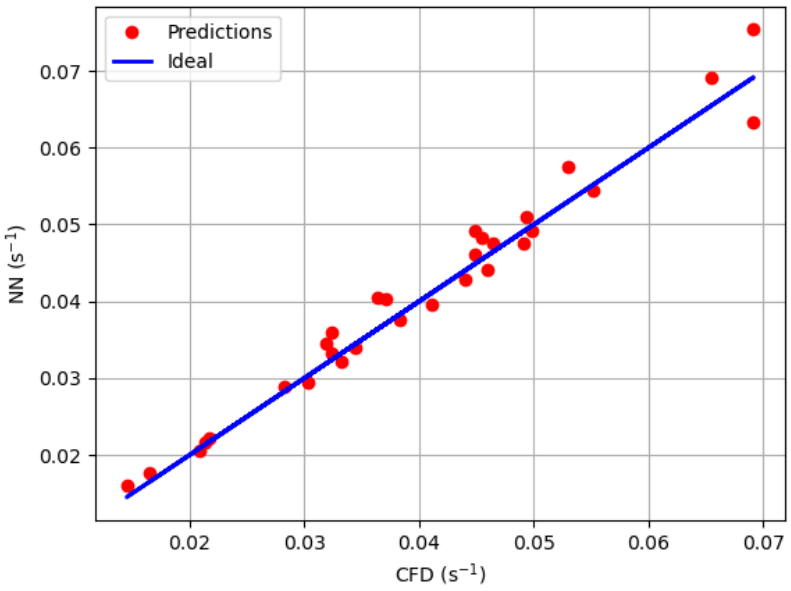


(b) Andamento delle loss con le epoche (scala semi-logaritmica).

Figura 6.31: Inferenza sul dataset generale relativa al Filtration rate.

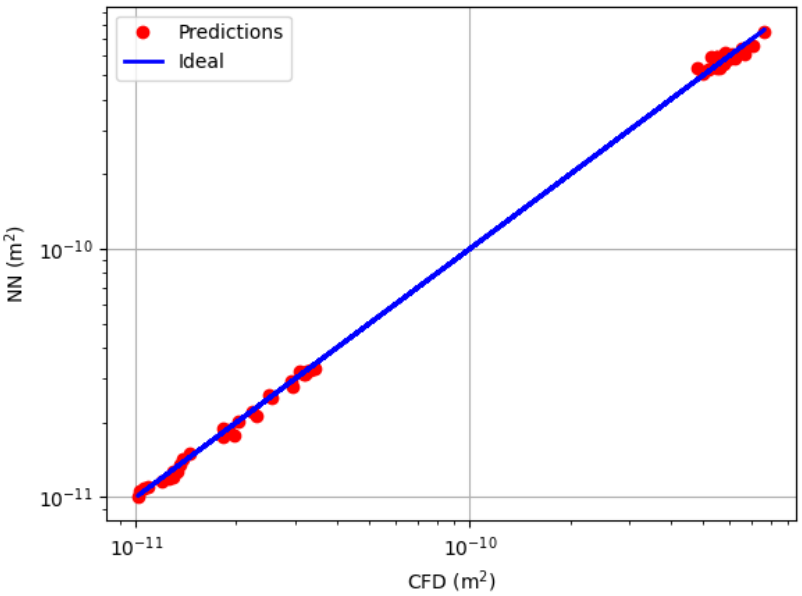


(a) Predizioni nel range dei cilindri.

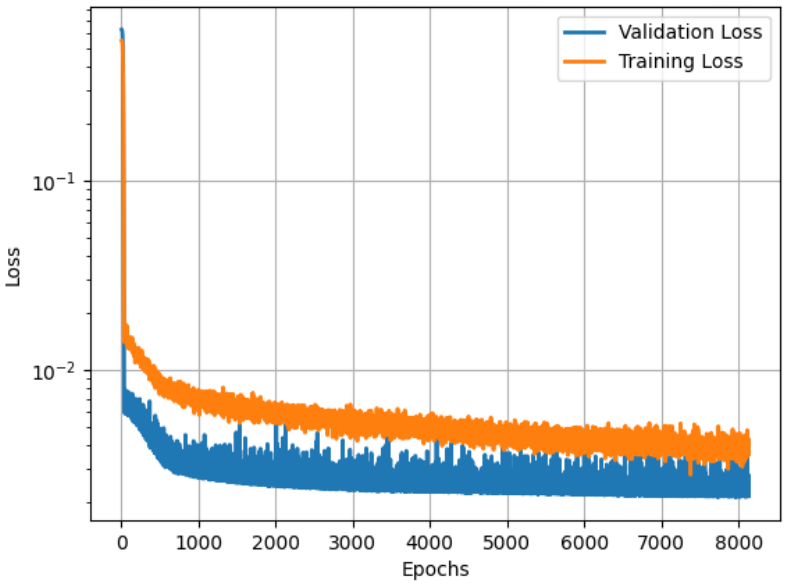


(b) Predizioni nel range delle sfere.

Figura 6.32: Zoom sui due intervalli geometrici delle predizioni - Filtration rate.

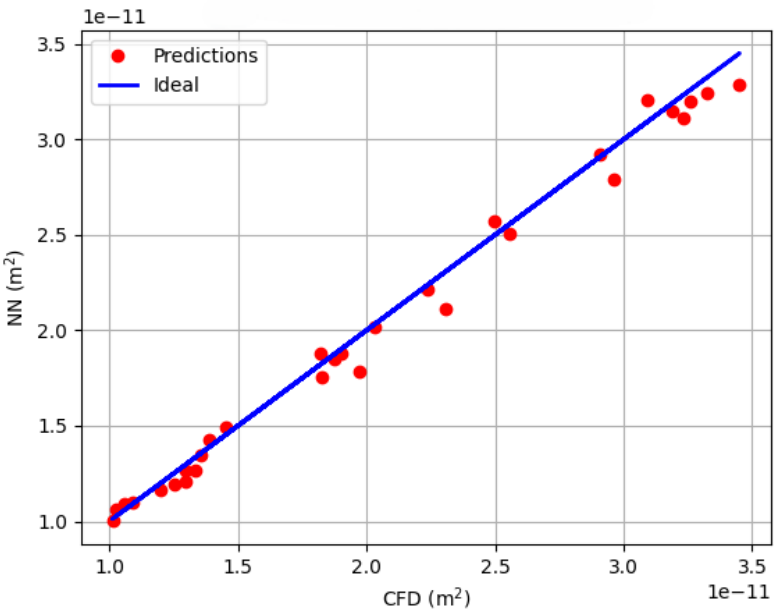


(a) *Parity diagram* generale in scala logaritmica.

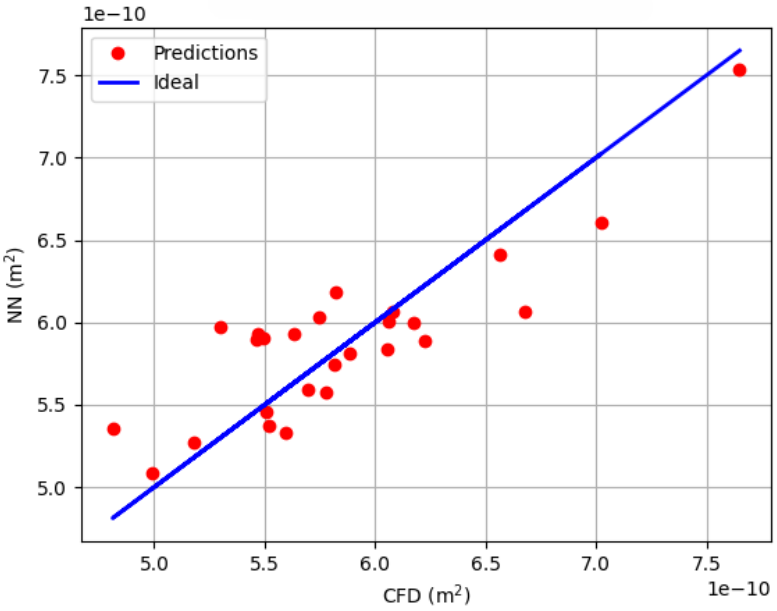


(b) Andamento delle loss con le epoche (scala semi-logaritmica)..

Figura 6.33: Inferenza sul dataset generale relativa alla permeabilità.



(a) Predizioni nel range delle sfere.



(b) Predizioni nel range dei cilindri.

Figura 6.34: Zoom sui due intervalli geometrici delle predizioni - Permeabilità.

Come visibile dalla Figura 6.34b, la situazione nella predizione dei valori di permeabilità riguardante i cilindri sembra essere nettamente migliorata rispetto alla predizione costante riscontrata inizialmente. Ciò mette sicuramente in rilevanza il ruolo giocato dalla controparte sferica (Figura 6.34a) nel garantire una maggiore variabilità geometrica e numerica del dataset, fattore cruciale per attivare ed ottimizzare la rete neurale, sia in fase di addestramento che di inferenza.

D'altra parte, i risultati avuti per la predizione del filtration rate non sono affatto inaspettati: le potenzialità della rete di funzionare in modo soddisfacente sia utilizzando un dataset di sole sfere che di soli pellet è quanto riscontrato nei precedenti paragrafi, con un'ulteriore riprova in questa sede (Figure 6.31-32).

Capitolo 7

Conclusioni

Al termine di questo lavoro, è possibile tratte alcune considerazioni generali circa quanto svolto.

Coerentemente con gli obiettivi iniziali, l'inserimento di uno step di *machine* e *deep learning* ha agevolato notevolmente la predizione delle caratteristiche desiderate senza ricorrere allo svolgimento di simulazioni CFD, portando alla costruzione di un dataset eterogeneo di geometrie sferiche e cilindriche e di modelli flessibili e robusti. Pur essendo caratterizzate ancora da una certa regolarità nella descrivibilità della loro forma solida, queste possono certamente essere un punto di partenza futuro per l'analisi altri range e altri formati geometrici. In particolare, il comportamento predittivo costante della permeabilità osservato nel solo dataset di pellet cilindrici (Figura 6.29a) - e la conseguente distribuzione concentrata dei valori - è con ogni probabilità attribuibile al fatto che il diametro dei cilindri è stato per scelta mantenuto fisso a 1 mm. Questo vincolo geometrico riduce i gradi di libertà della variazione spaziale, rendendo gli impaccamenti meno variabili rispetto al caso delle sfere, in cui la modifica del diametro comporta una variazione simultanea lungo tutte e tre le dimensioni.

Tra le possibili alternative geometriche, alcune delle più complesse potrebbero prevedere l'utilizzo di geometrie irregolari, come frammenti (ad esempio *char*) o materiali schiumosi. In tali casi, l'approccio *fully-connected* risulterebbe con ogni probabilità inapplicabile, a causa dell'impossibilità di definire una parametrizzazione geometrica descrittiva.

Le reti CNN implementate, d'altra parte, potrebbero dover subire degli adattamenti in termini di iperparametri - o, ancor più radicalmente, nell'assetto dei layer convolutivi - per adeguarsi a questi nuovi scenari geometrici.

Ulteriori sviluppi potrebbero coinvolgere anche l'analisi di nuove *features* oltre a filtration rate e permeabilità, come la simulazione di casi reattivi (velocità di reazione e campi di concentrazione delle specie coinvolte) e trasporto di energia (trasporto di calore e campi di temperatura, ad esempio).

Ciononostante, tutti questi sviluppi prospettivi dovranno necessariamente tenere conto della disponibilità delle risorse di *high-performance computing* impiegate: un'estensione del dataset e delle features esplorate vorrebbero dire un onere maggiore per quanto concerne il costo computazionale e il carico di lavoro.

Appendice A

Reti neurali convolutive

A.1 Permeabilità

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self, input_shape0, batch_norm):
        super().__init__()
        self.batch_norm = batch_norm

        self.conv1_1 = nn.Conv3d(input_shape0[0], 4, kernel_size
   =3, padding=1)
        self.conv1_2 = nn.Conv3d(4, 4, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool3d(kernel_size=2)
        self.bn1 = nn.BatchNorm3d(4) if self.batch_norm else nn.
   Identity()

        self.conv2_1 = nn.Conv3d(4, 8, kernel_size=3, padding=1)
        self.conv2_2 = nn.Conv3d(8, 8, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool3d(kernel_size=2)
        self.bn2 = nn.BatchNorm3d(8) if self.batch_norm else nn.
   Identity()

        self.conv3_1 = nn.Conv3d(8, 16, kernel_size=3, padding=1)
        self.conv3_2 = nn.Conv3d(16, 16, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool3d(kernel_size=2)
        self.bn3 = nn.BatchNorm3d(16) if self.batch_norm else nn.
   Identity()

        self.conv4_1 = nn.Conv3d(16, 64, kernel_size=3, padding=1)
```

```
        self.conv4_2 = nn.Conv3d(64, 64, kernel_size=3, padding=1)
        self.pool4 = nn.MaxPool3d(kernel_size=2)
        self.bn4 = nn.BatchNorm3d(64) if self.batch_norm else nn.
    Identity()

        self.conv5_1 = nn.Conv3d(64, 64, kernel_size=3, padding=1)
        self.conv5_2 = nn.Conv3d(64, 64, kernel_size=3, padding=1)
        self.pool5 = nn.MaxPool3d(kernel_size=2)
        self.bn5 = nn.BatchNorm3d(64) if self.batch_norm else nn.
    Identity()

        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(self._calculate_flatten_size(
    input_shape0), 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 1)

    def _calculate_flatten_size(self, input_shape0):
        with torch.no_grad():
            x = torch.zeros(1, *input_shape0)
            x = F.relu(self.conv1_1(x))
            x = F.relu(self.conv1_2(x))
            x = self.bn1(x) if self.batch_norm else x
            x = self.pool1(x)
            x = F.relu(self.conv2_1(x))
            x = F.relu(self.conv2_2(x))
            x = self.bn2(x) if self.batch_norm else x
            x = self.pool2(x)
            x = F.relu(self.conv3_1(x))
            x = F.relu(self.conv3_2(x))
            x = self.bn3(x) if self.batch_norm else x
            x = self.pool3(x)
            x = F.relu(self.conv4_1(x))
            x = F.relu(self.conv4_2(x))
            x = self.bn4(x) if self.batch_norm else x
            x = self.pool4(x)
            x = F.relu(self.conv5_1(x))
            x = F.relu(self.conv5_2(x))
            x = self.bn5(x) if self.batch_norm else x
            x = self.pool5(x)
            return self.flatten(x).shape[1]

    def forward(self, x):
        x = F.relu(self.conv1_1(x))
        x = F.relu(self.conv1_2(x))
        x = self.bn1(x) if self.batch_norm else x
        x = self.pool1(x)

        x = F.relu(self.conv2_1(x))
```

```
        x = F.relu(self.conv2_2(x))
        x = self.bn2(x) if self.batch_norm else x
        x = self.pool2(x)

        x = F.relu(self.conv3_1(x))
        x = F.relu(self.conv3_2(x))
        x = self.bn3(x) if self.batch_norm else x
        x = self.pool3(x)

        x = F.relu(self.conv4_1(x))
        x = F.relu(self.conv4_2(x))
        x = self.bn4(x) if self.batch_norm else x
        x = self.pool4(x)

        x = F.relu(self.conv5_1(x))
        x = F.relu(self.conv5_2(x))
        x = self.bn5(x) if self.batch_norm else x
        x = self.pool5(x)

        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

A.2 Filtration rate

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class build_net(nn.Module):
    def __init__(self, input_shape0, batch_norm):
        super(build_net, self).__init__()

        self.batch_norm = batch_norm

        self.conv1 = nn.Conv3d(input_shape0[-1], 2, kernel_size=3,
    padding=1)
        self.bn1 = nn.BatchNorm3d(2) if batch_norm else nn.
   Identity()
        self.pool1 = nn.MaxPool3d(2)

        self.conv2 = nn.Conv3d(2, 2, kernel_size=3, padding=1)
```

```
        self.bn2 = nn.BatchNorm3d(2) if batch_norm else nn.
    Identity()
        self.pool2 = nn.MaxPool3d(2)

        self.conv3 = nn.Conv3d(2, 2, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm3d(2) if batch_norm else nn.
    Identity()
        self.pool3 = nn.MaxPool3d(2)

        self.conv4 = nn.Conv3d(2, 2, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm3d(2) if batch_norm else nn.
    Identity()
        self.pool4 = nn.MaxPool3d(2)

        self.conv5 = nn.Conv3d(2, 2, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm3d(2) if batch_norm else nn.
    Identity()
        self.pool5 = nn.MaxPool3d(2)

        self.flatten = nn.Flatten()
        self.dropout = nn.Dropout(0.5)

        self.fc1 = nn.Linear(self._calculate_flatten_size(
    input_shape0), 10)
        self.fc2 = nn.Linear(10, 1)

        self.fc3 = nn.Linear(2, 32)
        self.fc4 = nn.Linear(32, 1)

        self.fc6 = nn.Linear(2, 32)
        self.fc7 = nn.Linear(32, 1)

    def _calculate_flatten_size(self, input_shape0):
        with torch.no_grad():
            x = torch.zeros(1, *input_shape0)

            x = x.permute(0, 4, 1, 2, 3)
            x = self.pool1(F.relu(self.bn1(self.conv1(x))))
            x = self.pool2(F.relu(self.bn2(self.conv2(x))))
            x = self.pool3(F.relu(self.bn3(self.conv3(x))))
            x = self.pool4(F.relu(self.bn4(self.conv4(x))))
            x = self.pool5(F.relu(self.bn5(self.conv5(x))))
            return self.flatten(x).shape[1]

    def forward(self, input_image, input_pressure, input_diff):
        input_image = input_image.permute(0, 4, 1, 2, 3)

        x1 = self.pool1(F.relu(self.bn1(self.conv1(input_image))))
        x1 = self.pool2(F.relu(self.bn2(self.conv2(x1))))
```

```
        x1 = self.pool3(F.relu(self.bn3(self.conv3(x1))))
        x1 = self.pool4(F.relu(self.bn4(self.conv4(x1))))
        x1 = self.pool5(F.relu(self.bn5(self.conv5(x1))))

        x1 = self.flatten(x1)
        x1 = self.dropout(x1)
        x1 = F.relu(self.fc1(x1))
        x1 = self.fc2(x1)

        x2 = torch.cat([input_pressure, input_diff], dim=1)
        x2 = F.relu(self.fc3(x2))
        x2 = self.fc4(x2)

        x_combined = torch.cat([x1, x2], dim=1)
        x_combined = F.relu(self.fc6(x_combined))
        x_out = self.fc7(x_combined)

        return x_out
```

Appendice B

Generazione degli impaccamenti e gestione delle simulazioni

B.1 Generatore delle geometrie

```
import yaml
import random
import subprocess
import os
import shutil

def modify_yaml(file_path, modifications):
    with open(file_path, 'r') as file:
        data = yaml.safe_load(file)
    for key, value in modifications.items():
        keys = key.split('.')
        temp = data
        for k in keys[:-1]:
            temp = temp.setdefault(k, {})
        temp[keys[-1]] = value

    with open(file_path, 'w') as file:
        yaml.dump(data, file, default_flow_style=False)

    diameter = data["grain_size_distribution"]["diameter"]
    mean = data["grain_size_distribution"]["mean"]
    std_dev = data["grain_size_distribution"]["std_dev"]
```

```
    return diameter, mean, std_dev

def extract_location_from_log(log_file):
    try:
        with open(log_file, 'r') as file:
            lines = file.readlines()
        for line in lines:
            if "Location in Mesh" in line:
                coordinates = line.split(":")[1].strip().strip("[]
").split()
                x, y, z = map(float, coordinates)
                return x, y, z
    except FileNotFoundError:
        print(f"Attenzione: il file {log_file} non è stato trovato
.")
        return None, None, None

def generate_samples(file_path, txt_path, n_samples):

    folder_path = "./"
    items_to_delete = ["b1-beforeSimulation.blend", "b2-
endSimulation.blend", "b3-cleanFinalPositions.blend", "b4-
swissGrid.blend", "b5-deletedGridBorder.blend", "stl_files", "
geometry_comsol.txt", "simulation.log", "dataset.txt"]

    for item in items_to_delete:
        item_path = os.path.join(folder_path, item)

        if os.path.isfile(item_path):
            os.remove(item_path)
            print(f"File eliminato: {item_path}")

        elif os.path.isdir(item_path):
            shutil.rmtree(item_path)
            print(f"Cartella eliminata: {item_path}")

    stl_folder = 'stl_files'
    if not os.path.exists(stl_folder):
        os.makedirs(stl_folder)

    with open(txt_path, 'a') as txtfile:

        if os.stat(txt_path).st_size == 0:
            txtfile.write("Sample\tDiameter\tMean\tStd_Dev\
tLocation_X\tLocation_Y\tLocation_Z\n")

        for sample_number in range(221, n_samples + 1):
            mean = random.uniform(1.0, 2.0)
            modifications = {
```

```
                "additional_options.scale_factor": 1.0,
                "additional_options.smoothness_level": 0,
                "container.shape": "box",
                "container.user_def_size": 14,
                "fragmentation.flag": False,
                "fragmentation.noise": 0.2,
                "fragmentation.number_of_fragments": 20,
                "grain_size_distribution.diameter": 1.0,
                "grain_size_distribution.distr": "gaussian",
                "grain_size_distribution.grains": 4000,
                "grain_size_distribution.mean": mean,
                "grain_size_distribution.pellet": "Cylinder",
                "grain_size_distribution.std_dev": random.uniform
    (0, 0.3*mean),
                "rigid_body.friction": 0.3,
                "rigid_body.refine_level": 3,
                "rigid_body.restitution": 0,
            }


            diameter, mean, std_dev = modify_yaml(file_path,
    modifications)

            subprocess.run(['blender', '-b', './empty.blend', '-P'
    , './bsand.py'])

            log_file = 'simulation.log'
            location_x, location_y, location_z =
    extract_location_from_log(log_file)

            txtfile.write(f"{sample_number}\t{diameter}\t{mean}\t{
    std_dev}\t{location_x}\t{location_y}\t{location_z}\n")

            stl_file_name = f'sample_edt_{sample_number:03d}.stl'
            source_file = 'merged.stl'
            destination_file = os.path.join(stl_folder,
    stl_file_name)

            if os.path.exists(source_file):
                shutil.move(source_file, destination_file)
            else:
                print(f"Attenzione: il file {source_file} non è
    stato trovato.")

     print(f"{n_samples} campioni generati e salvati in {txt_path}
    e nella cartella {stl_folder}")

if __name__ == "__main__":
     file_path = "stmCase.yaml"
```

```
    txt_path = "dataset.txt"
    n_samples = 280
    generate_samples(file_path, txt_path, n_samples)
```

B.2 Orchestratore

```
import os
import shutil
import random
import csv

base_dir = os.getcwd()
geometries_dir = os.path.join(base_dir, 'geometries')
microscale_dir = os.path.join(base_dir, 'microscale')
simulations_dir = os.path.join(base_dir, 'simulations')
txt_file = os.path.join(base_dir, 'Cylinders.txt')

def read_txt(file_path):
    samples = []
    try:
        with open(file_path, mode='r', newline='') as file:
            reader = csv.DictReader(file, delimiter='\t')
            print(f"Column names in TXT: {reader.fieldnames}")
            for row in reader:
                if not row['Sample'].strip():
                    continue
                try:
                    sample_id = int(row['Sample'])
                    dg_value = random.uniform(30e-09, 100e-09)
                    sample = {
                        'Sample': f"{sample_id:03d}",
                        'Diameter': row.get('Diameter'),
                        'Mean': row.get('Mean'),
                        'Std_Dev': row.get('Std_Dev'),
                        'Location_X': float(row['Location_X']),
                        'Location_Y': float(row['Location_Y']),
                        'Location_Z': float(row['Location_Z']),
                        'Pressure': random.uniform
    (6.54235433573448E-07, 7.771040393909E-06
),
                        'dg': dg_value,
                        'DT': float((1.380649e-23 * 298) / (3*
    3.14159265359 * 8.9e-4 * dg_value))
                    }
                    samples.append(sample)
```

```
                except ValueError:
                    print(f"Warning: Invalid data in row {row}")
        return samples
    except FileNotFoundError:
        print(f"Error: TXT file not found: {file_path}")
        return []


def update_case_setup(file_path, xloc, yloc, zloc, pressure):
    if not os.path.exists(file_path):
        print(f"Warning: caseSetup file not found: {file_path}")
        return
    with open(file_path, 'r') as file:
        lines = file.readlines()
    with open(file_path, 'w') as file:
        for line in lines:
            if line.startswith('xloc'):
                file.write(f'xloc    {xloc};\n')
            elif line.startswith('yloc'):
                file.write(f'yloc    {yloc};\n')
            elif line.startswith('zloc'):
                file.write(f'zloc    {zloc};\n')
            elif line.startswith('p0'):
                file.write(f'p0      {pressure};\n')
            else:
                file.write(line)

def update_transport_properties(file_path, DT):
    if not os.path.exists(file_path):
        print(f"Warning: transportProperties file not found: {
   file_path}")
        return
    with open(file_path, 'r') as file:
        lines = file.readlines()
    with open(file_path, 'w') as file:
        for line in lines:
            if line.lstrip().startswith('DT'):
                file.write(f'DT            DT [0 2 -1 0 0 0 0] {DT
   };\n')
            else:
                file.write(line)

def ignore_git(dir, files):
    if '.git' in files:
        return ['.git']
    return []

def orchestrate_simulations():
    if not os.path.exists(txt_file):
```

```
        print(f"Error: Cylinders.txt file not found: {txt_file}")
        return

    samples = read_txt(txt_file)
    if not samples:
        return

    try:
        with open(txt_file, mode='w', newline='') as outfile:
            fieldnames = ['Sample', 'Diameter', 'Mean', 'Std_Dev',
    'Location_X', 'Location_Y', 'Location_Z', 'Pressure', 'dg', '
DT']
            writer = csv.DictWriter(outfile, fieldnames=fieldnames
, delimiter='\t')
            writer.writeheader()
            for sample in samples:
                sample_id = sample['Sample']
                print(f"Processing sample {sample_id}")

                microscale_copy_dir = os.path.join(simulations_dir
, f'microscale_{sample_id}')
                microscale_rev_dir = os.path.join(
microscale_copy_dir, 'microscale-REV', 'Flow')

                os.makedirs(microscale_rev_dir, exist_ok=True)
                os.makedirs(os.path.join(microscale_rev_dir, '
constant', 'triSurface'), exist_ok=True)

                shutil.copytree(microscale_dir,
microscale_copy_dir, dirs_exist_ok=True, ignore=ignore_git)

                geometry_file = os.path.join(geometries_dir, f'
sample_edt_{sample_id}.stl')
                if os.path.exists(geometry_file):
                    shutil.copy(geometry_file, os.path.join(
microscale_copy_dir, 'microscale-REV', 'Flow', 'constant', '
triSurface'))
                else:
                    print(f"Warning: Geometry file not found for
sample {sample_id}: {geometry_file}")

                case_setup_file = os.path.join(microscale_rev_dir,
 'caseSetup')
                update_case_setup(case_setup_file, sample['
Location_X'], sample['Location_Y'], sample['Location_Z'],
sample['Pressure'])
```

```
                transport_properties_file = os.path.join(
    microscale_copy_dir, 'microscale-REV', 'Filtration', 'constant'
    , 'transportProperties')
                update_transport_properties(
    transport_properties_file, sample['DT'])

                snappy_hex_mesh_dict_path = os.path.join(
    microscale_rev_dir, 'system', 'snappyHexMeshDict')
                if os.path.exists(snappy_hex_mesh_dict_path):
                    with open(snappy_hex_mesh_dict_path, 'r') as
    file:
                        lines = file.readlines()
                    with open(snappy_hex_mesh_dict_path, 'w') as
    file:
                        for line in lines:
                            if 'Liquirizia.stl' in line:
                                file.write(line.replace('
    Liquirizia.stl', f'sample_edt_{sample_id}.stl'))
                            else:
                                file.write(line)
                else:
                    print(f"Warning: snappyHexMeshDict file not
    found: {snappy_hex_mesh_dict_path}")

                writer.writerow(sample)
                print(f"Processed sample {sample_id}")
            except Exception as e:
                print(f"Error: An unexpected error occurred: {e}")

if __name__ == "__main__":
    orchestrate_simulations()
```

B.3 Lanciatore

```
import os
import subprocess

def submit_jobs():
    current_dir = os.getcwd()
    simulations_dir = os.path.join(current_dir, 'simulations')
    if not os.path.exists(simulations_dir):
        print(f"Error: 'simulations' directory not found in {
    current_dir}.")
        return
```

```
    microscale_dirs = [d for d in os.listdir(simulations_dir) if d
    .startswith("microscale_") and os.path.isdir(os.path.join(
    simulations_dir, d))]
    for microscale_dir in microscale_dirs:
        microscale_rev_dir = os.path.join(simulations_dir,
    microscale_dir, "microscale-REV")
        if os.path.exists(microscale_rev_dir):
            print(f"Found {microscale_rev_dir}, submitting sbatch
    job...")
            sbatch_file = os.path.join(microscale_rev_dir, "micro.
    sbatch")  # Usa il nome corretto per il file sbatch
            try:
                os.chdir(microscale_rev_dir)
                result = subprocess.run(['sbatch', sbatch_file],
    stdout=subprocess.PIPE, stderr=subprocess.PIPE)
                os.chdir(current_dir)
                if result.returncode == 0:
                    print(f"Job for {microscale_rev_dir} submitted
     successfully!")
                    print(f"sbatch output: {result.stdout.decode()
    }")
                else:
                    print(f"Error submitting job for {
    microscale_rev_dir}: {result.stderr.decode()}")
                    print(f"sbatch output: {result.stdout.decode()
    }")
            except Exception as e:
                print(f"Failed to submit job for {
    microscale_rev_dir}: {e}")
        else:
            print(f"Directory {microscale_rev_dir} not found,
    skipping...")

if __name__ == "__main__":
    submit_jobs()
```

B.4 *Post-process*

```
import os
import re
import csv
import shutil

def get_latest_time(folder_name, subfolder):
```

```
    path = os.path.join(folder_name, 'microscale-REV', 'Flow', '
    postProcessing', subfolder)
    if os.path.exists(path):
        times = [d for d in os.listdir(path) if os.path.isdir(os.
    path.join(path, d)) and d.replace('.', '', 1).isdigit()]
        if times:
            return max(times, key=float)
    return None


def extract_inlet_area(folder_name):
    latest_time = get_latest_time(folder_name, 'inlet_average')
    if not latest_time:
        print(f"Warning: Inlet area data not found for {
    folder_name}")
        return None
    inlet_file = os.path.join(folder_name, 'microscale-REV', 'Flow
    ', 'postProcessing', 'inlet_average', latest_time, '
    surfaceFieldValue.dat')
    if not os.path.exists(inlet_file):
        print(f"Warning: File not found for inlet area: {
    inlet_file}")
        return None
    with open(inlet_file, 'r') as f:
        lines = f.readlines()
    for line in lines:
        if line.strip().startswith("# Area"):
            parts = line.split(':')
            if len(parts) >= 2:
                try:
                    return float(parts[1].strip())
                except ValueError:
                    print(f"Warning: Unable to convert inlet area
    value: {line}")
                    return None
    print(f"Warning: Inlet Area not found in {inlet_file}")
    return None

def extract_inlet_velocity_z(folder_name):
    latest_time = get_latest_time(folder_name, 'inlet_average')
    if not latest_time:
        print(f"Warning: Inlet velocity data not found for {
    folder_name}")
        return None
    inlet_file = os.path.join(folder_name, 'microscale-REV', 'Flow
    ', 'postProcessing', 'inlet_average', latest_time, '
    surfaceFieldValue.dat')
    if not os.path.exists(inlet_file):
```

```
        print(f"Warning: File not found for inlet velocity: {
    inlet_file}")
        return None
    with open(inlet_file, 'r') as f:
        lines = f.readlines()
    for line in reversed(lines):
        if line.strip() and line[0] != '#':
            try:
                flow_z_str = line.split('\t')[1].strip('()')
                flow_z_values = re.findall(r"[-+]?\d*\.?\d+(?:[eE
    ][-+]?\d+)?", flow_z_str)
                if len(flow_z_values) == 3:
                    return float(flow_z_values[2])
                else:
                    print(f"Warning: Incorrect number of values in
     inlet velocity data: {line}")
                    return None
            except (ValueError, IndexError):
                print(f"Warning: Unable to convert inlet velocity
    value or invalid line format: {line}")
                return None
    print(f"Warning: Inlet velocity data not found in {inlet_file}
    ")
    return None

def extract_outlet_area(folder_name):
    latest_time = get_latest_time(folder_name, 'outlet_average')
    if not latest_time:
        print(f"Warning: Outlet area data not found for {
    folder_name}")
        return None
    outlet_file = os.path.join(folder_name, 'microscale-REV', '
    Flow', 'postProcessing', 'outlet_average', latest_time, '
    surfaceFieldValue.dat')
    if not os.path.exists(outlet_file):
        print(f"Warning: File not found for outlet area: {
    outlet_file}")
        return None
    with open(outlet_file, 'r') as f:
        lines = f.readlines()
    for line in lines:
        if line.strip().startswith("# Area"):
            parts = line.split(':')
            if len(parts) >= 2:
                try:
                    return float(parts[1].strip())
                except ValueError:
                    print(f"Warning: Unable to convert outlet area
     value: {line}")
```

```
                    return None
    print(f"Warning: Outlet Area not found in {outlet_file}")
    return None

def extract_outlet_velocity_z(folder_name):
    latest_time = get_latest_time(folder_name, 'outlet_average')
    if not latest_time:
        print(f"Warning: Outlet velocity data not found for {
    folder_name}")
        return None
    outlet_file = os.path.join(folder_name, 'microscale-REV', '
    Flow', 'postProcessing', 'outlet_average', latest_time, '
    surfaceFieldValue.dat')
    if not os.path.exists(outlet_file):
        print(f"Warning: File not found for outlet velocity: {
    outlet_file}")
        return None
    with open(outlet_file, 'r') as f:
        lines = f.readlines()
    for line in reversed(lines):
        if line.strip() and line[0] != '#':
            try:
                flow_z_str = line.split('\t')[1].strip('()')
                flow_z_values = re.findall(r"[-+]?\d*\.?\d+(?:[eE
    ][-+]?\d+)?", flow_z_str)
                if len(flow_z_values) == 3:
                    return float(flow_z_values[2])
                else:
                    print(f"Warning: Incorrect number of values in
    outlet velocity data: {line}")
                    return None
            except (ValueError, IndexError):
                print(f"Warning: Unable to convert outlet velocity
    value or invalid line format: {line}")
                return None
    print(f"Warning: Outlet velocity data not found in {
    outlet_file}")
    return None

def extract_medium_volume(folder_name):
    latest_time = get_latest_time(folder_name, 'volume_average')
    if not latest_time:
        print(f"Warning: Medium volume data not found for {
    folder_name}")
        return None
    volume_file = os.path.join(folder_name, 'microscale-REV', '
    Flow', 'postProcessing', 'volume_average', latest_time, '
    volFieldValue.dat')
    if os.path.exists(volume_file):
```

```
        with open(volume_file, 'r') as f:
            lines = f.readlines()
        for line in lines:
            if line.startswith("# Volume"):
                parts = line.split(':')
                if len(parts) >= 2:
                    try:
                        volume = float(parts[1].strip())
                        return volume
                    except ValueError:
                        print(f"Warning: Line conversion error for
    medium volume: {line}")
                        return None
    print(f"Warning: Medium volume not found in {volume_file}")
    return None

def extract_average_concentration(folder_name):
    conc_file = os.path.join(folder_name, 'microscale-REV', '
   Filtration', 'postProcessing', 'volume_average', '0', '
   volFieldValue.dat')
    if os.path.exists(conc_file):
        try:
            with open(conc_file, 'r') as f:
                lines = f.readlines()
            last_value = None
            for line in lines:
                if line.strip() and line[0] != '#':
                    parts = line.split()
                    if len(parts) >= 2:
                        try:
                            last_value = float(parts[1].strip())
                        except ValueError:
                            print(f"Warning: Line conversion error
    for average concentration: {line}")
                            return None
            if last_value is not None:
                return last_value
            else:
                print(f"Warning: No numeric value found for
   average concentration in {conc_file}")
                return None
        except FileNotFoundError:
            print(f"Warning: File not found for average
   concentration: {conc_file}")
            return None
        except Exception as e:
            print(f"Warning: Error while reading average
   concentration file: {e}")
            return None
```

```
    else:
        print(f"Warning: File not found for average concentration:
    {conc_file}")
        return None

def extract_results_data(folder_name):
    results_file = os.path.join(folder_name, 'microscale-REV', '
   Flow', 'Results', 'Results.dat')
    porosity = permeability = tortuosity = reynolds =
   darcyan_velocity = mesh_volume = rev_volume = None

    if os.path.exists(results_file):
        try:
            with open(results_file, 'r') as f:
                for line in f:
                    line = line.strip()
                    if line.startswith("Porosity:"):
                        try:
                            porosity = float(line.split(":")[1].
   strip())
                        except ValueError:
                            print(f"Warning: Error reading
   Porosity: {line}")
                    elif line.startswith("Permeability"):
                        try:
                            parts = line.split(":")
                            if len(parts) > 1:
                                permeability = float(parts[1].
   strip())
                        except ValueError:
                            print(f"Warning: Error reading
   Permeability: {line}")
                    elif line.startswith("Tortuosity:"):
                        try:
                            tortuosity = float(line.split(":")[1].
   strip())
                        except ValueError:
                            print(f"Warning: Error reading
   Tortuosity: {line}")
                    elif line.startswith("Reynolds:"):
                        try:
                            reynolds = float(line.split(":")[1].
   strip())
                        except ValueError:
                            print(f"Warning: Error reading
   Reynolds: {line}")
                    elif line.startswith("Darcyan velocity (m/s):"
   ):
                        try:
```

```
                            darcyan_velocity = float(line.split(":
    ")[1].strip())
                        except ValueError:
                            print(f"Warning: Error reading Darcyan
     velocity: {line}")
                    elif line.startswith("Mesh volume (m^3):"):
                        try:
                            mesh_volume = float(line.split(":")
    [1].strip())
                        except ValueError:
                            print(f"Warning: Error reading Mesh
    volume: {line}")
                    elif line.startswith("REV volume (m^3):"):
                        try:
                            rev_volume = float(line.split(":")[1].
    strip())
                        except ValueError:
                            print(f"Warning: Error reading REV
    volume: {line}")
            except Exception as e:
                print(f"Warning: Error during reading of {results_file
    }: {e}")
        else:
            print(f"Warning: File not found: {results_file}")

        return porosity, permeability, tortuosity, reynolds,
    darcyan_velocity, mesh_volume, rev_volume

def extract_breakthrough_data(folder_name):
    breakthrough_file = os.path.join(folder_name, 'microscale-REV'
    , 'Filtration', 'breakthrough_results.dat')
    if os.path.exists(breakthrough_file):
        try:
            with open(breakthrough_file, 'r') as f:
                lines = f.readlines()
            if lines:
                last_line = lines[-1].split()
                if len(last_line) >= 2:
                    try:
                        ftotout = float(last_line[-1])
                        ftotin = float(last_line[-2])
                        tmeanout = float(last_line[1])
                        return ftotin, ftotout, tmeanout
                    except ValueError:
                        print(f"Warning: Could not convert
    breakthrough data to float in last line: {last_line}")
                        return None, None, None
                else:
```

```
                    print(f"Warning: Not enough columns in the
    last line of breakthrough data: {last_line}")
                    return None, None, None
            else:
                print(f"Warning: breakthrough_results.dat is empty
     in {folder_name}")
                return None, None, None
        except FileNotFoundError:
            print(f"Warning: breakthrough_results.dat not found in
     {folder_name}")
            return None, None, None
        except Exception as e:
            print(f"Warning: Error reading breakthrough_results.
    dat in {folder_name}: {e}")
            return None, None, None
     else:
        print(f"Warning: breakthrough_results.dat not found in {
    folder_name}")
        return None, None, None

def process_simulations(base_dir, txt_file):
    samples = []
    with open(txt_file, 'r', newline='') as infile:
        reader = csv.DictReader(infile, delimiter='\t')
        samples = list(reader)
    updated_samples = []
    for sample in samples:
        sample_id = sample['Sample']
        microscale_folder = f"microscale_{sample_id}"
        microscale_path = os.path.join(base_dir, microscale_folder
    )
        if os.path.isdir(microscale_path):
            inlet_area = extract_inlet_area(microscale_path)
            inlet_velocity_z = extract_inlet_velocity_z(
    microscale_path)
            outlet_area = extract_outlet_area(microscale_path)
            outlet_velocity_z = extract_outlet_velocity_z(
    microscale_path)
            volume = extract_medium_volume(microscale_path)
            average_concentration = extract_average_concentration(
    microscale_path)
            porosity, permeability, tortuosity, reynolds,
    darcyan_velocity, mesh_volume, rev_volume =
    extract_results_data(microscale_path)
            ftotin, ftotout, tmeanout = extract_breakthrough_data(
    microscale_path)

            sample['Inlet Area'] = inlet_area
            sample['Inlet velocity'] = inlet_velocity_z
```

```
            sample['Outlet Area'] = outlet_area
            sample['Outlet velocity'] = outlet_velocity_z
            sample['Volume'] = volume
            sample['Average Concentration'] =
    average_concentration
            sample['Porosity'] = porosity
            sample['Permeability'] = permeability
            sample['Tortuosity'] = tortuosity
            sample['Reynolds'] = reynolds
            sample['Ftotin'] = ftotin
            sample['Ftotout'] = ftotout
            sample['Average area concentration out'] = tmeanout
            sample['Darcyan velocity'] = darcyan_velocity
            if rev_volume is not None and mesh_volume is not None:
                sample['Porous medium volume'] = rev_volume -
    mesh_volume
            else:
                print(f"Warning: Volume not calculable for {
    microscale_path} (rev or mesh volume missing)")
                sample['Porous medium volume'] = None

            updated_samples.append(sample)
        else:
            print(f"Warning: Folder not found: {microscale_path}")

    output_file = os.path.join(base_dir, 'Cylinders_output.txt')
    with open(output_file, 'w', newline='') as outfile:
        fieldnames = updated_samples[0].keys() if updated_samples
    else []
        writer = csv.DictWriter(outfile, fieldnames=fieldnames,
    delimiter='\t')
        writer.writeheader()
        writer.writerows(updated_samples)
    shutil.move(output_file, txt_file)

    print(f"Data processing completed. Warnings may have been
    issued.")

if __name__ == "__main__":
    base_dir = os.path.join(os.getcwd(), 'simulations')
    txt_file = 'Cylinders.txt'

    process_simulations(base_dir, txt_file)
```

Acronimi

AI

Artificial Intelligence

ANN

Artificial Neural Network

CFD

Computational Fluid Dynamics

CNN

Convolutional Neural Network

ECDF

Empirical Cumulative Distribution Function

FC

Fully Connected

FCNN

Fully Connected Neural Network

FFNN

Feedforward Neural Network

FDM

Finite Difference Method

FEM

Finite Element Method

FVM
Finite Volume Method

HPC
High Performance Computing

KDE
Kernel Density Estimation

LDS
Linear Differencing Scheme

LUDS
Linear Upwind Differencing Scheme

MAE
Mean Absolute Error

MSE
Mean Squared Error

PCM
Phase Change Material

ReLU
Rectified Linear Unit

REV
Representative Elementary Volume

RNN
Recurrent Neural Network

SIMPLE
Semi-Implicit Method for Pressure Linked Equations

SLURM
Simple Linux Utility for Resource Management

UDS
Upwind Differencing Scheme

Lista dei simboli

Simbolo	Significato e Unità di misura
α	learning rate, [-]
β	coefficiente di inerzia (o di Forchheimer), $[m^{-1}]$
b	bias della rete neurale, [-]
Γ	coefficiente di diffusività generico, $[m^2\ s^{-1}]$
C	concentrazione, $[mol\ m^{-3}]$
$\langle C \rangle$	concentrazione media nel volume, $[mol\ m^{-3}]$
$\mathcal{D}$	coefficiente di diffusività di materia, $[m^2\ s^{-1}]$
d_c	diametro del colloide, $[m]$
d_g	diametro dei grani, $[m]$
ϵ	porosità, [-]
F_{tot}	flusso molare totale (advettivo e diffusivo), $[mol\ s^{-1}]$
g	accelerazione di gravità, $[m\ s^{-2}]$
J_i	flusso molare in direzione i, $[mol\ m^{-2}\ s^{-1}]$
k	permeabilità, $[m^2]$
k_b	costante di Boltzmann, $1.380649 \times 10^{-23}\,[J\ K^{-1}]$
K_f	velocità di filtrazione, $[s^{-1}]$
λ	learning rate della rete neurale, [-]
L	lunghezza del mezzo poroso, $[m]$
μ	viscosità dinamica, $[Pa\ s]$
ν	viscosità cinematica, $[m^2\ s^{-1}]$
π	pi greco, 3.14159 [-]
p	pressione, $[Pa]$

Simbolo	**Significato e Unità di misura**
q	velocità superficiale (o di Darcy), $[m\ s^{-1}]$
ρ	densità, $[kg\ m^{-3}]$
σ	deviazione standard, [var]
σ_{norm}	coefficiente di variazione, [-]
T	temperatura, $[K]$
v_i	velocità lungo la componente i, $[m\ s^{-1}]$
V	volume, $[m^3]$
w_i	peso sinaptico di un neurone, [-]
x_i	input feature della rete neurale, [var]
$\hat{y}_i$	output predetto dalla rete neurale, [var]
y_i	target della rete, [var]
ϕ	generica grandezza scalare, [var]
z_i	valore di combinazione lineare in una rete neurale, [var]

Bibliografia

[1] Davood Domairry Ganji e Sayyid Habibollah Hashemi Kachapi. «Chapter 7 - Nanofluid Flow in Porous Medium». In: *Application of Nonlinear Systems in Nanomechanics and Nanofluids*. A cura di Davood Domairry Ganji e Sayyid Habibollah Hashemi Kachapi. Micro and Nano Technologies. Oxford: William Andrew Publishing, 2015, pp. 271–316. ISBN: 978-0-323-35237-6. DOI: https://doi.org/10.1016/B978-0-323-35237-6.00007-8. URL: https://www.sciencedirect.com/science/article/pii/B9780323352376000078 (cit. a p. 1).

[2] Saideep Pavuluri, Ran Holtzman, Luqman Kazeem, Malyah Mohammed, Thomas Daniel Seers e Harris Sajjad Rabbani. «Interplay of viscosity and wettability controls fluid displacement in porous media». In: *Phys. Rev. Fluids* 8 (9 set. 2023), p. 094002. DOI: 10.1103/PhysRevFluids.8.094002. URL: https://link.aps.org/doi/10.1103/PhysRevFluids.8.094002 (cit. a p. 2).

[3] Friedrich-Alexander-Universität Erlangen-Nürnberg. *Deep Conversion in Packed Bed Reactors*. https://www.dip.fau.de/home/the-project/deep-conversion-in-packed-bed-reactors/. Accessed: 2025-05-23 (cit. a p. 3).

[4] Gang Liu, Tian Xiao, Junfei Guo, Pan Wei, Xiaohu Yang e Kamel Hooman. «Melting and solidification of phase change materials in metal foam filled thermal energy storage tank: Evaluation on gradient in pore structure». In: *Applied Thermal Engineering* 212 (2022), p. 118564. ISSN: 1359-4311. DOI: https://doi.org/10.1016/j.applthermaleng.2022.118564. URL: https://www.sciencedirect.com/science/article/pii/S1359431122005142 (cit. a p. 4).

[5] G. Boccardo, D. L. Marchisio e R. Sethi. «Microscale simulation of particle deposition in porous media». In: *Journal of Colloid and Interface Science* 417 (2014), pp. 227–237. DOI: 10.1016/j.jcis.2013.11.007 (cit. alle pp. 7, 70).

[6] Shunhua Zhou, X. Zhang, D. Wu e H. Di. «Mathematical modeling of slurry infiltration and particle dispersion in saturated sand». In: *Transport in Porous Media* 124 (2018), pp. 91–116. DOI: 10.1007/s11242-018-1054-x. URL:

https://www.researchgate.net/publication/324722338_Mathematical_Modeling_of_Slurry_Infiltration_and_Particle_Dispersion_in_Saturated_Sand (cit. a p. 9).

[7] R.B. Bird, W.E. Stewart e E.N. Lightfoot. *Transport Phenomena*. J. Wiley, 2002. ISBN: 9780471364740. URL: https://books.google.it/books?id=wYnRQwAACAAJ (cit. a p. 8).

[8] Daniele Marchisio. *Fenomeni di trasporto e Fluidodinamica computazionale*. Lezioni. 2023 (cit. a p. 8).

[9] Gianluca Boccardo, Eleonora Crevacore, Rajandrea Sethi e Matteo Icardi. «A robust upscaling of the effective particle deposition rate in porous media». In: *Journal of Contaminant Hydrology* 212 (2018), pp. 3–13. DOI: 10.1016/j.jconhyd.2017.09.002. URL: https://doi.org/10.1016/j.jconhyd.2017.09.002 (cit. a p. 12).

[10] Agnese Marcato, Gianluca Boccardo e Daniele Marchisio. «From Computational Fluid Dynamics to Structure Interpretation via Neural Networks: An Application to Flow and Transport in Porous Media». In: *Industrial & Engineering Chemistry Research* 61.24 (2022), pp. 8530–8541. DOI: 10.1021/acs.iecr.1c04760. URL: https://doi.org/10.1021/acs.iecr.1c04760 (cit. alle pp. 12, 39, 59, 62, 63, 86).

[11] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0 (cit. alle pp. 15, 17, 23, 24).

[12] Luca Magri. *Introduction to Scientific Machine Learning*. Course Notes. Copyright © 2024 by Luca Magri. All rights reserved. 2024 (cit. a p. 15).

[13] Prajit Datta. *The Concept of Artificial Neurons (Perceptrons) in Neural Networks*. Accessed: April 3, 2025. Feb. 2020. URL: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc/ (cit. a p. 15).

[14] Arnaud Nguembang Fadja, Evelina Lamma e Fabrizio Riguzzi. «Vision inspection with neural networks». In: *R.i.C.e.R.c.A: RCRA Incontri E Confronti*. A cura di Marco Maratea e Mauro Vallati. Vol. 2272. CEUR WORKSHOP PROCEEDINGS. Aachen, Germany: CEUR-WS, 2018, pp. 1–10. DOI: n/a. URL: http://ceur-ws.org/Vol-2272/paper1.pdf (cit. a p. 20).

[15] OpenDataScience. *Recurrent Neural Networks in the Cloud and Edge*. https://opendatascience.com/recurrent-neural-networks-in-the-cloud-and-edge/. Accessed: 2025-05-03 (cit. a p. 21).

[16] Lei Lei, Yaxiong Yuan, Thang X. Vu, Symeon Chatzinotas e Björn Ottersten. «Learning-Based Resource Allocation: Efficient Content Delivery Enabled by Convolutional Neural Network». In: *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. 2019, pp. 1–5. DOI: 10.1109/SPAWC.2019.8815447 (cit. a p. 22).

[17] Fadil Moukalled, Lilianne Mangani e Mazen Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM® and Matlab®*. Springer International Publishing, 2016. ISBN: 978-3-319-16858-6. DOI: 10.1007/978-3-319-16874-6. URL: https://doi.org/10.1007/978-3-319-16874-6 (cit. alle pp. 28, 29, 33).

[18] Joel H. Ferziger, Milovan Perić e Robert L. Street. *Computational Methods for Fluid Dynamics*. Fourth Edition. Springer International Publishing, 2019. ISBN: 978-3-662-55030-9. DOI: 10.1007/978-3-642-56026-2. URL: https://doi.org/10.1007/978-3-642-56026-2 (cit. alle pp. 28, 30).

[19] N. P. Waterson e H. Deconinck. «Design principles for bounded higher-order convection schemes - a unified approach». In: *Journal of Computational Physics* 224 (2007), pp. 182–207. DOI: 10.1016/j.jcp.2007.01.021. URL: https://doi.org/10.1016/j.jcp.2007.01.021 (cit. a p. 32).

[20] Imane Khalil e Issam Lakkis. *Computational Fluid Dynamics: An Introduction to Modeling and Applications*. 1st. McGraw Hill, 2023. ISBN: 9781264274949 (cit. a p. 34).

[21] CFD Direct. *OpenFOAM User Guide v9: 4. Case Structure*. Accessed: 2025-04-24. Lug. 2021. URL: https://doc.cfd.direct/openfoam/user-guide-v9/cases#x15-1210004 (cit. a p. 36).

[22] Xianhui Zhao, Devin M. Walker, Debtanu Maiti, Amanda D. Petrov, Matthew Kastelic, Babu Joseph e John N. Kuhn. «NiMg/Ceria-Zirconia Cylindrical Pellet Catalysts for Tri-reforming of Surrogate Biogas». In: *Industrial & Engineering Chemistry Research* 57.3 (2018), pp. 862–871. DOI: 10.1021/acs.iecr.7b03669 (cit. a p. 52).

[23] Philipp Kaiser, Ferdinand Pöhlmann e Andreas Jess. «Intrinsic and Effective Kinetics of Cobalt-Catalyzed Fischer-Tropsch Synthesis in View of a Power-to-Liquid Process Based on Renewable Energy». In: *Chemical Engineering & Technology* 37.11 (2014), pp. 1891–1900. DOI: 10.1002/ceat.201300815 (cit. a p. 52).

[24] Kyle M. Brunner, Hector D. Perez, Robson P. S. Peguin, Joshua C. Duncan, Luke D. Harrison, Calvin H. Bartholomew e William C. Hecker. «Effects of Particle Size and Shape on the Performance of a Trickle Fixed-Bed Recycle

Reactor for Fischer–Tropsch Synthesis». In: *Industrial & Engineering Chemistry Research* 54.8 (2015), pp. 2358–2367. DOI: 10.1021/ie504338w (cit. a p. 52).

[25] Nathaniel E. Helwig. *Density and Distribution Estimation*. Accessed: 2025-05-04. 2017. URL: http://users.stat.umn.edu/~helwig/notes/den-Notes.pdf (cit. a p. 55).

[26] Jacob Bear. *Dynamics of Fluids in Porous Media*. New York: Dover Publications, 1988. ISBN: 978-0486656755 (cit. a p. 64).

[27] Gianluca Boccardo, Frédéric Augier, Yacine Haroun, Daniel Ferré e Daniele L. Marchisio. «Validation of a novel open-source work-flow for the simulation of packed-bed reactors». In: *Chemical Engineering Journal* 279 (2015), pp. 809–820. ISSN: 1385-8947. DOI: https://doi.org/10.1016/j.cej.2015.05.032. URL: https://www.sciencedirect.com/science/article/pii/S138589471500683X (cit. a p. 66).

[28] Om Pramod. *Cross Validation*. URL: https://medium.com/@ompramod9921/cross-validation-623620ff84c2 (visitato il giorno 11/05/2025) (cit. a p. 78).

[29] Bala Priya C. *Neural Network Weight Initialization*. Accessed: 2025-05-25. 2023. URL: https://www.pinecone.io/learn/weight-initialization/ (cit. a p. 91).

[30] TensorFlow. *tf.keras.layers.Conv3D | TensorFlow Core v2.16.1*. Accessed: 2025-05-25. 2024. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv3D (cit. a p. 91).

[31] TensorFlow. *tf.keras.layers.Dense | TensorFlow Core v2.16.1*. Accessed: 2025-05-25. 2024. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense (cit. a p. 91).

[32] PyTorch. *torch.nn.Conv3d — PyTorch 2.7 documentation*. Accessed: 2025-05-25. 2024. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv3d.html (cit. a p. 91).

[33] PyTorch. *torch.nn.Linear — PyTorch 2.7 documentation*. Accessed: 2025-05-25. 2024. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear (cit. a p. 91).

[34] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. a p. 98).

[35] Chandrahas Mishra e D. L. Gupta. «Deep Machine Learning and Neural Networks: An Overview». In: *IAES International Journal of Artificial Intelligence (IJ-AI)* 6.2 (2017), pp. 66–73. DOI: 10.11591/ijai.v6.i2.pp66-73. URL: https://ijai.iaescore.com/index.php/IJAI/article/view/7539.

[36] Dimple Patil, Nitin Liladhar Rane, Pravin Desai e Jayesh Rane. «Machine learning and deep learning: Methods, techniques, applications, challenges, and future research opportunities». In: *International Journal of System Assurance Engineering and Management* 15 (2024), pp. 1–28. DOI: 10.70593/978-81-981367-4-9_2. URL: https://doi.org/10.70593/978-81-981367-4-9_2.

[37] Mohammad Mustafa Taye. «Understanding of Machine Learning with Deep Learning: Architectures, Workflow, Applications and Future Directions». In: *Preprints* (2021). DOI: 10.3390/computers12050091. URL: https://doi.org/10.3390/computers12050091.