



ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Transformers to Time Series Models

Relatore:
Vittorio Maniezzo

Presentata da:
Ramzi Gallala

Sessione II
Anno Accademico 2025/2026

dedica

Introduzione

Il continuo aumento della quantità di dati disponibili nel mondo moderno ha reso possibile analizzare fenomeni complessi e realizzare previsioni sempre più precise riguardo alla loro evoluzione futura. Questa capacità predittiva sta assumendo un ruolo centrale in molteplici settori, contribuendo a guidare scelte strategiche e operative in modo più consapevole ed efficiente. In un contesto in cui le decisioni devono essere rapide, fondate e spesso automatizzate, la previsione basata su dati rappresenta un **vantaggio competitivo fondamentale**.

Pertanto, l'era digitale, spesso definita come quella dei *Big Data*, ha innescato una trasformazione radicale nei processi decisionali, spostando l'equilibrio da un approccio basato sull'esperienza e sull'intuizione verso uno fondato sull'evidenza empirica. Al centro di questa rivoluzione si colloca la **previsione del futuro**, non più come semplice esercizio statistico, ma come **leva strategica** per anticipare cambiamenti, ottimizzare risorse e generare valore. Tra i molteplici ambiti applicativi, il settore **automobilistico** si distingue per la sua forte interconnessione con l'economia globale, le dinamiche del mercato, le politiche ambientali e le innovazioni tecnologiche. Fin dagli albori della produzione industriale, con l'introduzione della catena di montaggio da parte di Ford nei primi del Novecento, l'automotive è stato un simbolo dell'evoluzione tecnologica. Tuttavia, se inizialmente la previsione della domanda si basava su dati aggregati e tendenze macroeconomiche, oggi la situazione è profondamente cambiata.

Un veicolo moderno è, di fatto, un *centro dati su ruote*, mentre le fabbriche sono diventate nodi intelligenti di una rete globale interconnessa di fornitori, produttori e distributori. In questo ecosistema complesso, la capacità di generare previsioni affidabili non è più un'opzione, ma una **necessità strategica**. Alcuni esempi emblematici includono:

- **Ottimizzazione della supply chain:** previsioni accurate sulla domanda di specifici modelli, allestimenti o persino colori permettono di calibrare la produzione, ridurre gli stock inutilizzati e migliorare la reattività dell'intera catena logistica.
- **Strategie di marketing e pricing dinamico:** comprendere come varia la domanda in funzione di incentivi governativi, prezzi del carburante o campagne promozionali permette di affinare le strategie commerciali, massimizzando l'impatto degli investimenti.

L'industria automobilistica, dunque, si presta particolarmente bene a essere studiata come **caso d'uso concreto** in cui la previsione può offrire vantaggi misurabili e tangibili. Nel corso degli anni, questa esigenza ha portato allo sviluppo di diversi strumenti previsionali: dai *modelli statistici tradizionali* (come *ARIMA* [1]) fino a metodi più recenti basati su *machine learning*. Tuttavia, l'aumento esponenziale della complessità dei dati e delle dinamiche di mercato ha messo in luce i limiti dei metodi classici, spesso inadatti a gestire **relazioni molto complesse, pattern temporali discontinui o shock improvvisi**, come una pandemia o una crisi finanziaria.

È proprio l'interesse verso queste sfide che ha stimolato la mia curiosità personale e accademica. Questo percorso mi ha portato a chiedermi: *Quali architetture computazionali permettono di catturare dinamiche complesse e variabili? È possibile migliorare ulteriormente l'accuratezza di tali previsioni utilizzando tecniche all'avanguardia?*

In questo contesto, il rapido progresso dell'intelligenza artificiale – e in particolare del **deep learning** – ha aperto nuove strade promettenti. Negli ultimi anni, i modelli **Transformer**, inizialmente sviluppati per il trattamento del linguaggio naturale, hanno mostrato performance eccellenti anche in ambito *time series forecasting*, grazie alla loro capacità di modellare relazioni a lungo termine, gestire serie eterogenee e adattarsi a input non sequenziali [2].

Da queste premesse nasce questa tesi, con l'obiettivo di esplorare una nuova metodologia per affrontare il problema della previsione delle vendite di auto in Europa, sfruttando modelli avanzati di deep learning.

Obiettivi del lavoro

L'obiettivo principale di questa tesi è quello di investigare l'efficacia dei modelli *Transformer* nell'ambito delle previsioni di serie temporali, cercando di rispondere a una domanda centrale: **i Transformer possono rappresentare una metodologia più accurata rispetto alle tecniche tradizionalmente utilizzate, come ad esempio le LSTM (Long Short-Term Memory)?**

Negli ultimi anni, l'elaborazione di serie temporali ha visto un'evoluzione significativa grazie all'introduzione di tecniche di *deep learning*, in particolare delle *reti neurali ricorrenti LSTM*, che ha dimostrato una buona capacità di catturare dipendenze a lungo termine nei dati sequenziali [3]. Tuttavia, questi modelli presentano alcune **limitazioni**, come la difficoltà di *parallelizzazione* durante l'addestramento, la perdita di informazioni a lungo termine nei casi più complessi e tempi di addestramento relativamente lunghi.

Al contrario, i modelli *Transformer*, introdotti inizialmente nel contesto del *Natural Language Processing (NLP)*, si sono dimostrati estremamente efficaci nella modellazione di sequenze complesse grazie all'utilizzo del meccanismo di *self-attention* [2]. Questa architettura permette al modello di apprendere le relazioni tra tutti gli elementi di una sequenza **in parallelo**, superando molti dei limiti strutturali delle *RNN*. Di conseguenza, negli ultimi anni sono nate numerose varianti e adattamenti dei *Transformer* per l'ambito delle serie temporali, come *Time Series Transformer*, *Informer* [4], *Autoformer* [5] e *Temporal Fusion Transformer*.

Questa tesi si propone quindi di:

- **Analizzare criticamente** le principali tecniche attualmente impiegate nella previsione di serie temporali, con particolare attenzione ai modelli basati su **LSTM**;
- **Studiare l'architettura dei Transformer** e i motivi del loro recente successo in ambiti anche diversi da quello linguistico;
- **Implementare** uno o più modelli basati su *Transformer* per la previsione di serie temporali, confrontandone le performance con quelle ottenute da modelli *LSTM* su uno *dataset* di riferimento, utilizzando codice scritto in linguaggio *Python* e librerie come *PyTorch* o *TensorFlow*;
- **Valutare in modo sistematico** le prestazioni dei diversi modelli in termini di *accuracy predittiva*, *efficienza computazionale* e *capacità di generalizzazione*, cercando di comprendere se effettivamente possano superare le tecniche tradizionali in termini di precisione e affidabilità.

Metodologia adottata

Lo sviluppo di questo lavoro di tesi è stato guidato da un approccio *sperimentale* e *orientato al problema*, adattandosi progressivamente alle complessità emerse durante l'analisi dei dati e il confronto tra i diversi modelli di previsione.

Il lavoro è partito da un'approfondita analisi della **letteratura scientifica** e dello **stato dell'arte**. Questa attività ha permesso di comprendere le fondamenta della previsione di serie storiche, dai metodi classici fino alle architetture di *deep learning* più consolidate, come le reti **LSTM**. L'analisi ha fatto emergere le **criticità principali** di queste ultime, in particolare la loro potenziale difficoltà nel modellare *dipendenze temporali a lungo raggio* e i limiti legati alla loro *natura sequenziale*. Da qui è nata l'esigenza di investigare paradigmi alternativi come i **Transformer**.

Sulla base delle conoscenze acquisite, sono stati definiti i **requisiti principali** del framework sperimentale. I requisiti funzionali includevano la capacità di gestire ed elaborare *serie temporali complesse* (normalizzazione, creazione di lag), l'implementazione di diverse *architetture neurali* e la visualizzazione dei risultati per un'*analisi comparativa*. I requisiti

non funzionali si sono concentrati sulla *riproducibilità degli esperimenti*, la *robustezza della valutazione* (tramite un'adeguata suddivisione dei dati) e la *comparabilità oggettiva* delle performance dei modelli.

Il lavoro è stato organizzato attorno a quattro **componenti modellistiche principali**, sviluppate in un ambiente Python. L'infrastruttura tecnologica si è avvalsa di librerie standard del settore come Pandas e NumPy per la manipolazione dei dati, Scikit-learn per le operazioni di *pre-processing* e la valutazione, e il framework PyTorch (o TensorFlow) per la costruzione e l'addestramento delle reti neurali. Il flusso di lavoro è stato standardizzato:

- Acquisizione e analisi della serie storica grezza;
- *Pre-processing* e *feature engineering* per trasformare i dati in un formato adatto ai modelli;
- Addestramento e validazione dei quattro modelli: **LSTM (baseline)**, **Transformer Encoder-Decoder**, **Transformer Decoder-Only** e **Chronos**;
- Test e valutazione finale su un *set di dati inedito* per confrontare le performance in modo imparziale.

L'intero processo di **ricerca e sviluppo** ha seguito un approccio *incrementale e iterativo*, ispirato alla *metodologia sperimentale* tipica del *machine learning*. Ogni modello è stato costruito, testato e affinato progressivamente. I cicli di sviluppo sono stati guidati dall'analisi delle *metriche di errore* sul set di validazione (es. MAE, RMSE), il cui feedback è stato utilizzato per guidare l'*ottimizzazione degli iperparametri* e le eventuali modifiche architettonali. Questo metodo ha permesso di **affinare sistematicamente ogni modello** per massimizzarne le potenzialità prima del confronto finale.

Nella progettazione dell'esperimento sono stati adottati anche alcuni **principi guida fondamentali** della ricerca scientifica. Tra questi, il principio *KISS (Keep It Simple, Stupid)*, per la semplicità e poi il principio *DRY (Don't Repeat Yourself)*, per sssicurare che ogni pezzo di conoscenza o logica nel codice abbia una sola e unica rappresentazione.

Struttura della tesi

La tesi è articolata in cinque capitoli, ognuno dei quali analizza una fase specifica del progetto: si parte dallo stato dell'arte, per poi proseguire con la realizzazione dei modelli, la valutazione dei risultati ottenuti e l'individuazione di possibili miglioramenti.

Il **Capitolo 1** presenta lo stato dell'arte, introducendo i principali approcci utilizzati per la previsione di serie temporali, con particolare attenzione ai modelli basati su deep learning, come le reti LSTM. Viene inoltre motivato l'interesse verso l'utilizzo dei Transformer in questo ambito.

Il **Capitolo 2** è dedicato all'approfondimento teorico dell'architettura Transformer. Ne vengono analizzati i meccanismi fondamentali, come l'attenzione, la struttura Encoder–Decoder e il positional encoding, evidenziando i vantaggi e le sfide legate all'applicazione di questi modelli alle serie temporali.

Il **Capitolo 3** descrive il dataset utilizzato e le tecniche di pre-processing applicate per preparare i dati. Presenta in dettaglio le tre architetture implementate (LSTM, Transformer encoder-decoder, Transformer decoder-only) e il modello Chronos. Conclude con il setup sperimentale e le metriche adottate per valutare le performance.

Il **Capitolo 4** si concentra sulla valutazione dei risultati ottenuti. Le performance predittive dei vari modelli vengono confrontate utilizzando metriche quantitative e analisi qualitative, al fine di identificare punti di forza e criticità di ciascun approccio.

Il **Capitolo 5** chiude la tesi con una riflessione complessiva sul lavoro svolto, sintetizzando i risultati ottenuti, rispondendo alla domanda di ricerca iniziale e indicando possibili sviluppi futuri.

Indice

Introduzione	i
Obiettivi del lavoro	ii
Metodologia adottata	ii
Struttura della tesi	iv
1 Stato dell'arte delle previsioni su serie temporali	1
1.1 L'Evoluzione dei Metodi di Previsione	2
1.2 L'Emergere dei Transformers	9
2 I transformer	11
2.1 Il Limite della Ricorrenza e l'origine dei transformer	11
2.2 Self-attention	12
2.2.1 Limite della self-attention, l'ordine	13
2.2.2 Il Positional Encoding	14
2.2.3 Key, query e value	17
2.2.4 Multi-head self-attention	18
2.3 L'architettura originale del transformer	20
2.3.1 Masked self-attention	23
2.3.2 L'addestramento e la predizione	23
2.3.3 Adattamento dei transformers per le serie storiche	25
3 Sviluppo e implementazione dei modelli	27
3.1 Il Dataset	27
3.1.1 Origine e Contesto del Dataset	27
3.1.2 Struttura e Analisi Esplorativa	28
3.2 Preprocessing e Feature Engineering	29
3.2.1 Preprocessing	29
3.2.1.1 Selezione Iniziale e Gestione dei Dati Mancanti	29
3.2.1.2 Analisi di Correlazione per la Selezione delle Feature	29
3.2.2 Feature Engineering	32
3.2.2.1 Feature Basate sui Lag (Lag Features)	32
3.2.2.2 Feature Basate su Finestre Mobili (Rolling Window Features)	32
3.2.2.3 Feature Temporali (Time-based Features)	33
3.2.2.4 Feature Avanzate	33
3.2.2.5 Gestione dei Valori Mancanti Finali	33
3.2.3 Normalizzazione dei dati	33
3.3 Architetture	34
3.3.1 LSTM	34
3.3.2 Encoder-Decoder	36
3.3.3 Decoder	42
3.3.4 Chronos	46
3.4 Ambiente utilizzato e setup per il training	49
3.4.1 Training	49
3.4.1.1 Ciclo di Cross-Validation a Finestra Mobile	49
3.4.1.2 Addestramento, Ottimizzazione e Valutazione	49
3.4.1.3 Calcolo delle Prestazioni Finali	50
3.4.2 Ottimizzazione degli Iperparametri con Optuna	50

4 Analisi dei risultati	53
4.1 Introduzione	53
4.2 Metodologia di valutazione	54
4.2.1 Metriche di performance	54
4.2.2 Strategia di convalida	55
4.2.3 Strumenti di analisi visiva	55
4.3 Risultati LSTM	56
4.4 Risultati Transformer Encoder-Decoder	59
4.5 Risultati Transformer Decoder	62
4.6 Risultati Transformer Chronos	66
4.7 Analisi Comparativa e Discussione	68
4.7.1 Confronto Quantitativo	68
5 Conclusioni	71
5.1 Contributi Originali	71
5.2 Limiti e Sviluppi Futuri	72
Bibliografia	73

Elenco delle figure

1.1	Tasso di cambio dell'euro	1
1.2	Evoluzione dei metodi	2
1.3	rappresentazione stagionalità e trend	4
1.4	Rappresentazione modello ARIMA	4
1.5	Albero decisionale - La concessione di un prestito	6
1.6	Esempio funzionamento Gradient Boosting	7
1.7	Struttura Long-Short Term Memory [6]	8
2.1	Differenza tra RNN e Transformers	13
2.2	Matrice del positional encoding per la sequenza 'I am a robot' con $d_{model} = 4$. Ogni riga rappresenta il vettore di encoding per la posizione corrispondente.	15
2.3	Grafico delle funzioni seno e coseno per diverse dimensioni del Positional Encoding, evidenziando l'unicità del vettore per ogni posizione.	16
2.4	Calcolo della rappresentazione contestuale per la parola "palla"	17
2.5	Calcolo della rappresentazione contestuale per la parola "canestro"	18
2.6	Architettura della Multi-Head Self-Attention, che mostra il flusso parallelo delle teste di attenzione, la concatenazione e la proiezione finale.	19
2.7	L'architettura completa del modello Transformer, che evidenzia i blocchi Encoder (a sinistra) e Decoder (a destra) [7]	20
2.8	L'architettura dell'encoder del modello Transformer [7]	21
2.9	L'architettura del decoder del modello Transformer [7]	22
2.10	Rappresentazione di una sequenza di input durante la fase di addestramento. [7]	24
2.11	Processo di generazione autoregressiva durante la fase di previsione. [7]	25
3.1	Andamento delle immatricolazioni mensili di autovetture in Italia (Gennaio 1990 - Dicembre 2018)	28
3.2	Matrice di correlazione (<i>heatmap</i>) delle immatricolazioni mensili tra le nazioni europee selezionate	31
3.3	Architettura del modello LSTM di <i>baseline</i> . Mostra la concatenazione di due strati LSTM, ciascuno seguito da Dropout, e un layer Dense finale.	34
3.4	Diagramma dell'architettura Transformer Encoder-Decoder implementata, che mostra i flussi di dati paralleli per l'Encoder e il Decoder.	41
3.5	Diagramma di flusso dell'architettura Transformer.	43
3.6	fasi effettuate dal modello chronos per la generazione di una previsione [8]	47
4.1	Andamento della loss di training e di validazione del modello LSTM nell'ul- timo fold.	56
4.2	Confronto tra i valori reali e i valori predetti dal modello LSTM su tutti i fold di test.	57
4.3	Andamento dei residui del modello LSTM nel tempo.	58
4.4	Scatter plot dei valori reali vs. i valori predetti dal modello LSTM.	58
4.5	Andamento della loss di training e di validazione (Weighted MSE) del modello Transformer Encoder-Decoder nell'ultimo fold.	60
4.6	Confronto tra i valori reali e i valori predetti dal modello Transformer Encoder- Decoder su tutti i fold di test.	60
4.7	Andamento dei residui del modello Transformer Encoder-Decoder nel tempo.	61

4.8 Scatter plot dei valori reali vs. i valori predetti dal modello Transformer Encoder-Decoder.	61
4.9 Andamento della loss di training e di validazione del modello Transformer Decoder-only nell'ultimo fold.	63
4.10 Confronto tra i valori reali e i valori predetti dal modello Transformer Decoder-only su tutti i fold di test.	63
4.11 Andamento dei residui del modello Transformer Decoder-only nel tempo.	64
4.12 Scatter plot dei valori reali vs. i valori predetti dal modello Transformer Decoder-only.	65
4.13 Confronto tra i valori reali e i valori predetti dal modello Chronos (zero-shot) su tutti i fold di test.	66
4.14 Andamento dei residui del modello Chronos nel tempo.	67
4.15 Scatter plot dei valori reali vs. i valori predetti dal modello Chronos.	67
4.16 <i>Grafico a barre comparativo delle metriche chiave (RMSE e MAE Medi) per i quattro modelli.</i>	69

Elenco dei listati

3.1	Selezione delle feature basata sulla correlazione.	30
3.2	Implementazione del modello LSTM con TensorFlow/Keras.	36
3.3	Implementazione della classe PositionalEncoding in PyTorch.	37
3.4	Implementazione della classe TransformerAutoregressive in PyTorch.	38
3.5	Definizione della classe DecoderOnlyTransformer in PyTorch.	46
3.6	Definizione del modello chronos	47
3.7	Codice utilizzato per stabilizzare e gestire i trend	47
3.8	Codice utilizzato per ricavare l'input	48
3.9	Codice utilizzato per effettuare una previsione	48
3.10	Codice utilizzato per effettuare l'aggregazione delle Previsioni tramite Mediana	48
3.11	Codice utilizzato per definire l'ottimizzatore	49
3.12	Codice utilizzato per definire l'addestramento	50
3.13	Codice utilizzato per trovare le metriche	50
3.14	Codice utilizzato per definire le metriche finali	50

Capitolo 1

Stato dell'arte delle previsioni su serie temporali

L'analisi e la previsione delle **serie temporali** rappresentano una delle discipline più **cru-ciali** e **pervasive** nell'ambito dell'analisi dei dati. Una *serie temporale* (o *serie storica*) è una sequenza di osservazioni raccolte a intervalli di tempo regolari, che possono variare da millisecondi, nel caso di dati finanziari ad alta frequenza, a decenni, come nelle analisi climatologiche. Troviamo esempi di *serie temporali* in quasi ogni dominio della vita umana e della scienza: il prezzo giornaliero di un'azione, il numero di passeggeri mensili di una compagnia aerea, la temperatura media annuale del pianeta, i dati di vendita trimestrali di un'azienda o i segnali elettrici di un elettrocardiogramma.

Un esempio emblematico è rappresentato nel grafico sottostante, che mostra l'andamento di due *serie temporali finanziarie correlate* tra il 2009 e il 2015: il **tasso di cambio Dollar/Euro (USD/EUR)** e il **tasso di cambio effettivo nominale dell'Euro**. La linea **blu** mostra il tasso di cambio USD/EUR, mentre la linea **gialla** rappresenta il tasso di cambio effettivo nominale dell'euro [9].

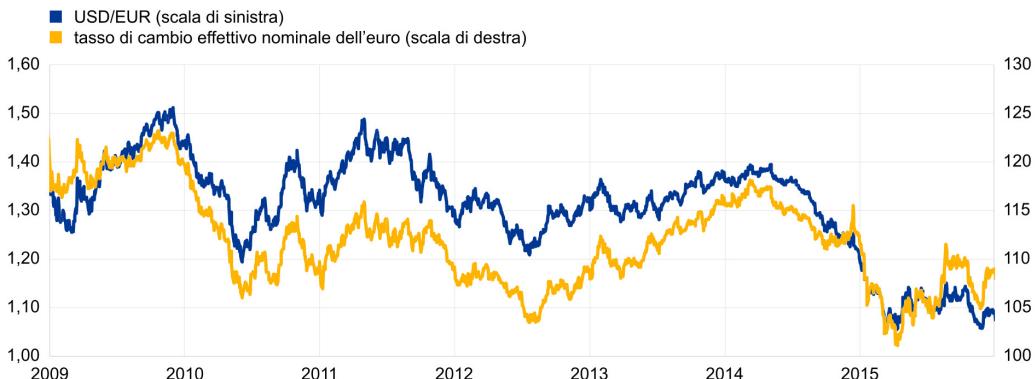


Figura 1.1: Tasso di cambio dell'euro

Questo grafico incapsula visivamente la natura di una **serie storica**: una successione di punti dati che fluttuano nel tempo, mostrando **trend**, **volatilità** e **complesse interdipendenze**. Osservando le due curve, si nota come i loro andamenti, pur non essendo identici, siano chiaramente **correlati**, evidenziando come l'analisi spesso debba tener conto di **più variabili contemporaneamente**.

L'importanza intrinseca delle *serie temporali* non risiede tanto nel dato passato, quanto nella sua capacità di **informare il futuro**. La *previsione* (o *forecasting*) è il processo che utilizza i dati storici e i modelli matematici per stimare e predire l'evoluzione futura di tali osservazioni. Questa capacità di *guardare avanti* non è un mero esercizio accademico, ma un'attività dal **valore strategico inestimabile**. In ambito aziendale, previsioni accurate sulla domanda permettono di ottimizzare la gestione delle scorte, pianificare la produzione e ridurre i costi, trasformandosi in un **vantaggio competitivo diretto**. In finanza, i modelli previsionali sono alla base delle strategie di *trading* e della gestione del rischio. In sanità pubblica, prevedere la diffusione di un'epidemia è fondamentale per allocare risorse e

preparare contromisure efficaci. In ingegneria, la *manutenzione predittiva* basata sull'analisi dei dati dei sensori previene guasti costosi e pericolosi.

Data la sua rilevanza, la ricerca di metodi previsionali sempre più **accurati** e **robusti** è stata una costante nella storia della statistica, dell'econometria e, più recentemente, dell'informatica.

Questo capitolo si propone di tracciare un **percorso evolutivo** di questa disciplina, partendo dalle fondamenta statistiche che ne hanno definito il rigore metodologico, passando per l'avvento dei modelli di *machine learning* che ne hanno ampliato le capacità *non lineari*, fino a giungere alle soglie della rivoluzione introdotta dalle architetture di *deep learning*. L'obiettivo è fornire un quadro completo dello **stato dell'arte**, creando il contesto necessario per comprendere l'importanza e l'innovazione dei modelli *Transformer*, che verranno discussi nel dettaglio nel capitolo successivo. Esploremo come ogni nuova famiglia di modelli abbia cercato di superare i limiti della precedente, in una continua rincorsa verso la rappresentazione più **fedele** e **predittiva** della complessa dinamica che governa i dati nel tempo.

1.1 L'Evoluzione dei Metodi di Previsione

La storia della previsione delle **serie temporali** è un viaggio attraverso decenni di **innovazione statistica e computazionale**. Ogni fase di questa evoluzione ha introdotto nuovi strumenti e paradigmi, ciascuno con i propri punti di forza e limiti, costruendo progressivamente le fondamenta metodologiche su cui poggiano le tecniche moderne.

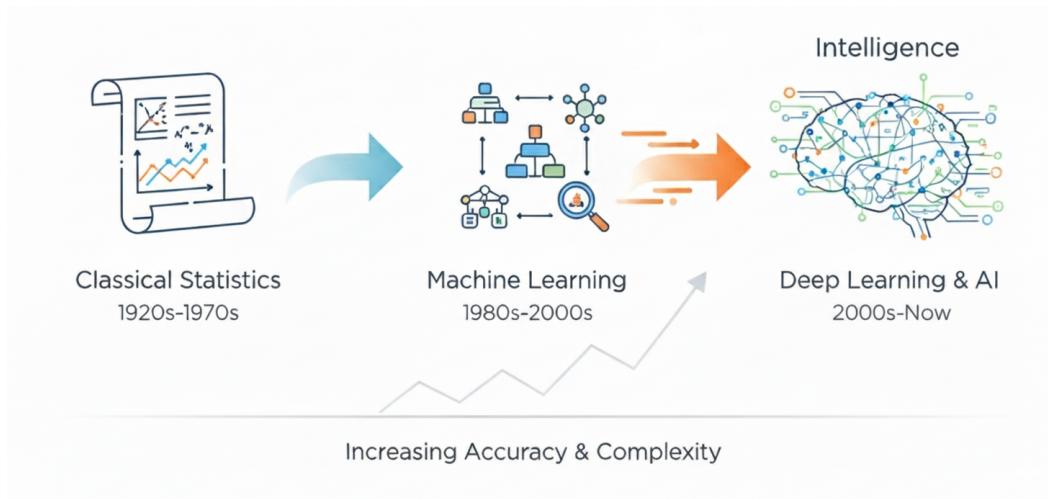


Figura 1.2: Evoluzione dei metodi

L'immagine sopra illustra il **percorso evolutivo** delle metodologie di previsione, delineando le principali **ere** che hanno caratterizzato questo campo.

- **Classical Statistics (1920s–1970s)**: Questa fase ha posto le fondamenta della previsione con modelli matematici rigorosi come ARIMA [1] e lo smorzamento esponenziale. Tali metodi, pur essendo **potenti**, richiedevano una modellazione **manuale** e si basavano su assunzioni precise riguardo la *struttura dei dati*.
- **Machine Learning (1980s–2000s)**: L'aumento della potenza di calcolo ha introdotto algoritmi flessibili come Random Forest, Gradient Boosting e gli Alberi decisionali. Questi modelli apprendono **relazioni complesse e non lineari**, superando alcuni limiti statistici, ma necessitano di un attento lavoro di *feature engineering* per gestire la **dipendenza temporale** [10].
- **Deep Learning e Intelligenza Artificiale (2000s–oggi)**: Quest'era è definita da architetture avanzate come le LSTM [3] e i Transformer [2], che apprendono le **dipendenze temporali** direttamente dai *dati grezzi*. Hanno rivoluzionato il campo,

gestendo la **sequenzialità** e le **dipendenze a lungo termine** con una capacità di modellazione **senza precedenti**.

Le Fondamenta Statistiche: Medie Mobili e Smorzamento Esponenziale

Agli albori della previsione, l'approccio più intuitivo era basato sull'idea che il futuro potesse essere approssimato da una **media del passato**. I primi e più semplici modelli, come il metodo *Naive*, postulavano che la previsione futura fosse semplicemente l'ultima osservazione disponibile ($\hat{y}_{t+1} = y_t$). Sebbene rudimentale, questo metodo serve ancora oggi come importante **baseline** per valutare modelli più complessi.

Un primo passo verso una maggiore sofisticazione è rappresentato dalle *Medie Mobili (Moving Averages)*. Invece di considerare solo l'ultimo punto, questo metodo calcola la media di un numero fisso k di osservazioni recenti. Ciò permette di **smussare le fluttuazioni casuali** e di identificare più chiaramente il trend di fondo. Tuttavia, la media mobile semplice presenta due **svantaggi**:

- Assegna lo stesso peso a tutte le osservazioni nel suo periodo di riferimento
- Soffre di un *ritardo intrinseco* nel reagire ai cambiamenti.

Per superare questi limiti, sono state introdotte le tecniche di *Smorzamento Esponenziale (Exponential Smoothing)*. Il metodo di *smorzamento esponenziale semplice* assegna pesi decrescenti in modo esponenziale alle osservazioni passate, dando quindi più importanza ai dati più recenti. La previsione è una media ponderata di tutte le osservazioni passate, con un unico parametro, α , che controlla il tasso di decadimento dei pesi.

Questa idea è stata poi estesa da **Charles Holt** per includere esplicitamente la gestione del *trend* (introducendo un secondo parametro, β), e successivamente da **Holt** e il suo studente **Peter Winters** per incorporare anche la *stagionalità* (con un terzo parametro, γ). I modelli *Holt-Winters* sono diventati uno strumento estremamente **popolare e robusto**, capace di modellare serie storiche che presentano contemporaneamente un livello di base, una tendenza e cicli stagionali.

Nell'immagine seguente, il concetto di **trend** e **stagionalità** viene visualizzato in una serie storica. La freccia rossa obliqua indica chiaramente un **trend** crescente, rappresentando la direzione generale della serie nel lungo periodo. Il pattern ripetitivo di picchi e valli, racchiuso nell'area gialla, illustra la **stagionalità**, un comportamento che si manifesta a intervalli regolari. Queste due componenti sono fondamentali per comprendere la dinamica di molte serie storiche e la loro capacità di essere modellate esplicitamente è ciò che rende i modelli come **Holt-Winters** ancora oggi ampiamente utilizzati in contesti industriali per la loro efficacia.

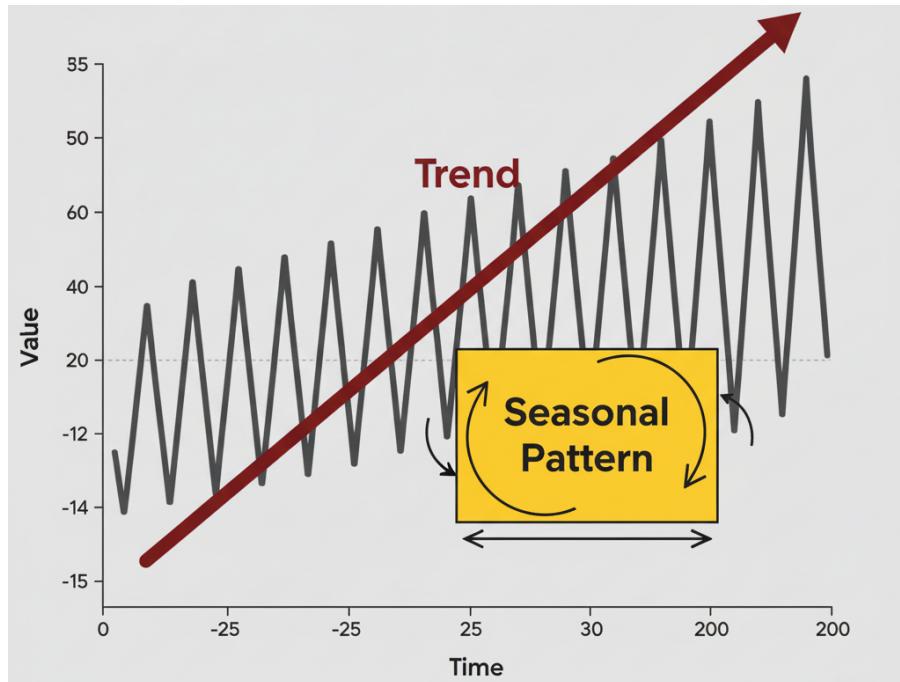


Figura 1.3: rappresentazione stagionalità e trend

La Rivoluzione Box-Jenkins: I Modelli ARIMA

Una vera e propria svolta metodologica si ebbe negli anni '70 con il lavoro di **George Box** e **Gwilym Jenkins**, che sistematizzarono un approccio rigoroso per l'identificazione, la stima e la validazione di modelli per serie temporali. Il frutto del loro lavoro sono i modelli **ARIMA** (*AutoRegressive Integrated Moving Average*), che rappresentano una delle famiglie di modelli statistici più **influenti e potenti** [1].

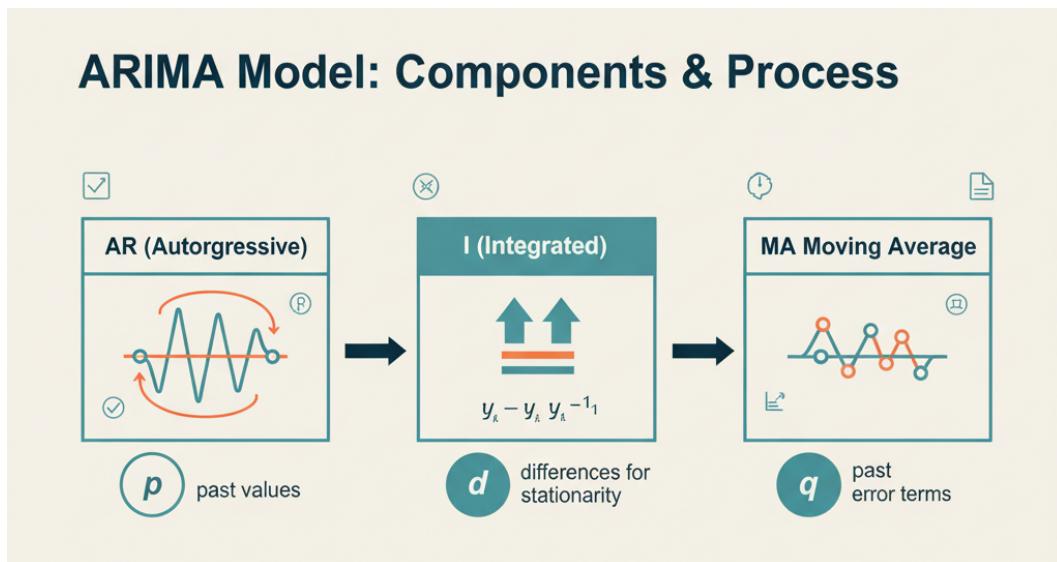


Figura 1.4: Rappresentazione modello ARIMA

I modelli ARIMA si basano sul presupposto che una serie temporale possa essere modellata come una combinazione lineare dei suoi valori passati e degli *errori di previsione* passati. L'acronimo **ARIMA** racchiude tre componenti fondamentali:

- **AR (Autoregressivo)**: di ordine p , assume che il valore attuale della serie dipenda linearmente dai suoi p valori precedenti. Matematicamente, y_t è funzione di $y_{t-1}, y_{t-2}, \dots, y_{t-p}$.

- **I (Integrato)**: di ordine d , rende la serie *stazionaria* attraverso un processo di differenziazione, ad esempio $y'_t = y_t - y_{t-1}$, rimuovendo trend e stabilizzando la serie.
- **MA (Media Mobile)**: di ordine q , assume che il valore attuale dipenda dagli *errori di previsione* commessi nei q istanti precedenti.

La metodologia **Box-Jenkins** prevede un ciclo iterativo in tre fasi:

1. **Identificazione**: scelta dei valori da assegnare a " p,d,q " analizzando le funzioni di autocorrelazione *ACF* e autocorrelazione parziale *PACF*, che mostrano la correllazione con i valori precedenti.
2. **Stima**: calcolo dei parametri del modello.
3. **Diagnostica**: verifica che i residui del modello si comportino come *rumore bianco*, indicando che il modello ha catturato tutta la struttura informativa presente nei dati.

Per superare il limite della gestione della stagionalità sono state sviluppate varianti come i modelli **SARIMA** (*Seasonal ARIMA*) in grado di gestire la stagionalità.

Oltre l'Univariato: Modelli Multivariati e a Spazio degli Stati

I modelli **ARIMA**, pur essendo potenti, sono intrinsecamente *univariati*: modellano una serie temporale basandosi unicamente sulla sua storia passata. Tuttavia, in molti scenari reali, l'andamento di una variabile è influenzato anche da altre variabili. Per affrontare questa complessità, sono stati sviluppati modelli *multivariati*. Il più noto è il modello **VAR** (*Vector Autoregression*), che generalizza il modello autoregressivo per analizzare le interdipendenze dinamiche tra più serie temporali contemporaneamente. In un modello **VAR**, ogni variabile è modellata come una funzione lineare dei valori passati di sé stessa e di tutte le altre variabili nel sistema.

L'Avvento del Machine Learning

Con l'espansione esponenziale della potenza computazionale e la crescente disponibilità di grandi dataset, il machine learning ha iniziato a essere impiegato con successo nella previsione temporale, segnando una significativa evoluzione nel campo. Questa transizione ha rappresentato un cambio di paradigma rispetto ai metodi statistici classici, che per decenni avevano dominato la disciplina. A differenza di questi ultimi, che spesso richiedono ipotesi stringenti sulla distribuzione dei dati (ad esempio, normalità) e sulla loro stazionarietà (media e varianza costanti nel tempo), i modelli di machine learning sono tipicamente non parametrici. Questa caratteristica li rende intrinsecamente più flessibili e capaci di apprendere relazioni complesse e non lineari direttamente dai dati, senza imporre a priori una specifica struttura funzionale. La loro capacità di adattarsi a pattern intricati li ha resi strumenti preziosi per affrontare la crescente complessità delle serie storiche moderne, dove le relazioni tra variabili possono essere multifattoriali e non lineari.

Un vantaggio cruciale del machine learning è la sua robustezza di fronte a dati incompleti o rumorosi e la capacità di integrare un'ampia gamma di variabili. Queste variabili esterne, non direttamente parte della serie temporale principale, possono fornire un contesto prezioso, migliorando significativamente la precisione delle previsioni. Ad esempio, per prevedere le vendite di un prodotto, oltre ai dati storici di vendita, si possono includere variabili come il prezzo, le promozioni, i dati meteorologici, gli eventi macroeconomici o i comportamenti dei concorrenti. Questa flessibilità nell'incorporare informazioni eterogenee distingue nettamente il machine learning da molti modelli statistici univariati. Di seguito sono spiegati i metodi più efficaci

Support Vector Machines (**SVR**)

Estendono le SVM, nate per la classificazione, al contesto della regressione [11]. L'obiettivo è trovare una funzione che approssimi i dati entro un margine di tolleranza (ϵ -tube), minimizzando solo gli errori al di fuori di questo margine. Questo approccio le rende robuste agli outlier e adatte a catturare relazioni non lineari grazie ai kernel.

Alberi Decisionali

Gli alberi decisionali suddividono ricorsivamente il dataset in base a semplici regole derivate dai dati, creando un modello intuitivo e facilmente interpretabile. Gestiscono bene dati misti e non richiedono normalizzazione, ma un singolo albero può soffrire di overfitting e risultare poco stabile.

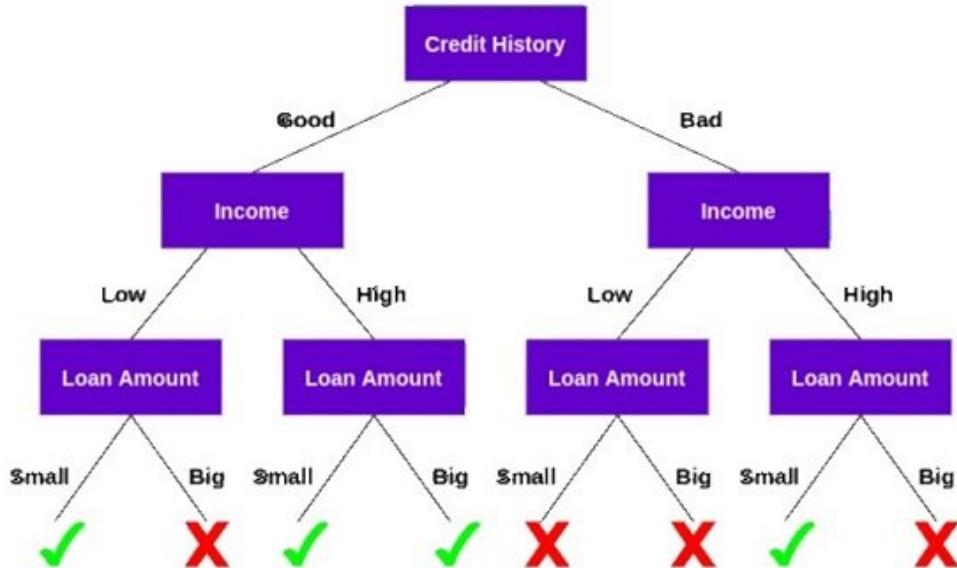


Figura 1.5: Albero decisionale - La concessione di un prestito

Nella figura soprastante, un esempio di albero decisionale che decide se il prestito del cliente debba essere approvato o meno [12].

Random Forest

È un esempio di algoritmo di **bagging**. Costruisce un "bosco" di alberi decisionali, ognuno addestrato su un sottoinsieme casuale dei dati e considerando un sottoinsieme casuale delle feature a ogni split. Le previsioni di tutti gli alberi vengono poi aggregate (ad esempio, tramite media per la regressione) per produrre il risultato finale. Questo riduce drasticamente l'overfitting e migliora la robustezza e la precisione del modello, pur mantenendo una buona interpretabilità [10].

Gradient Boosting

A differenza del **bagging**, il boosting costruisce i modelli in modo sequenziale. Ogni nuovo modello cerca di correggere gli errori (i "residui") commessi dai modelli precedenti. In questo processo iterativo, i modelli successivi si concentrano sui dati che i modelli precedenti hanno classificato o previsto erroneamente, imparando gradualmente a ridurre l'errore complessivo. Le implementazioni moderne come **XGBoost** [13] e **LightGBM** hanno ottimizzato questo processo, introducendo tecniche per migliorare la velocità e l'accuratezza, rendendoli tra gli algoritmi più performanti per un'ampia gamma di problemi di machine learning, inclusa la previsione [14]. Di seguito è rappresentato un esempio del suo funzionamento [15].

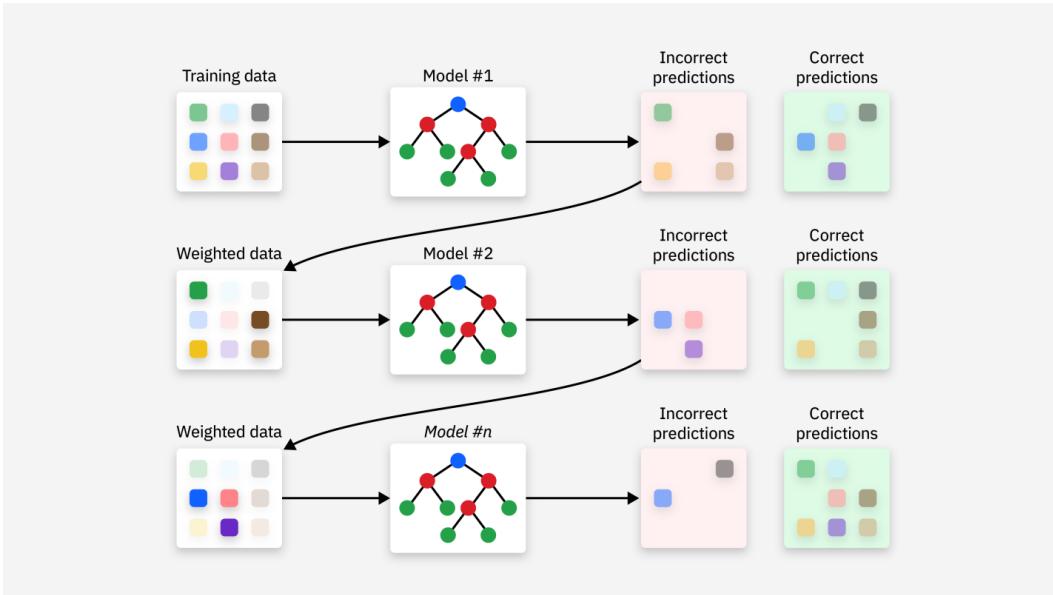


Figura 1.6: Esempio funzionamento Gradient Boosting

Questi modelli sono potenti nel gestire **feature esogene** (scollegate), catturare **interazioni complesse** e non richiedono trasformazioni manuali. Tuttavia, trattano la previsione come un problema di *regressione statica*, dove ogni istante temporale è visto come un'osservazione indipendente, perdendo così la naturale sequenzialità dei dati. Per ovviare a ciò, è necessaria una **feature engineering** mirata (lag, medie mobili, ecc.).

L'Era del Deep Learning: Le Reti Neurali Ricorrenti (RNN)

Il passo più **rivoluzionario** nella previsione di serie temporali è stato l'ingresso del **deep learning**. Le architetture di reti neurali classiche, come i *Multi-Layer Perceptrons* (MLP), pur essendo potenti approssimatori di funzioni, trattano ogni input in modo indipendente. Questa caratteristica le rende intrinsecamente inadatte a modellare dati sequenziali, dove l'ordine e il contesto storico sono fondamentali. Un MLP, di fronte a una serie storica, non ha alcun meccanismo per ricordare le osservazioni passate e usarle per informare la previsione futura; per esso, ogni istante temporale è un evento a sé stante.

Per superare questo limite concettuale, sono state introdotte le **Reti Neurali Ricorrenti (RNN)**. La loro innovazione fondamentale è l'introduzione di un **ciclo interno** nella loro architettura, che consente la persistenza dell'informazione, creando una forma di **memoria temporale**. In una RNN, l'output di un passo temporale non viene solo usato per la previsione, ma viene anche reimmesso nella rete come parte dell'input per il passo successivo. Questo meccanismo di feedback permette alla rete di mantenere uno *stato nascosto* (*hidden state*), un vettore numerico che agisce come un riassunto cumulativo di tutte le informazioni rilevanti della sequenza processata fino a quel momento. In questo modo, la previsione al tempo t non dipende solo dall'input al tempo t , ma dall'intera storia passata della serie, incapsulata nello stato nascosto.

Tuttavia, le RNN "semplici" soffrono di un grave problema noto come **gradiente che svanisce o esplode** (*vanishing/exploding gradient*). Durante l'addestramento, che avviene tramite un processo chiamato *Backpropagation Through Time* (BPTT), i gradienti dell'errore vengono propagati all'indietro attraverso la sequenza. Nelle sequenze lunghe, questi gradienti, moltiplicati ripetutamente ad ogni passo temporale, possono diventare esponenzialmente piccoli (svanire) o grandi (esplodere). Un gradiente che svanisce impedisce alla rete di apprendere le dipendenze a lungo termine, poiché l'errore non riesce a propagarsi abbastanza indietro nel tempo per aggiornare i pesi dei passi iniziali. Di conseguenza, la rete diventa "miope", incapace di connettere eventi distanti nel tempo.

La Soluzione: Long Short-Term Memory (LSTM)

Per risolvere il problema della memoria a breve termine delle RNN, **Sepp Hochreiter** e **Jürgen Schmidhuber** hanno introdotto l'architettura **Long Short-Term Memory (LSTM)**. Le LSTM sono un tipo specializzato di RNN, progettate esplicitamente per evitare i problemi del gradiente e per apprendere dipendenze a lungo termine.

Il segreto del successo delle LSTM risiede nella loro complessa unità di base, la **cella di memoria**, che è molto più sofisticata di quella di una RNN standard. Oltre allo *stato nascosto* (*hidden state*), una cella LSTM introduce uno *stato di cella* (*cell state*), che può essere visto come un ulteriore memoria.

La caratteristica più importante delle LSTM è la loro capacità di aggiungere o rimuovere informazioni dallo stato di cella in modo selettivo, attraverso delle strutture chiamate **porte** (*gates*). Una porta è un meccanismo composto da una rete neurale con una funzione di attivazione sigmoide, che produce un output tra 0 e 1. Questo valore viene poi usato per "regolare" il flusso di informazioni: un valore di 0 significa "non far passare nulla", mentre un valore di 1 significa "lascia passare tutto".

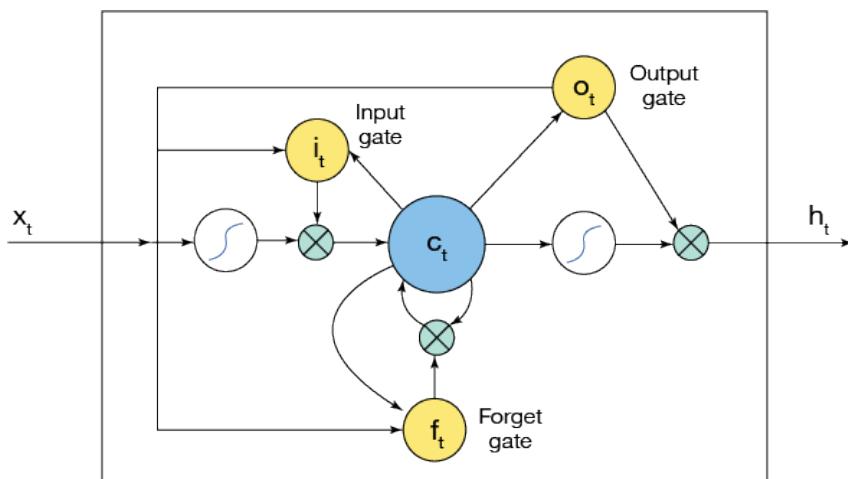


Figura 1.7: Struttura Long-Short Term Memory [6]

Una cella LSTM è dotata di tre porte principali, ognuna con un ruolo specifico:

- **Forget Gate (Porta della Dimenticanza):** Questa porta decide quali informazioni devono essere eliminate dallo stato di cella.
- **Input Gate (Porta di Input):** Questa porta decide quali nuove informazioni dell'input corrente sono abbastanza importanti da essere memorizzate nello stato di cella.
- **Output Gate (Porta di Output):** Questa porta determina quale parte dello stato di cella corrente deve essere utilizzata per calcolare l'output (la previsione) e il nuovo stato nascosto che verrà passato al passo temporale successivo.

Questo sofisticato meccanismo di *gating* permette alle LSTM di gestire la memoria in modo intelligente: possono mantenere informazioni cruciali per centinaia di passi temporali, scartando quelle irrilevanti e aggiungendo nuove osservazioni pertinenti.

Gated Recurrent Units (GRU)

Una variante popolare e leggermente più semplice delle LSTM sono le **Gated Recurrent Units (GRU)**. Le GRU combinano la forget gate e la input gate in un'unica "update gate" e fondono lo stato di cella e lo stato nascosto. Questo le rende computazionalmente più efficienti e più veloci da addestrare, pur mantenendo performance molto simili a quelle delle LSTM su molti compiti [16].

Grazie a queste innovazioni, le architetture **LSTM** e **GRU** sono diventate per anni lo **standard de facto** per la modellazione sequenziale. La loro capacità di catturare **pattern**

complessi e dipendenze a lungo termine ha permesso di superare i metodi tradizionali in una vasta gamma di applicazioni, stabilendo un nuovo e potente stato dell'arte per la previsione di serie temporali.

1.2 L'Emergere dei Transformers

Nonostante l'enorme successo delle architetture ricorrenti come le *LSTM*, anche queste presentano dei **limiti**. La loro natura **intrinsecamente sequenziale**, che processa i dati un passo alla volta, rappresenta un **collo di bottiglia computazionale** che ne limita la *parallelizzazione* e, di conseguenza, la **scalabilità** su sequenze molto lunghe. Inoltre, sebbene progettate per mitigare il problema, anche le *LSTM* possono ancora faticare a catturare **relazioni tra punti temporalmente molto distanti** tra loro.

In questo contesto, una nuova architettura, nata nel 2017 nel campo del *Natural Language Processing (NLP)*, ha iniziato a mostrare un **potenziale rivoluzionario** anche per altri domini sequenziali. Si tratta del **Transformer** [2].

L'innovazione fondamentale del *Transformer* è l'**abbandono completo della ricorrenza** a favore di un meccanismo chiamato *self-attention* (o *auto-attenzione*). Questo meccanismo permette al modello di **pesare l'importanza** di tutte le altre posizioni nella sequenza di input quando elabora una specifica posizione, **indipendentemente dalla loro distanza**. In pratica, il modello può “guardare” simultaneamente a tutti i punti passati e decidere quali siano i più **rilevanti** per la previsione, superando i limiti delle *dipendenze a lungo termine*. Questo **cambio di paradigma** non solo ha portato a risultati **all'avanguardia** nel mondo del testo, ma ha aperto la strada a una **nuova era di ricerca** anche per le serie temporali. L'idea di applicare un modello capace di **catturare relazioni complesse** in modo *parallelo* e *non sequenziale* ha dato il via a un'intensa attività di ricerca per adattare l'architettura *Transformer* al **dominio numerico e continuo** delle serie storiche.

Il capitolo successivo spiegherà nel dettaglio questa architettura, **esplorandone il funzionamento** e le **varianti proposte** per affrontare le sfide della previsione temporale.

Capitolo 2

I transformer

È giunto il momento di analizzare l'architettura che ha definito un nuovo **stato dell'arte** non solo nel suo dominio di origine, il *Natural Language Processing (NLP)*, ma in quasi ogni campo che si occupa di **dati sequenziali**: il *Transformer*. Questo capitolo si propone di sezionare in dettaglio questa architettura, partendo dalle **limitazioni dei modelli precedenti** che ne hanno motivato la creazione, fino ad analizzarne ogni **componente fondamentale**. Infine, verrà discusso come questo potente *paradigma* possa essere adattato dalle complessità del linguaggio umano a quelle, altrettanto sfidanti, delle **serie temporali**.

2.1 Il Limite della Ricorrenza e l'origine dei transformer

Per comprendere appieno la **portata rivoluzionaria del Transformer**, è indispensabile analizzare le **sfide intrinseche delle architetture che lo hanno preceduto**. Le *Reti Neurali Ricorrenti*, in particolare le *LSTM* e le *GRU*, hanno rappresentato per anni la soluzione d'elezione per la modellazione di sequenze. La loro capacità di mantenere uno "stato nascosto" (*hidden state*) che si aggiorna a ogni passo temporale ha permesso, per la prima volta, di **catturare le dipendenze temporali** in modo efficace. Tuttavia, il meccanismo stesso che conferisce loro questo potere – la *ricorrenza* – è anche la **fonte delle loro più grandi limitazioni**.

Il Problema del Gradiente e le Dipendenze a Lungo Termine

Come discusso nel capitolo precedente, il **problema del gradiente che svanisce o esplode** (*vanishing/exploding gradient*) è una sfida fondamentale nell'addestramento delle *RNN*. Durante il processo di *backpropagation*, il gradiente dell'errore viene propagato all'indietro attraverso ogni passo temporale della sequenza. Matematicamente, questo comporta una serie di moltiplicazioni matriciali. Se i valori in queste matrici sono, in media, inferiori a 1, il gradiente si ridurrà esponenzialmente fino a diventare quasi nullo, impedendo ai pesi dei primi passi della sequenza di essere aggiornati. La rete, di fatto, diventa **incapace di apprendere le connessioni tra eventi temporalmente distanti**. Sebbene le architetture come le *LSTM*, con i loro *meccanismi di gating*, siano state progettate proprio per mitigare questo problema, non lo eliminano del tutto. Su sequenze estremamente lunghe, anche le *LSTM* possono faticare a propagare l'informazione in modo efficace, perdendo il contesto a lungo raggio.

Il Collo di Bottiglia Computazionale: La Mancanza di Parallelizzazione

La limitazione più stringente delle *RNN* nell'era moderna del *deep learning* è però di natura computazionale. La loro struttura è intrinsecamente sequenziale: per calcolare lo stato nascosto al tempo t , è necessario aver già calcolato lo stato nascosto al tempo $t-1$. Questo processo, per definizione, non può essere parallelizzato. Ogni passo deve attendere il completamento del precedente, creando un **collo di bottiglia** che impedisce di sfruttare appieno la potenza delle moderne unità di calcolo hardware come le *GPU (Graphics Processing Unit)*, progettate per eseguire migliaia di operazioni in parallelo. In un'epoca in cui i modelli vengono addestrati su *dataset* di dimensioni immense, questa limitazione si traduce in **tempi di addestramento proibitivi** e in una ridotta capacità di scalare a problemi più complessi.

Le Origini dell'Attention

Prima dell'avvento del *Transformer*, la comunità scientifica aveva già iniziato a esplorare meccanismi per superare questi limiti. Una delle idee più promettenti è stata quella dell'*attention*. Introdotti per la prima volta nel contesto della *traduzione automatica* con architetture *RNN Encoder-Decoder*, i meccanismi di *attention* (come quello di *Bahdanau*) permettevano al *decoder*, durante la generazione di una parola, di **"prestare attenzione"** a tutte le parole della frase di input, **pesandone l'importanza in modo dinamico** [17]. Invece di fare affidamento solo sull'ultimo stato nascosto dell'*encoder* (un riassunto compreso e potenzialmente imperfetto dell'intera sequenza), il *decoder* poteva creare scorciatoie, **collegamenti diretti a ogni elemento dell'input**.

Questo approccio si è rivelato estremamente potente, migliorando notevolmente le performance. Tuttavia, era ancora vincolato a un'architettura di base ricorrente. La domanda fondamentale che ha portato alla nascita del *Transformer* è stata tanto semplice quanto audace: **"Cosa succederebbe se eliminassimo del tutto la ricorrenza e costruissimo un'architettura basata esclusivamente sul meccanismo di attention?"**. La risposta, fornita nel celebre paper del 2017 *"Attention Is All You Need"* di Vaswani et al. [2], ha cambiato per sempre il panorama del *deep learning*.

2.2 Self-attention

Per comprendere appieno l'innovazione radicale introdotta dalla *self-attention*, è utile partire da un problema classico che affligge le architetture sequenziali come le *LSTM*: la **gestione delle dipendenze a lungo raggio**. I modelli ricorrenti, per loro natura, processano le informazioni in sequenza, mantenendo una **"memoria"** che può indebolirsi con l'aumentare della distanza tra le parole.

Consideriamo, ad esempio, la seguente frase, volutamente complessa:

"Il relatore, che nel corso della sua carriera ha scritto numerosi libri acclamati dalla critica, ha fondato startup innovative nel settore tecnologico e ha tenuto conferenze in decine di università prestigiose in tutto il mondo, ora inizia il suo discorso."

Il Limite della Memoria Sequenziale (*LSTM*)

Il problema fondamentale qui è la **concordanza grammaticale** tra il soggetto, *relatore*, e il suo verbo, *inizia*, che sono separati da una distanza enorme. Un modello *LSTM* analizza la frase parola per parola, aggiornando il suo stato di memoria a ogni passo.

- All'inizio, leggendo "Il relatore", la sua *memoria* registra correttamente che il soggetto è singolare.
- Successivamente, viene inondata da una lunga clausola intermedia ricca di sostantivi al plurale (*libri, startup, conferenze, università*). Il rischio concreto è che l'**informazione originale e cruciale ("il soggetto è singolare") venga progressivamente diluita o sovrascritta**.
- Quando il modello arriva finalmente al verbo, la sua *memoria* potrebbe essere **"confusa"**, portandolo a predire erroneamente la forma plurale **"iniziano"**. **L'informazione critica si è "svanita"** lungo il percorso sequenziale.

La Soluzione: Uno Sguardo d'Insieme (*Self-Attention*)

Un modello basato sulla *self-attention*, come il *Transformer*, aggira completamente questo problema perché non legge la frase in sequenza, ma la **guarda tutta insieme in un colpo solo**.

Quando deve determinare la forma del verbo, la *self-attention* **crea una connessione diretta** tra la posizione del verbo e tutte le altre parole della frase. Calcolando dei *punteggi di rilevanza*, **scopre immediatamente che la parola più importante per la concordanza è il soggetto, "relatore"**, indipendentemente dal fatto che si trovi a 20 parole di distanza.

In pratica, la *self-attention* crea una scorciatoia che **scavalca l'intera clausola intermedia**, permettendo al modello di **vedere la relazione relatore con la parola inizia in modo chiaro e diretto**. È proprio questa capacità di **creare una rete di connessioni dinamiche e non sequenziali** tra le parole che le permette di catturare il contesto in modo così efficace e che la pone al centro dell'architettura del *Transformer*. Un altro esempio grafico è rappresentato dall'immagine sottostante che spiega in maniera chiara come le RNN analizzino una parola alla volta mentre i Transfomer prendano in considerazione tutta la frase [18].

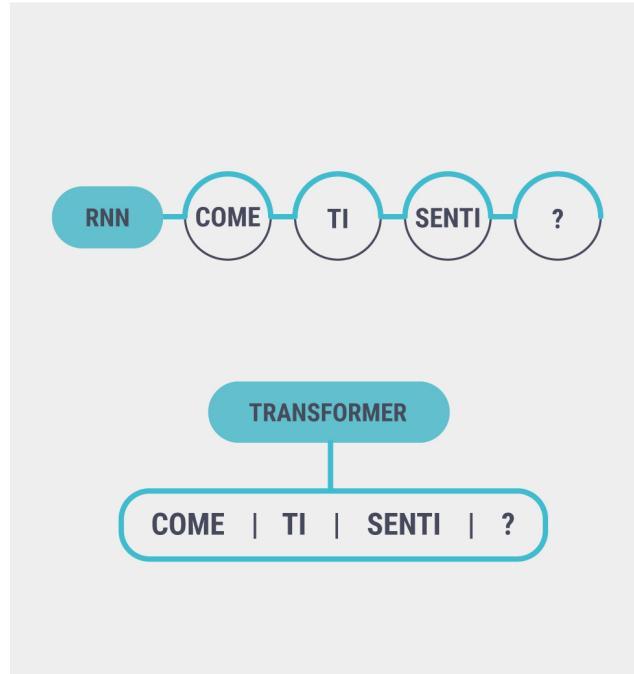


Figura 2.1: Differenza tra RNN e Transformers

2.2.1 Limite della self-attention, l'ordine

Nonostante la sua straordinaria capacità di catturare dipendenze contestuali a lungo raggio e di parallelizzare i calcoli, il meccanismo di *self-attention*, nella sua forma pura, introduce un limite fondamentale: la **perdita dell'informazione posizionale**.

Il *self-attention* calcola la rilevanza tra ogni parola e tutte le altre senza tenere conto della loro posizione nella sequenza. Per il *self-attention*, la relazione tra la prima e la quinta parola è identica a quella tra la terza e la settima, purché siano separate dalla stessa "distanza" concettuale data dai *pesi di attenzione*. Di fatto, se si rimescolassero casualmente tutte le parole di una frase, ma si mantenessero le stesse relazioni semanticamente importanti, il blocco di *self-attention* produrrebbe essenzialmente lo stesso output (a meno di minime variazioni negli *score*).

Questo significa che per un modello basato puramente sulla *self-attention*, frasi come:

"Il cane insegue il gatto."
"Il gatto insegue il cane."

produrrebbero le stesse rappresentazioni interne per le parole "cane" e "gatto" in relazione l'una all'altra, poiché la loro interazione è vista come un insieme di relazioni, non come una sequenza ordinata. La nozione cruciale di ordine, e quindi la **distinzione tra soggetto e oggetto, viene persa**. Senza questa informazione, il modello non potrebbe distinguere chi sta compiendo l'azione e chi la subisce. È come avere un sacchetto di parole e voler ricostruire il significato di una frase, senza sapere quale parola viene prima e quale dopo. Per sopprimere a questo problema è stato introdotto il *positional encoding*.

2.2.2 Il Positional Encoding

Per reintrodurre l'informazione sull'ordine della sequenza, i *Transformer* utilizzano una componente fondamentale chiamata **Positional Encoding**. L'idea alla base è quella di aggiungere ai vettori che rappresentano gli elementi della sequenza (gli *embedding*) un'informazione numerica che codifichi la loro posizione relativa e assoluta all'interno della sequenza stessa. Questo permette al modello di distinguere tra sequenze con lo stesso contenuto ma con ordine diverso, come "Il cane insegue il gatto" e "Il gatto insegue il cane".

A differenza delle *RNN*, che intrinsecamente gestiscono l'ordine elaborando un elemento alla volta, e a differenza di un *embedding* standard che rappresenta solo il significato lessicale (o il valore) dell'elemento, il *Positional Encoding* fornisce un "**segnale di posizione**" unico e distintivo per ogni elemento all'interno della sequenza.

Questo segnale di posizione non è appreso direttamente dal modello attraverso l'addestramento, ma è **calcolato deterministicamente utilizzando funzioni sinusoidali** (*seno* e *coseno*) a diverse frequenze. La formula per il *Positional Encoding* per la posizione pos e la dimensione i del vettore è la seguente:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Dove:

- pos : Rappresenta l'indice della posizione dell'elemento nella sequenza (ad esempio, 0 per il primo elemento, 1 per il secondo, ecc.).
- i : Denota l'indice della dimensione all'interno del vettore di *positional encoding* (varia da 0 a $d_{model}/2 - 1$).
- d_{model} : Indica la dimensione del modello (corrisponde alla dimensione degli *embedding* di input).

Questa scelta delle funzioni sinusoidali non è arbitraria. Permette di generare un *encoding* unico per ogni posizione, scalabile a sequenze di lunghezza arbitraria (anche quelle non viste in fase di addestramento) e, crucialmente, consente al modello di apprendere facilmente la **posizione relativa** tra gli elementi. Ciò è possibile perché per ogni scostamento fisso k , il PE_{pos+k} può essere rappresentato come una funzione lineare di PE_{pos} , facilitando la comprensione delle relazioni spaziali all'interno della sequenza.

Esempio Pratico di Positional Encoding

Per illustrare il calcolo del *Positional Encoding*, consideriamo un esempio concreto con la sequenza di parole "I am a robot". La Figura 2.2 mostra la matrice di *positional encoding* generata per questa sequenza, assumendo una dimensione del modello (d_{model}) pari a 4.

Sequence	Index of token, k	Positional Encoding Matrix with $d=4$, $n=100$			
		$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0) = 0$	$P_{01}=\cos(0) = 1$	$P_{02}=\sin(0) = 0$	$P_{03}=\cos(0) = 1$
am	1	$P_{10}=\sin(1/1) = 0.84$	$P_{11}=\cos(1/1) = 0.54$	$P_{12}=\sin(1/10) = 0.10$	$P_{13}=\cos(1/10) = 1.0$
a	2	$P_{20}=\sin(2/1) = 0.91$	$P_{21}=\cos(2/1) = -0.42$	$P_{22}=\sin(2/10) = 0.20$	$P_{23}=\cos(2/10) = 0.98$
Robot	3	$P_{30}=\sin(3/1) = 0.14$	$P_{31}=\cos(3/1) = -0.99$	$P_{32}=\sin(3/10) = 0.30$	$P_{33}=\cos(3/10) = 0.96$

Positional Encoding Matrix for the sequence 'I am a robot'

Figura 2.2: Matrice del positional encoding per la sequenza 'I am a robot' con $d_{\text{model}} = 4$. Ogni riga rappresenta il vettore di encoding per la posizione corrispondente.

Analizziamo il processo illustrato in Figura 2.2 [19]:

Sequenza e Indice (pos)

- **Sequenza:** La frase di input è "I am a robot".
- **Indice del token, pos:** Rappresenta la posizione di ogni parola, a partire da 0. "I" si trova alla posizione pos=0, "am" alla pos=1, e così via.

La Matrice del positional encoding

Questo segmento rappresenta il **risultato del calcolo del Positional Encoding**. Ogni riga di questa matrice è il vettore di codifica posizionale finale per l'elemento corrispondente, e ogni colonna è una dimensione di tale vettore.

- **Parametri:** Come specificato nell'immagine, $d_{\text{model}} = 4$, indicando che il vettore di codifica posizionale per ogni posizione sarà a 4 dimensioni. L'indice i della formula varia da 0 a $d_{\text{model}}/2 - 1 = 1$.
- **Le Formule applicate:** I valori sono calcolati utilizzando le funzioni seno e coseno con le specifiche frequenze. Questo meccanismo genera **onde sinusoidali a velocità diverse**, assicurando che ogni combinazione pos, i produca un valore unico, contribuendo così a rendere unico il vettore finale per ogni posizione.

Il Risultato per ogni Posizione

Ad esempio, la posizione pos=1 (corrispondente alla parola "am") ottiene il vettore di codifica posizionale unico: [0.84, 0.54, 0.10, 1.0]. Questa specifica combinazione di valori, derivante da onde a frequenza diversa, crea un **segnale distintivo e informativo** per ogni posizione nella sequenza.

Rappresentazione Grafica del Positional Encoding

La natura ondulatoria del *Positional Encoding* può essere visualizzata graficamente. La Figura 2.3 presenta un grafico che illustra i valori delle funzioni seno e coseno per diverse dimensioni di un vettore di *positional encoding* al variare della posizione.

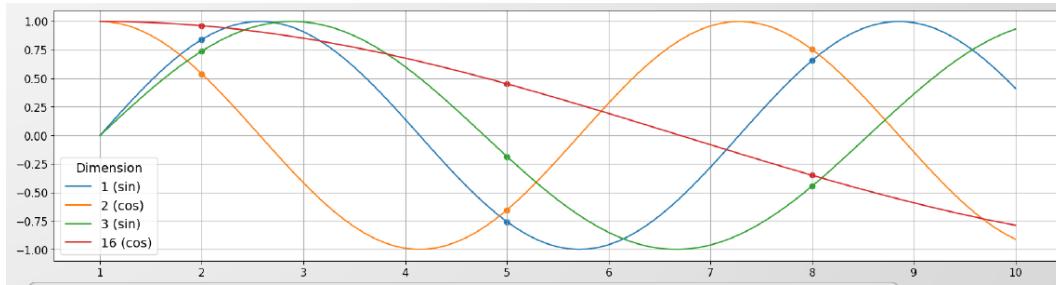


Figura 2.3: Grafico delle funzioni seno e coseno per diverse dimensioni del Positional Encoding, evidenziando l'unicità del vettore per ogni posizione.

Nella Figura 2.3 [7]:

Asse X (Position k) Rappresenta la posizione dell'elemento nella sequenza.

Asse Y (Valore) Indica il valore calcolato dalla funzione seno o coseno per una specifica dimensione del *Positional Encoding*.

Le Curve Colorate Ogni curva denota l'andamento dei valori per una specifica dimensione all'interno del vettore di *positional encoding*.

- Le curve **blu e arancione** (Dimensioni 1 e 2) utilizzano *sinusoidi a bassa frequenza* (onde con periodi più lunghi), fornendo una visione più "macro" della posizione.
- Le curve **verde e rossa** (Dimensioni 3 e 16) utilizzano *sinusoidi a frequenza più alta* (onde con periodi più corti), fornendo dettagli più "micro" sulla posizione.

Interpretazione dei Vettori di Posizione dal Grafico

I puntini colorati sul grafico sono fondamentali per la comprensione. Essi rappresentano i valori esatti che compongono le dimensioni di un vettore di *positional encoding* per una data posizione.

- **Ad esempio, osservando la linea verticale a Position (k) = 2:** I puntini che intersecano questa linea mostrano i valori che costituirebbero il vettore di *positional encoding* per il secondo elemento della sequenza. Il puntino sulla curva blu corrisponde al valore della Dimensione 1 per quella posizione, quello sulla curva arancione alla Dimensione 2, e così via.
- Analogamente, le linee verticali a Position (k) = 5 e Position (k) = 8 mostrano come i valori delle diverse dimensioni varino, creando *vettori di posizione unici* per ciascun punto della sequenza.

Questo approccio visivo conferma come la combinazione di sinusoidi a diverse frequenze generi un'impronta digitale unica per ogni posizione, consentendo al modello di distinguere l'ordine degli elementi.

Perché le funzioni sinusoidali?

Questa scelta non è casuale:

- **Unicità:** Ogni posizione pos ha un *Positional Encoding* unico.
- **Scalabilità:** Queste funzioni possono generare *encoding* per sequenze di lunghezza arbitraria, estendendo a lunghezze non viste in addestramento.
- **Relatività:** Un vantaggio fondamentale è che il *Positional Encoding* consente al modello di apprendere facilmente la posizione relativa tra gli elementi. Per ogni offset k costante, PE_{pos+k} può essere rappresentato come una funzione lineare di PE_{pos} . Questo è cruciale perché, ad esempio, per capire il ruolo di un aggettivo, è più importante sapere che è "prima" o "dopo" un nome, piuttosto che la sua posizione assoluta nella frase.

In pratica, ogni vettore di input, dopo essere stato trasformato nel suo *embedding* lessicale (o numerico nel caso delle serie temporali), viene semplicemente sommato al suo corrispondente vettore di *Positional Encoding*. Il risultato è un nuovo vettore che **incapsula sia il significato dell'elemento che la sua posizione** all'interno della sequenza. È questo vettore arricchito che viene poi passato ai blocchi di *self-attention*, permettendo loro di elaborare il contesto tenendo conto dell'ordine originale, e risolvendo così il dilemma dell'ambiguità posizionale.

2.2.3 Key, query e value

Per calcolare le interazioni tra gli elementi di una sequenza, il modello non usa direttamente i loro vettori di *embedding*, ma li proietta in tre spazi di rappresentazione distinti, generando tre nuovi vettori per ogni elemento: la *Query*, la *Key* e il *Value*.

Questi tre vettori sono il risultato di trasformazioni lineari (moltiplicazioni per matrici di pesi) apprese durante l'addestramento. Ogni vettore ha uno scopo specifico:

- **Query (Q) - La Domanda** La *Query* rappresenta l'elemento corrente che sta "cercando" informazioni per definire meglio il proprio contesto.
- **Key (K) - L'Etichetta** La *Key* agisce come un'etichetta per ogni elemento, descrivendo il tipo di informazione che può offrire. Viene confrontata con la *Query* per stabilire la compatibilità.
- **Value (V) - Il Contenuto** Il *Value* è il contenuto effettivo dell'elemento. Una volta stabilita la compatibilità tramite la *Key*, è il *Value* a essere trasmesso per arricchire la rappresentazione degli altri elementi.

Il Processo in Pratica: L'Esempio "La Palla va nel Canestro"

Immaginiamo di voler arricchire il significato della parola "palla" nella frase "La palla va nel canestro". Il modello calcolerà una nuova rappresentazione per "palla" basandosi sul suo contesto. Questo processo è illustrato nella Figura 2.4.

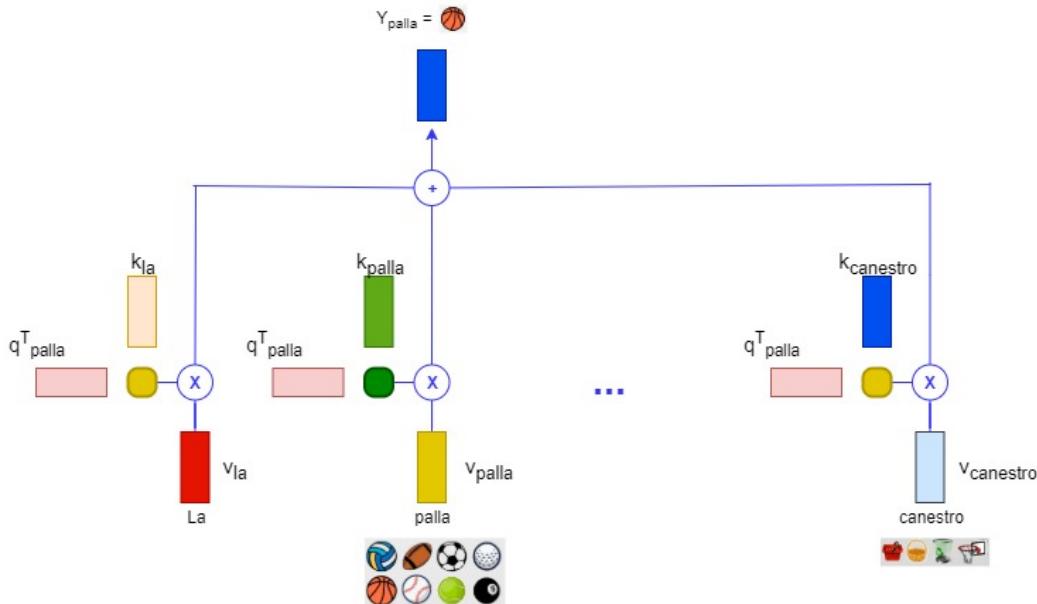


Figura 2.4: Calcolo della rappresentazione contestuale per la parola "palla".

La Query: La parola "palla" genera la sua *Query* (q^T_{palla}), che essenzialmente chiede: "Quali altre parole in questa frase sono rilevanti per me?".

Il Confronto con le Key: La *Query* della palla viene confrontata con la *Key* (k) di ogni parola della frase, inclusa se stessa. Questo confronto (un *prodotto scalare*) calcola un punteggio di compatibilità. Intuitivamente:

- $Query("palla")$ vs. $Key("La")$, $Key("va")$, $Key("nel")$: Punteggio basso. L'articolo non aggiunge molto contesto.
- $Query("palla")$ vs. $Key("palla")$: Punteggio alto. Una parola è sempre rilevante per se stessa.
- $Query("palla")$ vs. $Key("canestro")$: Punteggio alto. C'è una forte relazione semantica tra una palla e un cesto.

L'Aggregazione dei Value: I punteggi di compatibilità vengono normalizzati (tramite *softmax*) per diventare pesi di attenzione. Questi pesi determinano "quanta parte" del *Value* (v) di ogni parola verrà utilizzata per costruire la nuova rappresentazione di "palla". Il *Value* di "canestro" riceverà un peso significativo.

Il Risultato (Y_{palla}): La nuova rappresentazione contestuale della palla (Y_{palla}) sarà una media ponderata dei *Value* di tutte le parole. Il risultato non è più il significato generico di "palla", ma un significato arricchito dal contesto: una "palla-che-ha-a-che-fare-con-un-canestro", quindi una palla da basket.

Allo stesso modo, il modello esegue questo calcolo per ogni altra parola della frase. La Figura 2.5 mostra il processo dal punto di vista della parola "canestro".

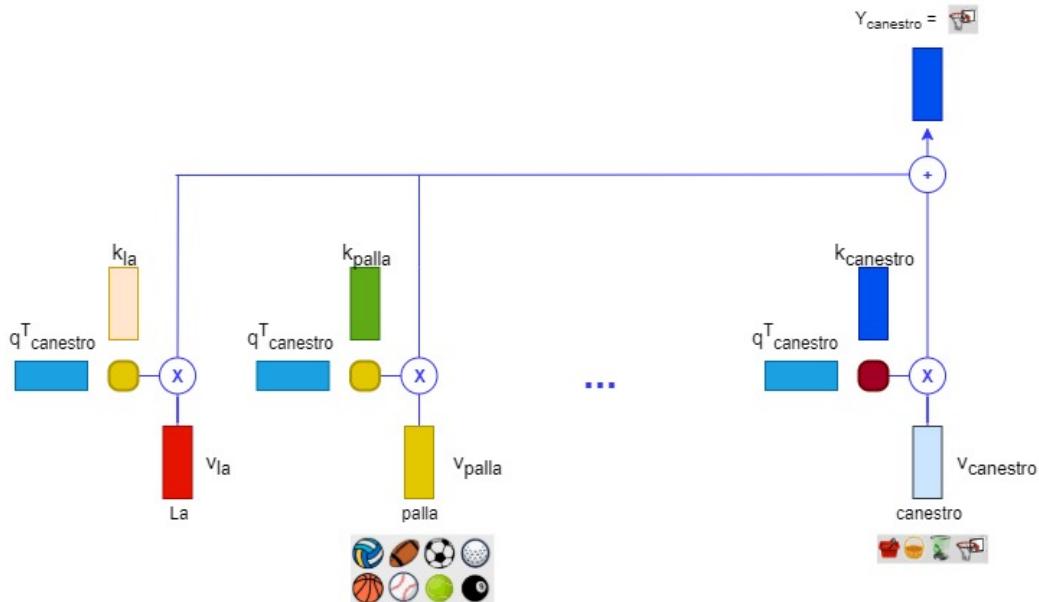


Figura 2.5: Calcolo della rappresentazione contestuale per la parola "canestro"

In questo caso, è la parola "canestro" a generare la $Query (q^T_{\text{canestro}})$. Questa $Query$ andrà a cercare le parole più rilevanti per definire il contesto del canestro. Anche in questo caso, la Key della parola "palla" (k_{palla}) risulterà molto compatibile, indicando una forte relazione. Di conseguenza, la nuova rappresentazione del canestro (Y_{canestro}) sarà fortemente influenzata dal *Value* della palla, diventando un "canestro-che-ha-a-che-fare-con-una-palla" e quindi il canestro da basket. Questo meccanismo, applicato in parallelo a ogni parola, permette al modello di costruire una comprensione profonda e interconnessa dell'intera sequenza.

2.2.4 Multi-head self-attention

Un meccanismo di self-attention standard, pur essendo potente, potrebbe essere limitato nel tipo di relazioni che riesce a modellare. Imparando un'unica serie di matrici per Query, Key e Value (Q, K, V), l'attenzione potrebbe specializzarsi nel catturare solo il tipo di dipendenza più dominante nei dati. Ad esempio, in una frase, potrebbe imparare a collegare sistematicamente il soggetto al suo verbo. Tuttavia, potrebbe faticare a cogliere simultaneamente altri tipi di relazioni, come quelle tra un aggettivo e il nome che modifica, o le relazioni sintattiche a più lungo raggio. È come chiedere a un critico d'arte di analizzare un quadro potendo concentrarsi su un solo aspetto alla volta, come il colore o la composizione, ma non su entrambi contemporaneamente.

Per superare questo limite, la Multi-Head Attention propone di eseguire il processo di self-attention non una, ma **h volte in parallelo**, con proiezioni lineari diverse. Ogni esecuzione parallela è chiamata "testa" (*head*), e ognuna di esse può specializzarsi nell'imparare un diverso tipo di relazione all'interno della sequenza.

Analisi dell'Architettura: Il Flusso dell'Informazione

La Figura 2.6 [7] illustra magnificamente l'architettura della Multi-Head Self-Attention. Analizziamo il flusso dei dati passo dopo passo, partendo dal basso, per comprendere come le diverse componenti interagiscono.

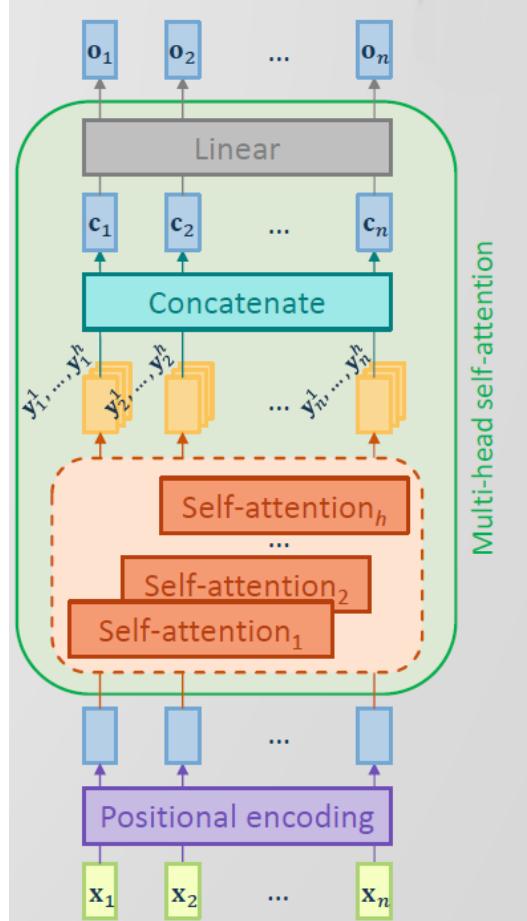


Figura 2.6: Architettura della Multi-Head Self-Attention, che mostra il flusso parallelo delle teste di attenzione, la concatenazione e la proiezione finale.

- 1. Input e Positional Encoding:** Alla base del diagramma (Figura 2.6) abbiamo la sequenza di input (x_1, x_2, \dots, x_n) , che rappresenta l'input. A questi viene sommato il **Positional Encoding**, che "inietta" l'informazione sull'ordine sequenziale, producendo i vettori di input finali che entrano nel blocco di attenzione.
- 2. Proiezione nelle Diverse "Teste":** Il passo cruciale avviene qui. Invece di calcolare un unico set di Query, Key e Value per l'intera sequenza, il modello apprende **h set distinti di matrici di proiezione** (W_i^Q, W_i^K, W_i^V per ogni testa i da 1 a h). I vettori di input vengono quindi proiettati in h sottospazi di rappresentazione diversi, uno per ogni testa. In pratica, ogni testa riceve una "versione" leggermente diversa e di dimensione ridotta della sequenza originale, su cui può concentrarsi.
- 3. Self-Attention in Parallello:** All'interno di ciascuna delle h teste (rappresentate dai blocchi arancioni "Self - attention₁", ..., "Self - attention_h"), viene applicato in parallelo il meccanismo di **Scaled Dot-Product Attention**. Ogni testa calcola i

propri punteggi di attenzione e produce un proprio vettore di output (y_1^i, \dots, y_n^i per la testa i). Poiché ogni testa opera su una diversa proiezione dell'input, imparerà a focalizzarsi su tipi di relazioni differenti. Ad esempio, in una frase, una testa potrebbe specializzarsi nel catturare le relazioni soggetto-verbo, un'altra le relazioni tra aggettivi e nomi. Nelle serie temporali, una testa potrebbe imparare a catturare la stagionalità settimanale, mentre un'altra potrebbe focalizzarsi sul trend a lungo termine.

4. **Concatenazione:** Una volta che tutte le h teste hanno completato i loro calcoli in parallelo, i loro vettori di output vengono **concatenati** insieme, come mostrato nel blocco 'Concatenate'. Questo crea, per ogni posizione nella sequenza, un unico vettore molto grande che contiene le informazioni provenienti da tutte le diverse "prospettive" apprese dalle singole teste.
5. **Proiezione Lineare Finale:** Il vettore concatenato, che ora contiene una rappresentazione molto ricca, viene infine passato attraverso un ultimo **strato lineare** (blocco 'Linear'). Questo strato, dotato di una propria matrice di pesi (W^O) appresa durante l'addestramento, ha due scopi: combinare in modo ottimale le informazioni provenienti dalle diverse teste e proiettare il vettore concatenato di nuovo nella dimensione originale del modello (d_{model}), producendo l'output finale del blocco (o_1, o_2, \dots, o_n).

2.3 L'architettura originale del transformer

Dopo aver sezionato le componenti fondamentali che costituiscono il cuore del Transformer possiamo ora assemblarle per comprendere l'architettura completa del modello. Nella sua forma originale, proposta nel celebre paper "Attention Is All You Need", il Transformer è un modello **Encoder-Decoder**.

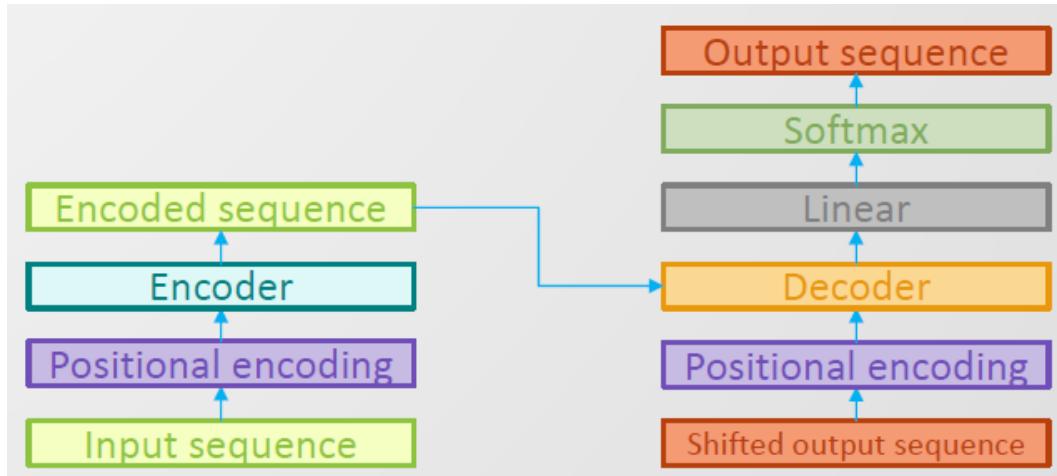


Figura 2.7: L'architettura completa del modello Transformer, che evidenzia i blocchi Encoder (a sinistra) e Decoder (a destra) [7]

L'architettura opera seguendo una filosofia intuitiva che segue il processo cognitivo umano, come quello di un traduttore. Il processo è suddiviso in due fasi distinte ma interconnesse:

1. **Fase di Codifica (Encoding):** L'**Encoder** ha il compito di "leggere" e "comprendere" l'intera sequenza di input. Riceve tutti gli elementi della sequenza simultaneamente e, attraverso una serie di strati di elaborazione, costruisce una rappresentazione numerica ricca di contesto per ogni singolo elemento.
2. **Fase di Decodifica (Decoding):** L'output dell'encoder è poi passato al **Decoder** che ha il compito di "scrivere" la sequenza di output. Opera in modo **autoregressivo**, ovvero genera un elemento alla volta, basandosi su due fonti di informazione cruciali: la rappresentazione completa della sequenza di input fornita dall'Encoder e gli elementi che ha già generato nei passi precedenti. L'output, ad esempio, diventa una frase in un'altra lingua.

Il Blocco Encoder

L'obiettivo primario dell'Encoder è trasformare una sequenza di embedding di input in una sequenza di rappresentazioni contestualizzate. Per raggiungere questo scopo, non si affida a un singolo strato di elaborazione, ma a una **pila di N strati identici** (nel paper originale, $N=6$). L'idea di impilare più strati è un concetto fondamentale nel deep learning: permette al modello di costruire **rappresentazioni gerarchiche**. Gli strati inferiori potrebbero catturare relazioni più semplici e locali (come le dipendenze sintattiche tra parole vicine), mentre gli strati superiori, elaborando le output degli strati inferiori, possono apprendere relazioni più complesse e astratte, arrivando a una comprensione semantica dell'intera frase.

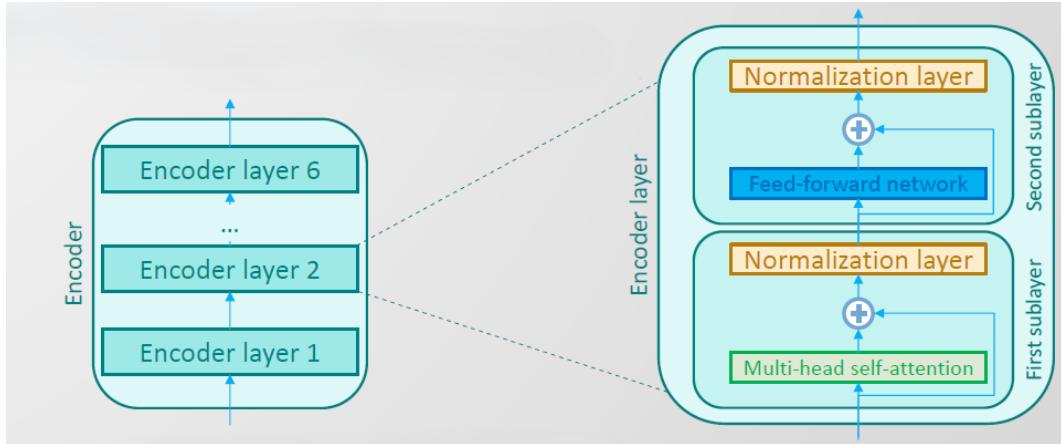


Figura 2.8: L'architettura dell'encoder del modello Transformer [7]

Ogni strato all'interno della pila dell'Encoder ha una struttura interna identica, composta da due sotto-strati principali.

1. **Multi-Head Self-Attention:** Il primo sotto-strato è un meccanismo di Multi-Head Self-Attention. Come analizzato in precedenza, questo è il motore che permette a ogni parola della sequenza di input di "guardare" e pesare l'importanza di tutte le altre parole nella stessa sequenza. Il risultato di questo processo è che il vettore di ogni parola viene aggiornato, passando da una rappresentazione isolata a una arricchita dal contesto globale della frase.
2. **Position-wise Feed-Forward Network (FFN):** Il secondo sotto-strato è una rete neurale feed-forward relativamente semplice, ma cruciale. È composta da due trasformazioni lineari con una funzione di attivazione non lineare (tipicamente ReLU, *Rectified Linear Unit*) nel mezzo. Questa rete viene applicata in modo identico e indipendente a ogni posizione della sequenza (da cui il nome *position-wise*). Sebbene la sua struttura sia semplice, il suo ruolo è fondamentale:
 - **Introduzione della Non-Linearità:** Lo strato di attention, di per sé, è principalmente lineare. La FFN introduce la non-linearità necessaria per permettere al modello di apprendere trasformazioni molto più complesse.
 - **Elaborazione per Posizione:** Agendo su ogni posizione separatamente, la FFN può essere vista come un piccolo Multi-Layer Perceptron che elabora e raffina ulteriormente la rappresentazione contestualizzata prodotta dallo strato di attention, portandola in uno spazio di rappresentazione più adatto per gli strati successivi.

Per garantire che un modello così profondo possa essere addestrato efficacemente senza incorrere in problemi come il *vanishing gradient*, ogni sotto-strato (sia l'attention che la FFN) è avvolto da due componenti architettoniche di vitale importanza:

- **Connessioni Residue (Add):** L'input di ogni sotto-strato viene sommato al suo output. Questa tecnica, nota come *residual connection*, crea una sorta di "scorciatoia" che permette al gradiente di fluire più facilmente all'indietro durante la backpropagation, prevenendo il problema del vanishing gradient.

- **Layer Normalization (Norm):** L'output della connessione residua viene immediatamente normalizzato. La *Layer Normalization* stabilizza l'addestramento ricalibrando media e varianza degli output di ogni strato, accelerando la convergenza del modello.

Il Blocco Decoder

Il Decoder ha il compito più complesso di generare la sequenza di output. Anch'esso è una **pila di N strati identici**, ma la sua struttura interna è leggermente più articolata di quella dell'Encoder, poiché deve gestire tre flussi di informazioni: la sequenza di input (tramite l'Encoder), la posizione corrente nella sequenza di output e gli elementi di output già generati.

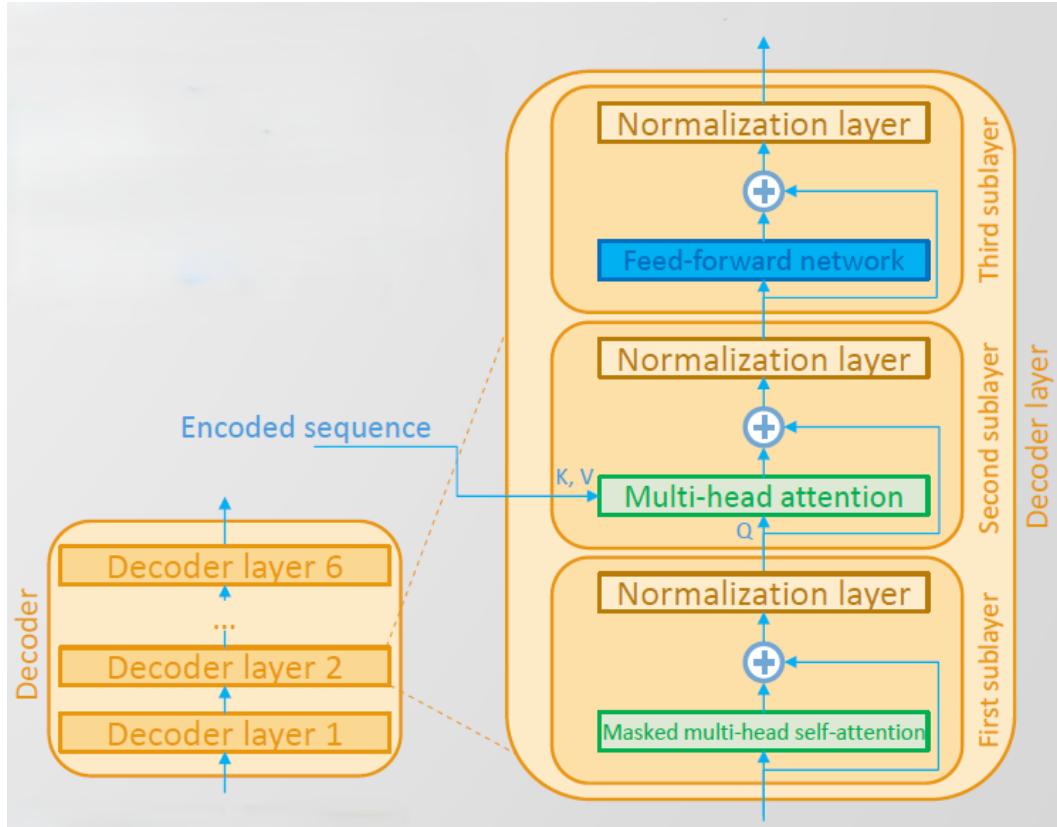


Figura 2.9: L'architettura del decoder del modello Transformer [7]

Ogni strato del Decoder contiene **tre sotto-strati principali**.

1. **Masked Multi-Head Self-Attention:** Il primo sotto-strato è un meccanismo di self-attention con una differenza cruciale: l'uso di una **maschera (mask)**. Poiché il Decoder opera in modo autoregressivo, durante la predizione della parola alla posizione i , non deve avere accesso alle parole nelle posizioni successive ($i + 1, i + 2, \dots$). La maschera assicura che il modello possa "guardare" solo al passato e al presente della sequenza che sta generando. Successivamente tale strato verrà approfondito.
2. **Encoder-Decoder Attention:** Questo è il cuore dell'interazione tra i due blocchi. Si tratta di un meccanismo di Multi-Head Attention in cui:
 - Le **Query (Q)** provengono dal sotto-strato precedente del Decoder.
 - Le **Key (K)** e i **Value (V)** provengono dall'**output finale dell'intera pila dell'Encoder**.

In questo modo, a ogni passo di decodifica, il Decoder utilizza il suo stato attuale (rappresentato dalle Query) per "consultare" la rappresentazione completa della sequenza di input (rappresentata dalle Key e dai Value) e decidere quali parti dell'input sono più rilevanti per generare la parola successiva.

3. Position-wise Feed-Forward Network (FFN): Il terzo sotto-strato è una rete feed-forward identica a quella presente nell'Encoder, che elabora ulteriormente le rappresentazioni ottenute.

Come nell'Encoder, anche qui ogni sotto-strato è avvolto da **connessioni residue** e **layer normalization** per garantire un addestramento stabile.

Dopo che la pila del Decoder ha elaborato l'informazione, l'output finale (il vettore corrispondente alla posizione corrente) viene passato attraverso un ultimo strato lineare la cui dimensione di output corrisponde alla grandezza del vocabolario, seguito da una funzione softmax per ottenere una distribuzione di probabilità. La parola con la probabilità più alta viene scelta come output per quel passo, e il processo si ripete fino a quando non viene generato un token speciale di "fine sequenza".

2.3.1 Masked self-attention

Durante la fase di addestramento, è fondamentale che l'output della *self-attention* in una posizione k venga calcolato prestando attenzione solo agli elementi della sequenza di input che precedono quella posizione. Il modello deve imparare a predire la parola successiva basandosi **unicamente sul contesto che ha già generato**. Se, durante l'addestramento, il modello potesse "sbirciare" le parole future, imparerebbe semplicemente a **copiare l'input**, senza sviluppare alcuna capacità predittiva.

Per imporre questo vincolo, il Decoder utilizza una variante del meccanismo di *attention* chiamata **Masked Self-Attention**.

Il principio di funzionamento è semplice ma efficace. Prima di applicare la funzione *softmax* per calcolare i pesi di attenzione, viene applicata una **"maschera"** ai punteggi di compatibilità. Questa maschera interviene per garantire che solo gli elementi di input che precedono la posizione di output corrente siano considerati nel calcolo dell'output per quella posizione. Pertanto, tutti i pesi di attenzione ω_{kj} dove $j > k$ vengono impostati a 0. Tecnicamente, questo si ottiene impostando i punteggi di attenzione non normalizzati ω'_{kj} a $-\infty$ per tutte le posizioni future. Quando la funzione *softmax* viene applicata a questi punteggi, i valori di $-\infty$ vengono trasformati in zero, annullando di fatto l'influenza di tutti i token futuri. In questo modo, la *Masked Self-Attention* assicura che il calcolo per l'output dell'elemento k -esimo consideri solo il contesto da 1 a $k - 1$, forzando il Decoder a operare in modo **autenticamente autoregressivo** e preparandolo per il suo compito di generazione durante la fase di previsione.

2.4 L'addestramento e la predizione

L'architettura del *Transformer*, pur essendo unica, opera in due modalità fondamentalmente diverse a seconda che si trovi in fase di **addestramento** (*training*) o di **previsione** (*inference*). Comprendere questa distinzione è cruciale per capire come il modello apprende e come, successivamente, genera nuove sequenze. La fase di addestramento è progettata per la massima efficienza e parallelizzazione, mentre la fase di previsione è un processo intrinsecamente **sequenziale e iterativo**.

La Fase di Addestramento

Durante l'addestramento, l'obiettivo è insegnare al modello a predire la parola successiva in una sequenza. Per fare ciò in modo efficiente, il *Transformer* utilizza una tecnica nota come *teacher forcing*. Invece di generare una parola e usare quella (potenzialmente errata) come input per il passo successivo, il modello riceve in input la sequenza di verità (*ground truth*) completa, ma "spostata" di una posizione.

Il vero punto di forza del *Transformer* in questa fase è la sua capacità di elaborare tutti gli elementi della sequenza **simultaneamente**. Grazie al meccanismo di *masked self-attention* nel **Decoder**, il modello può calcolare la perdita (l'errore di previsione) per ogni posizione della sequenza in un unico passaggio (*forward pass*), senza dover eseguire un ciclo iterativo.

Esempio Pratico di Addestramento

Prendiamo come esempio la famosa frase dantesca fornita nell'immagine.

1	Nel	Nel	Nel	Nel	Nel
2	mezzo	mezzo	mezzo	mezzo	mezzo
3	del	del	del	del	del
4	cammin	cammin	cammin	cammin	cammin
5	di	di	di	di	di
6	nostra	nostra	nostra	nostra	nostra
7	vita	vita	vita	vita	vita
8	mi	mi	mi	mi	mi
9	ritrovai	ritrovai	ritrovai	ritrovai	ritrovai
10	per	per	per	per	per
11	una	una	una	una	una
12	selva	selva	selva	selva	selva
13	oscura	oscura	oscura	oscura	oscura
14	ché	ché	ché	ché	ché
15	la	la	la	la	la

Figura 2.10: Rappresentazione di una sequenza di input durante la fase di addestramento. [7]

L'immagine illustra come il *Transformer* "vede" la sequenza durante l'addestramento. L'intera frase, da "Nel" a "la", è disponibile al modello in un unico istante. Il processo si svolge così:

- **Input all'Encoder:** L'**Encoder** riceve l'intera sequenza e ne calcola una rappresentazione contestuale completa.
- **Input e Obiettivo del Decoder:** Il **Decoder** è addestrato a predire ogni parola, avendo accesso solo a quelle che la precedono. Questo avviene in parallelo per tutte le posizioni:
 - Per predire la parola alla posizione 4, "cammin", il modello riceve come input "Nel mezzo del" e il suo obiettivo è produrre "cammin".
 - Per predire la parola alla posizione 11, "una", il modello riceve come input "Nel mezzo del cammin di nostra vita mi ritrovai per" e il suo obiettivo è produrre "una".
 - Per predire la parola alla posizione 14, "ché", il modello riceve "Nel mezzo ... selva oscura" e il suo obiettivo è "ché".

Grazie al mascheramento, tutte queste previsioni vengono calcolate **simultaneamente**, rendendo l'addestramento estremamente efficiente e scalabile.

La Fase di Previsione

La fase di **previsione** (o *inference*) è radicalmente diversa. Qui, il modello non conosce la sequenza futura e deve generarne una nuova. Il processo diventa **autoregressivo**: il modello genera una parola alla volta, e ogni parola generata viene aggiunta all'input per il passo successivo.

Il ciclo di generazione funziona in questo modo:

1. Il modello riceve una sequenza di partenza (il *prompt*).
2. L'**Encoder** elabora il *prompt*, e il **Decoder** genera una distribuzione di probabilità sulla parola successiva.
3. La parola più probabile viene scelta e aggiunta alla fine della sequenza di input.

4. La nuova sequenza, ora più lunga di una parola, viene data di nuovo in input al modello.
5. Il ciclo si ripete fino a quando il modello non genera un token speciale di "fine sequenza" o non raggiunge una lunghezza massima predefinita.

Esempio Pratico di Previsione

L'immagine successiva illustra perfettamente questo processo iterativo.

1	nostra	nostra	nostra	nostra
2	vita	vita	vita	vita
3	mi	mi	mi	mi
4	ritrovai	ritrovai	ritrovai	ritrovai
5	per	per	per	per
6	una	una	una	una
7	selva	selva	selva	selva
8	oscura	oscura	oscura	oscura
9	ché	ché	ché	ché
10	la	la	la	la
11	diritta	diritta	diritta	diritta
12		via	via	via
13			era	era
14				smarrita

Figura 2.11: Processo di generazione autoregressiva durante la fase di previsione. [7]

Immaginiamo che il modello abbia ricevuto come input la sequenza "... ché la".

Primo Passo Il modello processa l'input e predice la parola successiva più probabile: "diritta".

Secondo Passo La parola "diritta" viene aggiunta alla sequenza. Il nuovo input diventa "... ché la diritta". Il modello rielabora questa nuova sequenza e predice la parola successiva: "via".

Terzo e Quarto Passo Il processo si ripete. La parola "via" viene aggiunta, e il modello predice "era". Infine, "era" viene aggiunta, e il modello predice "smarrita".

Le frecce blu curve nell'immagine rappresentano visivamente questo ciclo **autoregressivo**: l'output di un passo diventa parte dell'input per il passo successivo. Questa natura sequenziale rende la previsione molto più lenta dell'addestramento, ma è ciò che permette al modello di generare testo coerente e contestualmente appropriato.

2.5 Adattamento dei transformers per le serie storiche

Trasportare un'architettura nata per il dominio discreto e simbolico del linguaggio naturale al mondo continuo e numerico delle serie temporali presenta alcune sfide. Sebbene il paradigma della *self-attention* sia potente e generalizzabile, sono necessari alcuni adattamenti specifici per permettere al Transformer di elaborare e prevedere efficacemente le serie storiche.

Sfide e Soluzioni Principali

- **Dati Continui vs. Discreti:** L'input di un Transformer per NLP è una sequenza di token discreti, ognuno mappato a un vettore tramite uno strato di *embedding* appreso da un vocabolario finito. Le serie temporali, invece, sono sequenze di valori scalari continui.

Soluzione: Si sostituisce lo strato di *embedding* testuale con uno strato di proiezione, tipicamente una rete neurale lineare (*Dense*) o, più efficacemente, una **convoluzione 1D**. Questo strato ha il compito di mappare il valore scalare (o un vettore di *feature* a quel *time step*) in uno spazio dimensionale più elevato (d_{model}), compatibile con il modello. In questo contesto, la normalizzazione dei dati di input (es. *Z-score*) diventa un passo di *pre-processing cruciale*.

- **Catturare Pattern Locali:** Il *self-attention* eccelle nel catturare dipendenze globali a lungo raggio, ma potrebbe faticare a cogliere pattern locali a corto raggio (es. piccole fluttuazioni o "motivi" ripetitivi) che sono invece ben gestiti da modelli come le *CNN*.

Soluzione: L'uso di strati **convoluzionali 1D** all'inizio dell'architettura (spesso come strato di *embedding*) è una modifica comune. Questo permette al modello di estrarre *feature* locali e di creare rappresentazioni più robuste prima di applicare il *self-attention* per modellare le dipendenze globali.

- **Complessità Quadratica per Serie Lunghe:** La complessità computazionale e di memoria del *self-attention* è quadratica rispetto alla lunghezza della sequenza ($O(n^2)$). Questo può diventare un ostacolo insormontabile quando si lavora con serie temporali molto lunghe, comuni in settori come la finanza o la sensoristica.

Soluzione: La ricerca ha prodotto diverse varianti di *attention* più efficienti (es. *Informer* [4], *Autoformer* [5], *FlashAttention*) che riducono questa complessità a quasi-lineare ($O(n \log n)$ o $O(n)$), rendendo i Transformer applicabili anche a problemi di previsione a lunghissimo termine.

Questi adattamenti dimostrano la **notevole flessibilità** del paradigma Transformer che, pur essendo nato in un contesto diverso, è generalizzabile anche per la complessa sfida della previsione di serie temporali.

Capitolo 3

Sviluppo e implementazione dei modelli

Dopo aver tracciato l’evoluzione teorica dei modelli di previsione, dalle fondamenta statistiche fino alle complesse architetture dei *Transformer*, questo capitolo tratterà il **cuore** della tesi. Verrà descritto in dettaglio il processo di sviluppo e implementazione dei modelli utilizzati per rispondere alla domanda di ricerca.

La spiegazione inizierà con l’analisi dei dati. Verrà descritto in dettaglio il dataset delle immatricolazioni automobilistiche europee, seguito da un’analisi approfondita delle fasi di *pre-processing* e *feature engineering*. Questi passaggi sono cruciali per trasformare i dati grezzi in un formato informativo e ottimale per l’addestramento dei modelli di *deep learning*.

Successivamente, ci concentreremo sull’implementazione delle architetture a confronto. Verrà presentata la struttura del **modello di baseline**, un’architettura **LSTM** consolidata, che servirà come metro di paragone per le tecniche più innovative. Seguirà la descrizione dettagliata dei due modelli basati su **architettura Transformer** sviluppati per questa tesi: un modello **Encoder-Decoder** completo e una sua variante **Decoder-Only**, progettata per operare in modo puramente autoregressivo.

Infine, per contestualizzare i risultati non solo rispetto a una *baseline* classica ma anche rispetto allo stato dell’arte attuale, verrà introdotto un quarto modello: **Chronos**. Si tratta di un modello *foundational* pre-allenato su vasta scala, che rappresenta un **benchmark esterno di altissimo livello** [20]. Verrà spiegato il suo funzionamento e come è stato integrato.

Attraverso questi passaggi, questo capitolo getterà le fondamenta tecniche per l'**analisi comparativa dei risultati** che verrà presentata nel capitolo successivo, illustrando ogni scelta progettuale e implementativa in modo **trasparente e rigoroso**.

3.1 Il Dataset

La validazione e la comparazione di modelli previsionali richiedono un dataset che sia non solo di **alta qualità** e **sufficientemente esteso**, ma che presenti anche caratteristiche complesse e rappresentative di fenomeni reali. Un dataset con *trend*, **stagionalità marcata** e **rotture strutturali** costituisce un banco di prova ideale per valutare la robustezza e la capacità di generalizzazione di architetture avanzate come i *Transformer*.

3.1.1 Origine e Contesto del Dataset

Il dataset utilizzato in questo lavoro di tesi è stato gentilmente fornito dal **Dr. Livio Fenga**, Senior Lecturer in Business Analytics, grazie alla mediazione del **Professor Vittorio Maniezzo**. I dati raccolti rappresentano le **immatricolazioni mensili di nuove auto-vetture in Europa**, comprensive di tutte le tipologie di propulsione (motori a combustione interna, ibridi, elettrici, ecc.). Queste caratteristiche rendono il dataset un **indicatore macroeconomico completo e fedele**.

L’ampia copertura geografica è uno dei punti di forza di questo dataset. Esso include le serie storiche di **22 nazioni europee**, offrendo una visione d’insieme eterogenea del mercato. Le nazioni comprese sono: Austria, Belgio, Danimarca, Finlandia, Francia, Germania, Grecia, Irlanda, Italia, Lussemburgo, Paesi Bassi, Spagna, Svezia, Regno Unito, Norvegia, Svizzera, Lettonia, Polonia, Ungheria, Repubblica Ceca, Estonia e Bulgaria.

Il periodo temporale coperto dai dati è notevolmente esteso, andando dal **gennaio 1990 al dicembre 2018**, portando a una dataset composto da 349 valori. Con quasi tre decenni di osservazioni mensili, il dataset offre una base solida per l'addestramento dei modelli, permettendo loro di apprendere pattern a lungo termine e di essere validati su un orizzonte temporale significativo.

3.1.2 Struttura e Analisi Esplorativa

Il dataset è strutturato in un formato tabellare standard, dove ogni riga rappresenta un'osservazione unica per una specifica nazione e data. Le colonne principali includono la data dell'osservazione (con frequenza mensile), il nome della nazione e il numero totale di immatricolazioni per quel mese. Un'analisi preliminare ha confermato la buona qualità dei dati, con una continuità temporale robusta e poche lacune, che sono state gestite in fase di *pre-processing*.

Prima di procedere con la modellazione, è stata condotta un'analisi esplorativa per comprendere le caratteristiche intrinseche delle serie storiche. Sebbene ogni nazione presenti peculiarità uniche, emergono pattern comuni e di grande interesse per la previsione. Per illustrare queste dinamiche, viene presentata come esempio rappresentativo la serie storica relativa all'Italia.

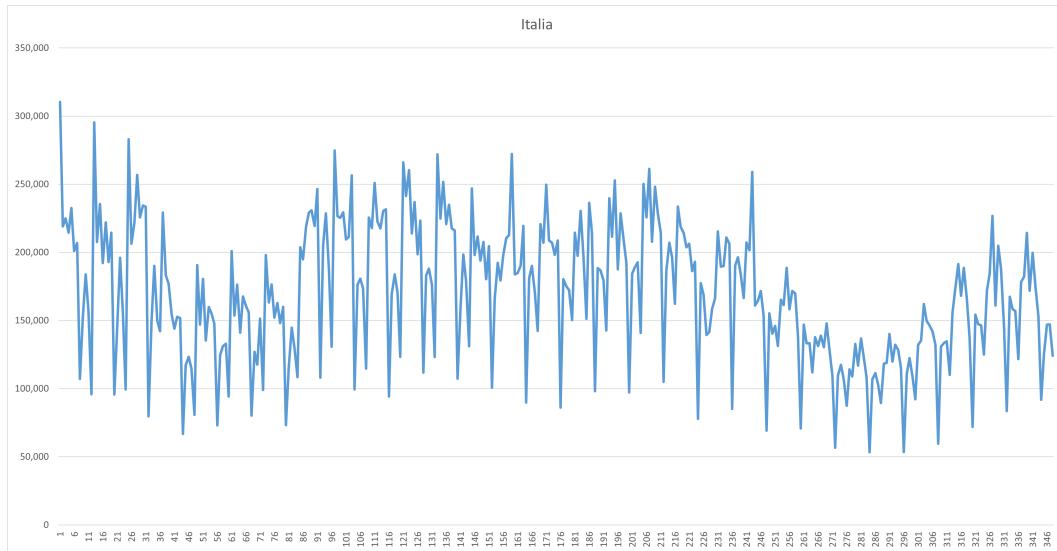


Figura 3.1: Andamento delle immatricolazioni mensili di autovetture in Italia (Gennaio 1990 - Dicembre 2018)

L'analisi del grafico 3.1 permette di identificare chiaramente le tre componenti classiche di una serie storica:

- **Stagionalità:** È visibile un pattern annuale molto **marcato e ricorrente**. Si notano picchi di immatricolazioni tipicamente nei primi mesi dell'anno e prima del periodo estivo, spesso legati a incentivi statali, al lancio di nuovi modelli e a scadenze fiscali. Al contrario, si osserva un **calo sistematico durante il mese di agosto**, a causa della chiusura estiva delle concessionarie e delle fabbriche.
- **Trend:** Al di là delle fluttuazioni stagionali, è possibile osservare *trend* di lungo periodo. Si nota un periodo di crescita e stabilità negli anni '90 (primi 120 valori) e primi 200 (dal valore in posizione 120 fino al 180), seguito da una drastica inversione di tendenza.
- **Rottura Strutturale (Crisi del 2007-2008):** L'elemento più significativo del grafico è la **drastica e prolungata discesa** delle immatricolazioni a partire dal 2008. Questo crollo non è una fluttuazione casuale, ma una **rottura strutturale causata dalla crisi finanziaria globale**. La crisi ha avuto un impatto devastante sulla fiducia dei consumatori e sulla loro capacità di spesa per beni durevoli come le automobili. Il

mercato automobilistico ha subito una delle contrazioni più gravi della sua storia, con un lento e faticoso recupero solo negli anni successivi.

È importante sottolineare che questo pattern, in discesa, non è un'anomalia del solo mercato italiano. La quasi totalità delle serie storiche presenti nel dataset mostra un calo simile e coevo, seppur con magnitudo diverse a seconda della resilienza economica di ciascuna nazione. La presenza di questo evento sistematico rende il dataset **particolarmente sfidante e realistico**. Pertanto mi ha permesso di scrivere modelli di previsione performanti in grado non solo di modellare la stagionalità regolare, ma anche di adattarsi a cambiamenti di regime così drastici. Un ottimo banco di prova per confrontare la capacità delle architetture tradizionali, come le *LSTM*, e dei modelli più moderni, come i *Transformer*, di gestire la complessità del mondo reale.

3.2 Preprocessing e Feature Engineering

3.2.1 Preprocessing

La fase di *preprocessing* è un **passaggio fondamentale** in qualsiasi progetto di *machine learning* e assume un'importanza ancora maggiore nel contesto delle serie temporali. L'obiettivo di questa fase è duplice: in primo luogo, **pulire e preparare i dati grezzi** per garantire la qualità e la coerenza dell'input; in secondo luogo, effettuare una **selezione preliminare delle feature** per costruire un dataset robusto e informativo, eliminando le fonti di rumore e le inconsistenze che potrebbero compromettere le performance del modello. Le operazioni descritte di seguito sono state applicate per trasformare il dataset grezzo in un formato ottimale per le successive fasi di *feature engineering* e modellazione.

3.2.1.1 Selezione Iniziale e Gestione dei Dati Mancanti

Il primo passo è stato quello di strutturare i dati in una matrice in cui ogni colonna rappresenta la serie storica delle immatricolazioni di una singola nazione e ogni riga corrisponde a un'osservazione mensile. In questa fase, le colonne contenenti la data e i nomi delle nazioni sono state momentaneamente escluse per concentrarsi esclusivamente sui soli valori numerici. Un'analisi preliminare ha rivelato la presenza di un numero significativo di **valori mancanti** all'inizio del periodo temporale per alcuni paesi. Nello specifico:

- **Islanda:** I primi 61 valori (corrispondenti a oltre 5 anni di dati) erano mancanti.
- **Altri Paesi dell'Est Europa** (Lettonia, Polonia, Ungheria, Repubblica Ceca, Estonia, Bulgaria): Presentavano circa 150 valori mancanti, il che significa che le loro serie storiche iniziavano solo intorno al 2002.

In un'analisi multivariata, la gestione standard dei dati mancanti tramite l'eliminazione delle righe contenenti valori `NaN` avrebbe comportato un **grave svantaggio**. Se avessi mantenuto questi paesi nel dataset, sarei stato costretto a troncare tutte le serie storiche all'anno 2002, **perdendo così oltre un decennio di dati storici preziosi** per i paesi con serie più complete. Per preservare la massima lunghezza possibile della serie storica e garantire un addestramento su un orizzonte temporale più ampio, si è deciso di **escludere completamente queste nazioni** dal dataset finale. Questa scelta pragmatica ha permesso di mantenere un periodo di analisi esteso dal 1990 al 2018 per il set di nazioni rimanenti.

3.2.1.2 Analisi di Correlazione per la Selezione delle Feature

Una volta ottenuto un insieme di serie storiche pulite e allineate, il passo successivo è stato quello di selezionare un sottoinsieme di nazioni da utilizzare come *feature* per la previsione. In un contesto multivariato, l'obiettivo è spesso quello di prevedere una serie **"target"** (ad esempio, le nuove immatricolazioni in Italia) utilizzando come informazione ausiliaria l'andamento di altre serie correlate. Per identificare le serie storiche più informative, è stata condotta un'**analisi di correlazione**.

La metodologia applicata ha seguito questi passaggi, come riassunto nel codice sottostante:

- **Trasformazione Logaritmica:** Per **stabilizzare la varianza** e ridurre l'impatto di *trend* esponenziali, ai dati è stata applicata una trasformazione logaritmica (`np.log1p`). Questo rende la correlazione di Pearson, che misura le relazioni lineari, più robusta e significativa.
- **Calcolo della Matrice di Correlazione:** È stata calcolata la **matrice di correlazione di Pearson** tra tutte le serie storiche. Questa matrice, visualizzata nella 3.2, quantifica la relazione lineare tra ogni coppia di paesi con un valore compreso tra -1 e +1. Più un valore sarà vicino a 1 maggiore sarà la correlazione con il paese.
- **Selezione delle Nazioni più Correlate:** È stata scelta una nazione ”*target*” e sono state selezionate le 7 nazioni le cui serie storiche presentavano la correlazione assoluta più alta con essa.

```

1 # Caricamento e trasformazione dei dati
2 data_df = pd.read_csv("dati_selezionati.csv")
3 data_log = np.log1p(data_df)
4
5 # Calcolo della matrice di correlazione
6 correlation_matrix = data_log.corr()
7
8 # Selezione delle N colonne Maggiormente correlate alla colonna target
9 target_col_index = 8 # Italy
10 top_n = 7
11 target_col_name = data_log.columns[target_col_index]
12 most_correlated_cols = correlation_matrix[target_col_name].abs().sort_values(ascending=False).index[1:top_n+1]
```

Listato 3.1: Selezione delle feature basata sulla correlazione.

Questo processo ha permesso di **ridurre la dimensionalità del problema**, mantenendo solo le serie storiche più rilevanti e interconnesse, ed eliminando quelle che avrebbero potuto introdurre rumore.

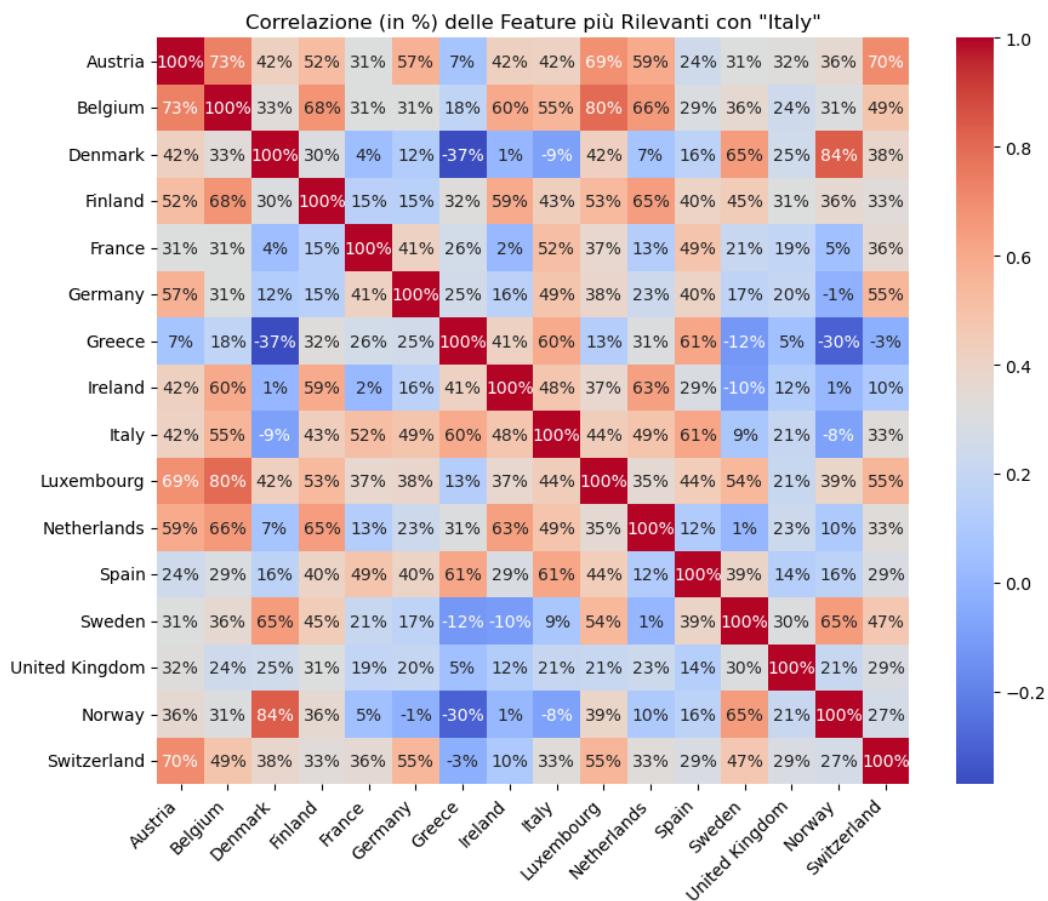


Figura 3.2: Matrice di correlazione (heatmap) delle immatricolazioni mensili tra le nazioni europee selezionate

La Figura 3.2 mostra una *heatmap* della matrice di correlazione, dove sia l'asse X che l'asse Y rappresentano le diverse nazioni europee incluse nell'analisi. I colori caldi (tendenti al rosso/arancione) indicano, come previsto, una **forte correlazione positiva** tra i mercati automobilistici, confermando che le economie sono interconnesse e rispondono in modo simile a shock macroeconomici, come la crisi del 2008.

In questo studio, è stata selezionata l'**Italia come serie storica target** per la previsione. Di conseguenza, l'analisi si è concentrata sull'identificare quali nazioni avessero un andamento più simile al mercato italiano. Le colonne corrispondenti agli indici [11, 6, 1, 4, 10, 5, 7] sono state identificate come quelle con la **correlazione più forte rispetto all'Italia**. La **selezione finale di questi paesi** ha permesso di creare il **dataset pulito e raffinato** su cui verranno eseguite le operazioni di *feature engineering* e l'addestramento dei modelli.

3.2.2 Feature Engineering

Una volta ottenuto un dataset pulito e coerente, il passo successivo è il *feature engineering*. Questo processo è stato uno dei passi fondamentali per migliorare le prestazioni dei modelli, in quanto consiste nel trasformare i dati grezzi in un insieme di *feature* informative che rendano i *pattern* sottostanti (come *trend* e *stagionalità*) più esplicativi e facilmente apprendibili dagli algoritmi.

Sono state create diverse categorie di *feature* per catturare le diverse dinamiche presenti nelle serie storiche delle immatricolazioni. Di seguito, verranno illustrate tutte le modalità utilizzate.

È importante sottolineare, tuttavia, che non tutte le *feature* create sono state impiegate per ogni architettura. Si è infatti riscontrato che la performance di ciascun modello è sensibile a un diverso sottoinsieme di input; pertanto, la selezione finale delle feature è stata personalizzata per ottimizzare ogni specifica architettura, poiché si è notato che alcuni modelli beneficiano maggiormente di determinate *feature* rispetto ad altri.

3.2.2.1 Feature Basate sui Lag (Lag Features)

La *feature* più fondamentale per la previsione di serie storiche è il valore della serie stessa in un istante temporale precedente. Queste *feature*, chiamate *lag*, si basano sul principio dell'autocorrelazione, secondo cui il valore attuale di una serie è spesso dipendente dai suoi valori passati.

Scopo: Fornire al modello una **memoria a breve e lungo termine** degli andamenti passati.

Implementazione: Utilizzando la funzione `.shift(n)`, sono stati creati diversi *lag* per la serie storica *target*:

- **target_lag_1**: Il valore del mese precedente. Cattura la dipendenza a brevissimo termine e l'inerzia del mercato.
- **target_lag_12**: Il valore dello stesso mese dell'anno precedente. Questa è una *feature cruciale per modellare la stagionalità annuale*.
- **target_lag_24**: Il valore dello stesso mese di due anni prima. Può aiutare a catturare cicli economici o di mercato a più lungo termine.

3.2.2.2 Feature Basate su Finestre Mobili (Rolling Window Features)

Queste *feature* calcolano statistiche aggregate su una finestra temporale mobile, fornendo una visione più "liscia" e stabile delle tendenze locali.

Scopo: Estrarre il *trend* locale e misurare la volatilità.

Implementazione:

- **Medie Mobili (rolling_mean_12)**: Calcolate con la funzione `.rolling(window=12).mean()`, rappresentano la media delle immatricolazioni negli ultimi 12 mesi. Questa *feature* smussa la stagionalità e il rumore, fornendo al modello una **chiara indicazione del trend annuale locale**.

- **Deviazioni Standard Mobili** (`rolling_std_3`, `rolling_std_6`, `rolling_std_12`): Calcolate con `.rolling(window=12).std()`, misurano la volatilità del mercato in finestre di 3, 6 e 12 mesi. Un valore alto indica un **periodo di forte incertezza o fluttuazione** (come durante la crisi del 2008), mentre un valore basso indica stabilità.

3.2.2.3 Feature Temporali (Time-based Features)

Per rendere il modello esplicitamente consapevole del contesto temporale, sono state create *feature* che codificano la data.

Scopo: Permettere al modello di apprendere *pattern* legati al calendario.

Implementazione:

- **Anno (year):** Una *feature* numerica che rappresenta l'anno. Aiuta il modello a catturare *trend* a lunghissimo termine o cambiamenti di regime che si sviluppano su più anni.
- **Codifica Ciclica del Mese** (`month_sin`, `month_cos`): Invece di usare il numero del mese (da 1 a 12), che creerebbe una falsa relazione ordinale (dicembre "lontano" da gennaio), i mesi sono stati codificati in uno spazio bidimensionale usando le funzioni seno e coseno. Questo approccio **preserva la natura ciclica della stagionalità**, rappresentando correttamente il fatto che dicembre è adiacente a gennaio.

3.2.2.4 Feature Avanzate

Per catturare dinamiche più sottili, sono state ingegnerizzate delle *feature* aggiuntive.

Scopo: Fornire segnali più reattivi e informazioni sui cambiamenti istantanei.

Implementazione:

- **Media Mobile Esponenziale** (`ewma_12`): Calcolata con `.ewm(span=12).mean()`, la EWMA è simile a una media mobile ma assegna pesi esponenzialmente decrescenti alle osservazioni passate. Questo la rende **più reattiva ai cambiamenti recenti** rispetto a una media mobile semplice.
- **Differenziazione** (`diff_1`, `seasonal_diff_12`): Invece dei valori assoluti, queste *feature* calcolano la variazione da un periodo all'altro.
 - `diff(1)`: La variazione mese su mese. Aiuta il modello a imparare il *momentum* a breve termine.
 - `diff(12)`: La variazione anno su anno. È una *feature* **molto potente** per catturare la crescita o la decrescita stagionale.

3.2.2.5 Gestione dei Valori Mancanti Finali

Tutte le operazioni che utilizzano dati passati (*lag*, finestre mobili, differenze) generano inevitabilmente valori `NaN` all'inizio del dataset. Ad esempio, per calcolare il `target_lag_24`, è necessario attendere che siano disponibili 24 osservazioni. Per garantire che il modello venisse addestrato solo su dati completi e validi, tutte le righe contenenti valori `NaN` sono state rimosse utilizzando `.dropna()`. Questo ha leggermente ridotto la lunghezza totale della serie storica, ma ha **garantito l'integrità e l'affidabilità dell'input** per i modelli.

3.2.3 Normalizzazione dei dati

Dopo il *feature engineering*, l'ultimo passaggio preparatorio è la *normalizzazione* dei dati. Le reti neurali come le *LSTM* e i *Transformer* **addestrano in modo più stabile ed efficiente** quando tutte le *feature* di *input* si trovano su una scala simile. Questo processo **previene problemi di convergenza e instabilità dei gradienti** causati da *feature* con ordini di grandezza molto diversi.

In questo studio sono state valutate due tecniche principali:

- **StandardScaler (Standardizzazione):** Questo metodo trasforma ogni *feature* in modo che abbia una **media di 0 e una deviazione standard di 1**. È un approccio robusto, molto comune nel *machine learning*, che centra i dati senza comprimerli in un *range* limitato.

- **MinMaxScaler (Scalatura Min-Max):** Questa tecnica, invece, riscalà ogni *feature* in un intervallo fisso, tipicamente tra 0 e 1. Sebbene utile, è **più sensibile agli outlier**, poiché un singolo valore estremo può influenzare la scala di tutti gli altri punti.

Poiché ciascuna serie storica presentava una distribuzione statistica distinta, con media e deviazione standard differenti, è stato addestrato uno **scaler separato per ogni colonna** del dataset. Questi *scaler* sono stati salvati per poter **applicare la trasformazione inversa** sulle previsioni del modello, riportandole così alla loro scala originale e rendendole interpretabili.

3.3 Architetture

Dopo aver preparato e ingegnerizzato il dataset, il passo successivo è la definizione e l'implementazione delle architetture di rete neurale che verranno confrontate in questo studio. Nelle pagine successive vedremo la spiegazione delle architetture utilizzate partendo dai modelli ricorrenti più tradizionali alle più recenti architetture basate su *Transformer*.

3.3.1 LSTM

Per stabilire un **solido punto di riferimento (baseline)**, è stata implementata un'architettura basata sulle *Long Short-Term Memory (LSTM)*. Le *LSTM*, come discusso in precedenza, sono una variante delle *Reti Neurali Ricorrenti (RNN)* che **eccellono nel modellare dipendenze a lungo termine** nelle sequenze, grazie alla loro complessa struttura interna con *gating mechanisms*. La loro comprovata efficacia in compiti di serie temporali le rende una **scelta naturale come modello di confronto**.

L'architettura *LSTM* implementata è una rete sequenziale composta da due strati *LSTM* seguiti da strati di *Dropout* e un *output layer Dense*. La struttura è visualizzata nella Figura 3.3 e il codice di implementazione è riportato di seguito.

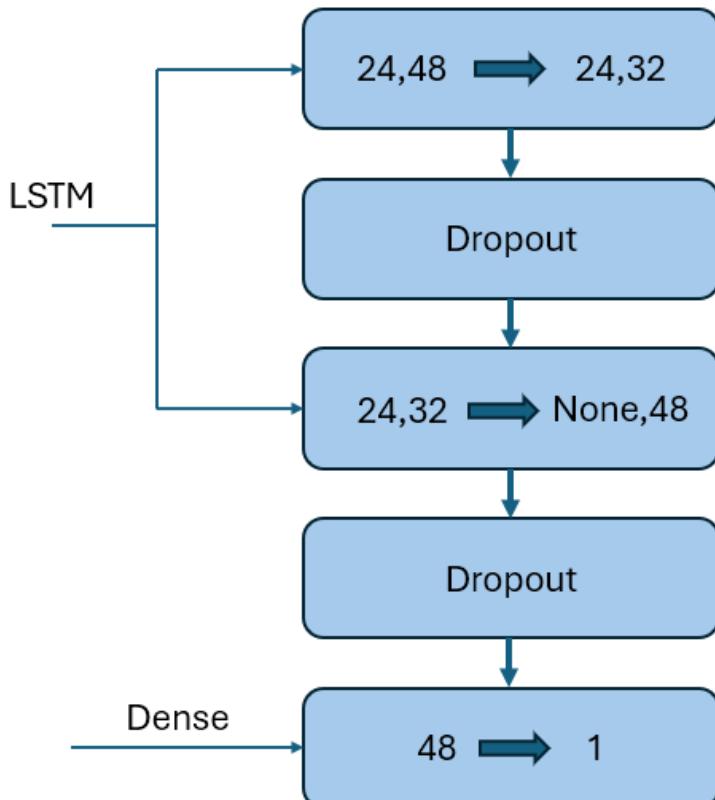


Figura 3.3: Architettura del modello LSTM di *baseline*. Mostra la concatenazione di due strati LSTM, ciascuno seguito da Dropout, e un layer Dense finale.

Primo strato

```
LSTM(32, return_sequences=True, kernel_regularizer,
      linput_shape=(sequence_length, num_features))
```

Questo è il primo strato LSTM della rete. Contiene 32 unità *LSTM* (neuroni). Il parametro `return_sequences=True` è **cruciale**: indica che lo strato deve restituire l'output per ogni *time step* della sequenza di input, piuttosto che solo l'output finale. Questo è necessario perché il secondo strato LSTM richiede input sequenziali. L'`input.shape ((sequence_length, num_features))` definisce la forma degli input che questo strato si aspetta; la `sequence_length` (lunghezza della finestra temporale di input) e `num_features` (numero di *feature* per ogni *time step*) verranno specificate in dettaglio nella sezione 3.4.1.

```
kernel_regularizer=l2(0.00082 ...)
```

Viene applicata una regolarizzazione L2 ai pesi del *kernel* dello strato. Questo aiuta a **prevenire l'overfitting** penalizzando i pesi grandi, promuovendo modelli più semplici e robusti.

```
Dropout(0.4)
```

Dopo il primo strato LSTM, è stato aggiunto uno strato di *Dropout* con un *rate* del 40%. Il *Dropout* è una tecnica di regolarizzazione che, durante l'addestramento, disattiva casualmente una percentuale di neuroni (in questo caso il 40%) in ogni passo. Questo **forza la rete a non dipendere eccessivamente da singoli neuroni** e a imparare rappresentazioni più ridondanti e robuste.

Secondo strato

```
LSTM(48,return_sequences=False, kernel_regularizer=l2(8.984581938078827e-05))
Dropout(0.2)
```

Questo è il secondo e ultimo strato LSTM, con 48 unità. Il parametro `return_sequences=False` (o omesso, poiché è il valore predefinito) indica che questo strato restituirà solo l'output dello stato nascosto finale, che **riassume le informazioni dell'intera sequenza**. Poi un ulteriore strato di *Dropout*, con un *rate* del 20%, viene applicato dopo il secondo strato LSTM per fornire un'ulteriore regolarizzazione.

Output e Compilazione del modello

```
Dense(1)
```

Questo è lo strato di output. È un semplice strato *Dense* (completamente connesso) con un'unica unità, poiché l'obiettivo è **prevedere un singolo valore numerico** (il numero di immatricolazioni) per il *time step* successivo.

```
optimizer = Adam(learning_rate=learning_rate)
model.compile(optimizer=optimizer, loss='mse')
```

Il modello viene poi compilato utilizzando l'ottimizzatore *Adam* [21], una **scelta popolare ed efficiente** per l'addestramento delle reti neurali, con un *learning rate* configurabile. Successivamente segue la funzione di costo (*loss function*). La scelta è ricaduta sul *Mean Squared Error (MSE)*. Il *MSE* è una metrica comune per i problemi di *regressione*, che **penalizza maggiormente gli errori più grandi**, promuovendo previsioni accurate. Di seguito è presente il codice utilizzato per creare il modello.

```

1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import LSTM, Dropout, Dense
3  from tensorflow.keras.regularizers import l2
4  from tensorflow.keras.optimizers import Adam
5
6  def create_lstm_model(sequence_length, num_features, learning_rate):
7      model = Sequential()
8
9      # Primo strato LSTM
10     model.add(LSTM(32, return_sequences=True,
11                     kernel_regularizer=l2(0.0008241284794813872),
12                     input_shape=(sequence_length, num_features)))
13     model.add(Dropout(0.4))
14
15     # Secondo strato LSTM
16     model.add(LSTM(48, return_sequences=False,
17                     kernel_regularizer=l2(8.984581938078827e-05)))
18     model.add(Dropout(0.2))
19
20     # Strato di output
21     model.add(Dense(1))
22
23     # Compilazione del modello
24     optimizer = Adam(learning_rate=learning_rate)
25     model.compile(optimizer=optimizer, loss='mse')
26
    return model

```

Listato 3.2: Implementazione del modello LSTM con TensorFlow/Keras.

3.3.2 Encoder-Decoder

Il secondo modello implementato in questo studio si basa sull’architettura **Transformer Encoder-Decoder**, la configurazione originale proposta da Vaswani et al. nel paper ”Attention Is All You Need”. Questa architettura è idealmente progettata per problemi di *sequence-to-sequence*, dove una sequenza di input (*source*) viene trasformata in una sequenza di output (*target*). Nel contesto della previsione di serie storiche multivariate, l’Encoder ha il compito di elaborare la sequenza storica di input, catturando dipendenze e pattern, mentre il Decoder genera la previsione del valore futuro della serie target, attingendo al contesto fornito dall’Encoder. L’implementazione è stata realizzata utilizzando il framework **PyTorch**, sfruttando i suoi moduli ad alto livello per costruire una struttura Transformer flessibile ed efficiente.

L’implementazione si articola in due classi principali: una per il **Positional Encoding** e una per l’architettura completa del **Transformer Encoder-Decoder**.

Implementazione del Positional Encoding

Come evidenziato nel capitolo teorico, il meccanismo di *self-attention* di per sé non codifica l’ordine degli elementi nella sequenza. Per fornire questa informazione temporale, è essenziale integrare il **Positional Encoding**. La classe `PositionalEncoding`, che eredita da `nn.Module` di PyTorch, è responsabile del calcolo e dell’aggiunta di questi vettori di posizione all’input.

```

1 import math
2 import torch
3 import torch.nn as nn
4
5 class PositionalEncoding(nn.Module):
6     def __init__(self, d_model, dropout=0.1, max_len=5000):
7         super(PositionalEncoding, self).__init__()
8         self.dropout = nn.Dropout(p=dropout)
9         pe = torch.zeros(max_len, d_model)
10        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
11        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) /
12                           d_model))
13        pe[:, 0::2] = torch.sin(position * div_term)
14        pe[:, 1::2] = torch.cos(position * div_term)
15        pe = pe.unsqueeze(0).transpose(0, 1)
16        self.register_buffer('pe', pe)
17
18    def forward(self, x):
19        x = x + self.pe[:x.size(0), :]
20        return self.dropout(x)

```

Listato 3.3: Implementazione della classe PositionalEncoding in PyTorch.

Analisi del Codice PositionalEncoding

Esaminiamo il codice un passo alla volta, concentrandoci sulla sua logica intrinseca e sulle sue **parti fondamentali**.

Il Costruttore

`__init__(self, d_model, dropout=0.1, max_len=5000)`

Questa è la **fase di preparazione**, il “*dietro le quinte*” del modulo. Qui vengono impostati e pre-calcolati tutti i componenti necessari al suo funzionamento.

- `self.dropout = nn.Dropout(p=dropout)`: Imposta il livello di *Dropout*.
- `pe = torch.zeros(max_len, d_model)`: Viene creata una “*tela*” vuota, una matrice di zeri chiamata `pe` (*Positional Encoding*). Le sue dimensioni sono definite dalla lunghezza massima della sequenza (`max_len`) e dalla dimensione dei vettori (`d_model`). Questa matrice servirà a contenere le informazioni sulla posizione di ogni token.
- `position = torch.arange(0, max_len, ...).unsqueeze(1)`: Si genera un vettore colonna che elenca esplicitamente le posizioni, da 0 a `max_len-1`. Questo rappresenta l'**indice di posizione assoluta** di ogni parola nella sequenza.
- `div_term = torch.exp(...)`: Rappresenta il termine che modula la frequenza delle onde sinusoidali. Verrà utilizzata nel passo successivo per avere il denominatore numericamente stabile ed efficiente.
- `pe[:, 0::2] = torch.sin(...)` e `pe[:, 1::2] = torch.cos(...)`: La matrice `pe` viene riempita. Sulle colonne **pari** vengono applicate le funzioni seno, mentre sulle colonne **dispari** si usano le funzioni coseno. Ogni posizione (`position`) viene moltiplicata per le diverse frequenze calcolate in `div_term`, generando un **pattern ondulatorio unico** per ogni riga della matrice.
- `pe = pe.unsqueeze(0).transpose(0, 1)`: Le dimensioni della matrice `pe` vengono riorganizzate per renderla compatibile con i tensori di input del modello, che tipicamente hanno una dimensione per il *batch*. La forma finale (`max_len, 1, d_model`) è pronta per essere sommata a un *batch* di dati.
- `self.register_buffer('pe', pe)`: Questo **passaggio è cruciale**. Con `register_buffer`, la matrice `pe` viene salvata come parte integrante dello stato del modello, ma viene contrassegnata come **non addestrabile**. Il *positional encoding* è una componente fissa e deterministica; non deve essere modificata durante il processo di *backpropagation*.

L'Applicazione Pratica

```
forward(self, x)
```

Questa funzione descrive cosa accade quando i dati attraversano il modulo. L'input `x` è un tensore contenente gli *embedding* della serie storica, con dimensioni (`lunghezza_sequenza, batch_size, d_model`).

- `x = x + self.pe[:x.size(0), :]`: **Qui avviene l'unione.** Il *positional encoding* pre-calcolato (`self.pe`) viene "tagliato" per corrispondere esattamente alla lunghezza della sequenza di input. Successivamente, viene **semplicemente sommato** al tensore degli *embedding* `x`. Grazie al meccanismo di *broadcasting* di PyTorch, questa matrice di posizioni viene sommata a ciascun elemento del *batch*.
- `return self.dropout(x)`: Infine, al tensore risultante viene applicato il *dropout* per la regolarizzazione, prima di passarlo al livello successivo del modello.

Assemblaggio del Modello Transformer Encoder-Decoder

La classe `TransformerAutoregressive` implementa un modello Transformer completo di tipo encoder-decoder, specificamente progettato per la previsione di serie storiche in modo autoregressivo.

```

1 import math
2 import torch
3 import torch.nn as nn
4
5 # Assumiamo che la classe PositionalEncoding sia definita altrove
6 class TransformerAutoregressive(nn.Module):
7     def __init__(self, num_features, embed_size, num_heads, num_layers, ff_hidden_dim,
8                  dropout=0.1):
9         super(TransformerAutoregressive, self).__init__()
10        self.embed_size = embed_size
11        self.num_features = num_features
12
13        self.input_linear = nn.Linear(num_features, embed_size)
14        self.target_linear = nn.Linear(num_features, embed_size)
15        self.pos_encoder = PositionalEncoding(embed_size, dropout)
16
17        self.transformer = nn.Transformer(
18            d_model=embed_size, nhead=num_heads,
19            num_encoder_layers=num_layers, num_decoder_layers=num_layers,
20            dim_feedforward=ff_hidden_dim, dropout=dropout, batch_first=False
21        )
22        self.fc_out = nn.Linear(embed_size, num_features)
23
24    def _generate_square_subsequent_mask(self, sz, device):
25        mask = (torch.triu(torch.ones(sz, sz, device=device)) == 1).transpose(0, 1)
26        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1,
27                               float(0.0))
28        return mask
29
30    def forward(self, src, tgt):
31        device = src.device
32        tgt_seq_len = tgt.size(0)
33        tgt_mask = self._generate_square_subsequent_mask(tgt_seq_len, device)
34
35        src_emb = self.input_linear(src) * math.sqrt(self.embed_size)
36        src_pos = self.pos_encoder(src_emb)
37
38        tgt_emb = self.target_linear(tgt) * math.sqrt(self.embed_size)
39        tgt_pos = self.pos_encoder(tgt_emb)
40
41        output = self.transformer(src_pos, tgt_pos, src_mask=None, tgt_mask=tgt_mask)
42        return self.fc_out(output)
43
44    def predict(self, src, horizon, device):
45        self.eval()
46        with torch.no_grad():
47            src_emb = self.input_linear(src) * math.sqrt(self.embed_size)
48            src_pos = self.pos_encoder(src_emb)
49            memory = self.transformer.encoder(src_pos)
50
51            decoder_input = torch.zeros(1, 1, self.num_features).to(device)
52            predictions = []
53
54            for _ in range(horizon):
55                tgt_seq_len = decoder_input.size(0)
56                tgt_mask = self._generate_square_subsequent_mask(tgt_seq_len, device)

```

```

55     tgt_emb = self.target_linear(decoder_input) * math.sqrt(self.embed_size)
56     tgt_pos = self.pos_encoder(tgt_emb)
57
58     decoder_output = self.transformer.decoder(tgt_pos, memory, tgt_mask=
59                     tgt_mask)
60     last_step_output = decoder_output[-1, :, :]
61     prediction = self.fc_out(last_step_output).unsqueeze(0)
62
63     predictions.append(prediction)
64     decoder_input = torch.cat([decoder_input, prediction], dim=0)
65
66     final_predictions = torch.cat(predictions, dim=0)
67     return final_predictions.permute(1, 0, 2)

```

Listato 3.4: Implementazione della classe TransformerAutoregressive in PyTorch.

Nelle sezioni seguenti verranno analizzati gli aspetti chiave del codice.

La Costruzione del Modello

`__init__`

In questa funzione vengono inizializzati tutti i componenti della rete.

`self.input_linear` e `self.target_linear`

- **Scopo:** Sono presenti due layer lineari distinti: uno per la sequenza di input dell'encoder (`src`) e uno per la sequenza di input del decoder (`tgt`). Entrambi proiettano i dati da `num_features` a `embed_size` per "arricchire" l'informazione. Avere due layer separati offre al modello maggiore flessibilità, permettendogli di apprendere trasformazioni diverse per l'input storico e per l'input che guida la previsione.

`self.pos_encoder = PositionalEncoding(...)`

- **Scopo:** Aggiunge l'informazione sulla posizione (tempo) a ogni punto delle sequenze di input e target dopo che sono state proiettate nello spazio di embedding.

`self.transformer = nn.Transformer(...)`

- **Scopo:** È il cuore del modello, contenente la pila di encoder e decoder. L'argomento `batch_first=False` (che è il default) viene esplicitato, chiarendo che il modello si aspetta i dati nel formato (`lunghezza_sequenza, batch_size, numero_features`).

`self.fc_out = nn.Linear(embed_size, num_features)`

- **Scopo:** Questo layer finale proietta l'output del decoder, di dimensione `embed_size`, alla dimensione originale `num_features`. Questa configurazione permette al modello di fare previsioni per serie storiche multivariate, predicendo il valore di tutte le feature per ogni passo temporale futuro.

Prevenire la Visione del Futuro

`_generate_square_subsequent_mask`

- **Scopo:** Genera una "maschera" che impedisce al decoder di "barare" durante l'addestramento. Per la previsione al tempo t , il modello deve poter usare solo le informazioni fino al tempo $t - 1$. Questa maschera viene applicata durante il meccanismo di self-attention del decoder, azzerando l'attenzione verso le posizioni future nella sequenza target. In pratica, forza il modello a imparare a prevedere il passo successivo basandosi solo sul passato.

Il Flusso dei Dati in Addestramento

`forward`

La funzione `forward` definisce come i dati fluiscono attraverso il modello durante la fase di training.

1. **Generazione Maschera:** La prima operazione è chiamare `self._generate_square_subsequent_mask` per creare la maschera del decoder in base alla lunghezza della sequenza target `tgt`.
2. **Processamento src (Encoder):** La sequenza di input `src` viene processata dal `self.input_linear` e poi dal `self.pos_encoder`.
3. **Processamento tgt (Decoder):** Similmente, la sequenza target `tgt` viene processata dal suo layer dedicato, `self.target_linear`, e poi dal `self.pos_encoder`.
4. **Chiamata al Transformer:** Gli input processati e la maschera `tgt_mask` vengono passati al `self.transformer`, che esegue l'intero processo di encoding e decoding.
5. **Output Finale:** L'output del decoder viene passato attraverso `self.fc_out` per ottenere la previsione finale nel formato corretto.

Generazione Autoregressiva delle Previsioni

`predict`

Definisce come il modello genera previsioni passo dopo passo (inferenza).

- **Scopo:** Prevedere una sequenza futura di lunghezza `horizon` a partire da una sequenza di input `src`. Il processo si svolge così:
 1. **Encoding dell'Input:** L'intera sequenza `src` viene processata dall'encoder **una sola volta**. L'output dell'encoder (`memory`) è una rappresentazione contestualizzata dell'intera storia e verrà riutilizzata a ogni passo di previsione.
 2. **Inizio della Generazione:** Si parte con un `decoder_input` fittizio (es. un tensore di zeri), che rappresenta l'inizio della sequenza da prevedere.
 3. **Ciclo Autoregressivo:** Si ripete un ciclo per `horizon` volte:
 - Il `decoder_input` (che contiene le previsioni generate fino a quel momento) viene processato e passato al decoder insieme alla `memory`.
 - Si seleziona solo l'output dell'**ultimo passo temporale**, che è la previsione per il nuovo istante.
 - Questa singola previsione viene salvata e **concatenata** al `decoder_input`, che quindi si allunga di un passo a ogni iterazione.
 4. **Output Finale:** Tutte le singole previsioni vengono concatenate per formare la sequenza finale, che viene permutata nel formato (`batch_size`, `lunghezza_sequenza`, `numero_features`).

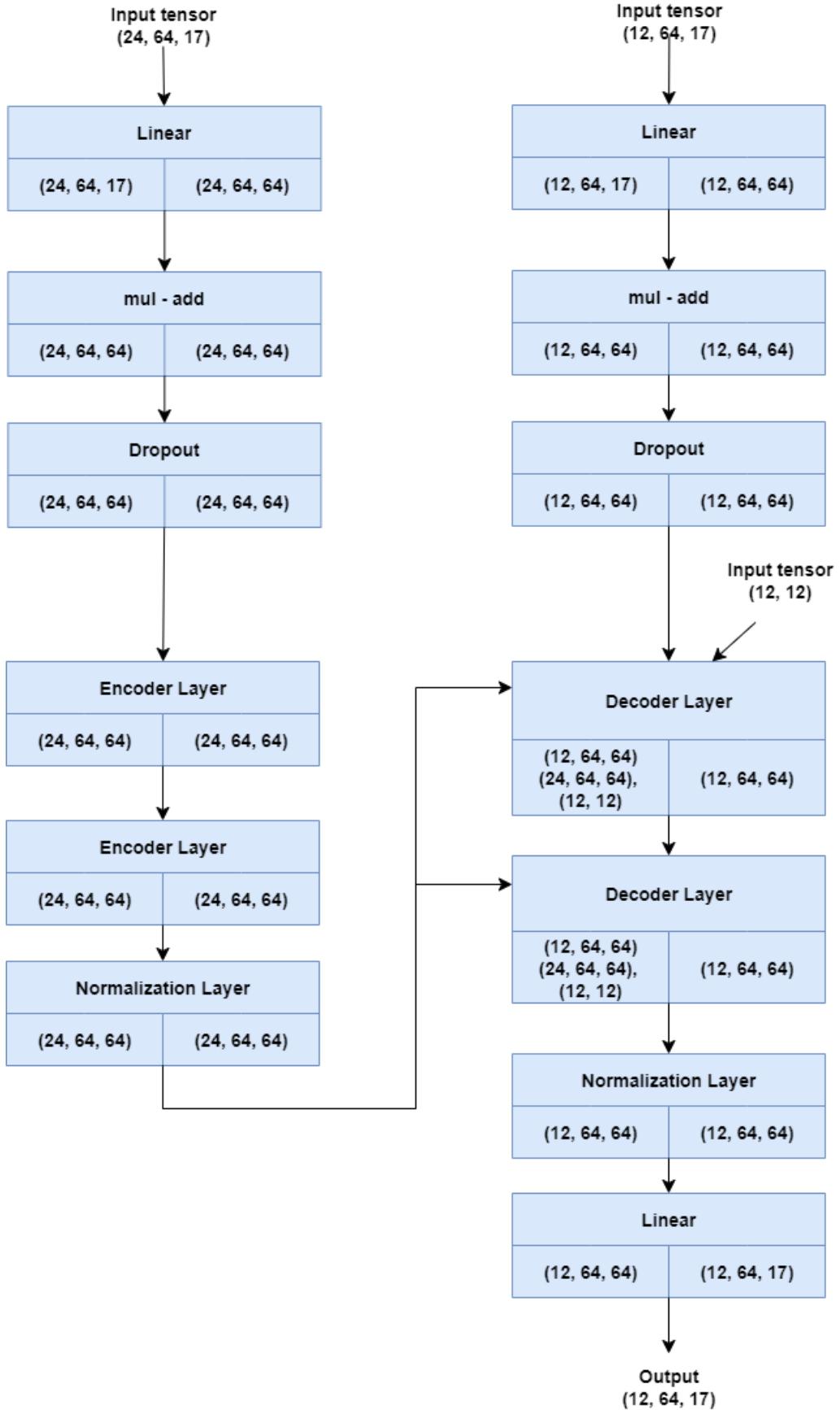


Figura 3.4: Diagramma dell'architettura Transformer Encoder-Decoder implementata, che mostra i flussi di dati paralleli per l'Encoder e il Decoder.

Per concludere, viene analizzato il flusso dei dati attraverso il modello, evidenziando come le dimensioni dei tensori si adattino ai parametri definiti nel codice. L'analisi si concentra sul processo di inferenza implementato nel metodo `predict`, poiché illustra chiaramente il meccanismo di generazione autoregressiva.

I parametri chiave derivati dal codice sono: `N_FEATURES=17`, `INPUT_LENGTH=24`, `embed_size=64` e `num_layers=2`.

1. Percorso dell'Encoder (lato sinistro del diagramma):

- **Input tensor (24, 1, 17):** Rappresenta la sequenza storica di input. Durante la previsione (`predict`), il modello processa un campione alla volta, quindi il `batch_size=1`. La lunghezza della sequenza è `INPUT_LENGTH=24` e il numero di feature è ora **17**.
- **Linear:** Lo strato `self.input_linear` proietta le **17** feature nella dimensione di embedding (`embed_size=64`), trasformando l'output in `(24, 1, 64)`.
- **mul - add e Dropout:** Viene aggiunto il *Positional Encoding* e applicato il *dropout*, mantenendo la dimensione del tensore invariata a `(24, 1, 64)`.
- **Encoder Layer (x2):** Il tensore attraversa una pila di due *Encoder Layer* (`num_layers=2`). L'output finale, che costituisce la **memoria** per il Decoder, avrà dimensione `(24, 1, 64)`.

2. Percorso del Decoder (lato destro del diagramma, in modalità predict):

- **Input tensor (1, 1, 17):** È l'input iniziale per il Decoder, un singolo *time step* (tipicamente un tensore di zeri) per avviare la generazione. La dimensione riflette (`lunghezza_sequenza_iniziale=1`, `batch_size=1`, `num_features=17`).
- **Linear, mul - add, Dropout:** Come per l'Encoder, questo input viene proiettato dallo strato `self.target_linear` e preparato, risultando in un tensore di forma `(1, 1, 64)`.
- **Decoder Layer (x2):** Il tensore `(1, 1, 64)` entra nella pila di due *Decoder Layer*. Ogni strato riceve sia questo input sia la **memoria** `(24, 1, 64)` dall'Encoder, eseguendo prima una *Masked Self-Attention* e poi una *Cross-Attention*, dove la query Q viene dal decoder mentre la K e la V vengono dall'encoder.
- **Linear finale:** L'output del Decoder, di forma `(1, 1, 64)`, viene proiettato dallo strato `self.fc_out` nella dimensione originale delle feature. Poiché `self.fc_out` è definito come `nn.Linear(embed_size, num_features)`, l'output sarà di forma `(1, 1, 17)`.
- **Ciclo Autoregressivo:** Questa previsione `(1, 1, 17)` viene quindi concatenata all'input del decoder per il passo successivo. Al secondo ciclo, l'input del decoder avrà forma `(2, 1, 17)`, verrà proiettato in `(2, 1, 64)` e così via, fino a completare l'`HORIZON` di previsione.

3.3.3 Decoder

Il modello che sarà spiegato in questa sezione a differenza del modello Transformer originale, che comprende sia un *Encoder* che un *Decoder*, adotta un approccio **"Decoder-Only"**. Questa scelta è particolarmente adatta per compiti di natura autoregressiva, come la previsione di serie storiche, dove l'obiettivo è predire i valori futuri di una sequenza basandosi esclusivamente sui valori passati.

L'architettura elabora una sequenza di dati passati per predire la sequenza successiva. Il flusso logico dei dati attraverso il modello è illustrato nella Figura 3.5 e dettagliato nei paragrafi successivi.

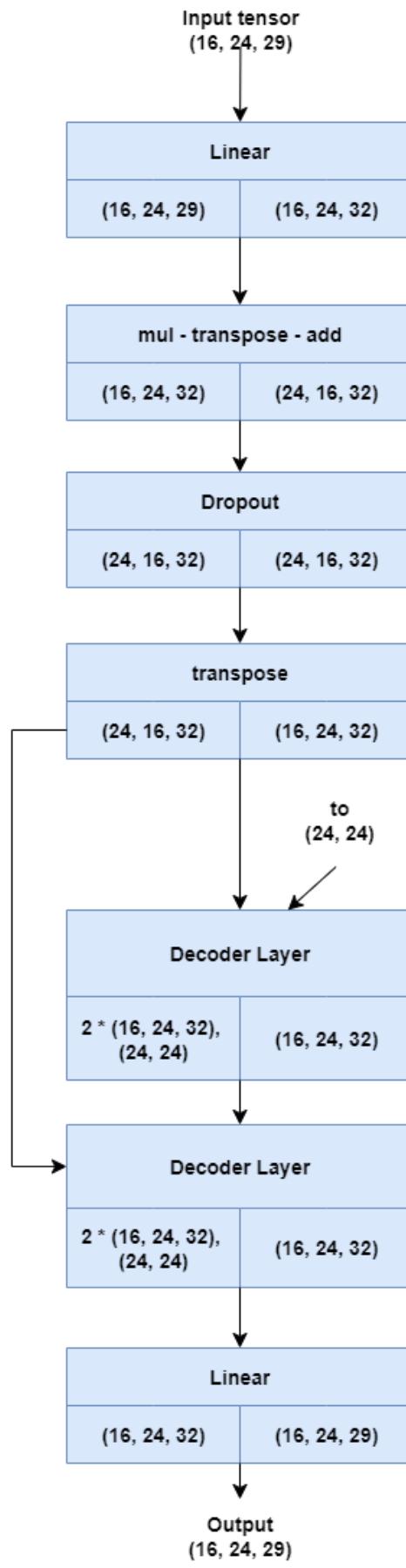


Figura 3.5: Diagramma di flusso dell'architettura Transformer.

Flusso Generale dei Dati

Il modello riceve in input un tensore tridimensionale con dimensioni (`batch_size`, `sequence_length`, `num_features`). Basandoci sugli iperparametri e sul processo di feature engineering del codice, queste dimensioni corrispondono a **(16, 24, N_FEATURES)**. La variabile `N_FEATURES` è determinata dinamicamente dalla funzione `preprocess_data`, che aggiunge diverse feature derivate (come lag, medie mobili, etc.) ai dati originali. Il modello elabora quindi *batch* di 16 sequenze, ciascuna lunga 24 passi temporali.

Il flusso di elaborazione può essere riassunto nei seguenti passaggi chiave:

1. **Embedding Iniziale:** Le `N_FEATURES` di input vengono proiettate da uno strato lineare in uno spazio a dimensione superiore, detto "spazio di embedding" (`embed_size`), che nel nostro caso è **32**.
2. **Codifica Posizionale (*Positional Encoding*):** Come nel design originale dei Transformer, un segnale posizionale viene sommato agli embedding per fornire al modello informazioni sull'ordine temporale di ogni punto nella sequenza.
3. **Blocchi Decoder:** Il cuore del modello è costituito da una pila di N (nel nostro caso, `num_layers=2`) blocchi *Decoder* identici. Ogni blocco applica un meccanismo di auto-attenzione mascherata (per evitare che il modello "veda" il futuro) seguito da una rete neurale *feed-forward*.
4. **Proiezione Finale:** L'output dell'ultimo blocco *Decoder*, che si trova ancora nello spazio di embedding, viene proiettato da uno strato lineare finale a una dimensione di 29. Questo è un punto cruciale: il modello è progettato per prevedere non solo il valore della variabile target al passo successivo, ma tutte le feature che come nel modello precedente serviranno come input per la predizione fino a completare la sequenza richiesta, come fatto anche con il modello precedente.

La Tabella 3.1 riassume la trasformazione del tensore attraverso il modello, riconciliando l'architettura con il codice fornito.

Fase del Modello	Operazione Principale	Dim. Input	Dim. Output
Input	Dati grezzi con feature aggiunte <code>nn.Linear</code>	(16, 24, 29)	-
Livello Lineare		(16, 24, 29)	(16, 24, 32)
Codifica Posizionale	Aggiunta di segnali sin/cos	(16, 24, 32)	(16, 24, 32)
Blocco Decoder 1	Auto-attenzione + Feed-Forward	(16, 24, 32)	(16, 24, 32)
Blocco Decoder 2	Auto-attenzione + Feed-Forward	(16, 24, 32)	(16, 24, 32)
Livello Lineare Finale	<code>nn.Linear</code>	(16, 24, 32)	(16, 24, 29)
Output	Previsione della variabile target	-	(16, 24, 29)

Tabella 3.1: Trasformazione delle dimensioni del tensore attraverso i livelli del modello, adattata al codice di previsione.

Analisi dei Componenti Chiave

Nelle prossime pagine saranno spiegate le parti del codice in riferimento al codice 3.5

Input

Linear

Il primo blocco del diagramma, `Linear`, prende l'input con `num_features` e lo proietta in uno spazio a 32 dimensioni (`embed_size`).

Nel codice, questo strato viene definito nel costruttore `__init__` della classe `DecoderOnlyTransformer`:

```
1 self.input_linear = nn.Linear(num_features, embed_size)
```

Viene poi utilizzato all'inizio della funzione `forward` per trasformare l'input `src`:

```
1 src_emb = self.input_linear(src) * math.sqrt(self.embed_size)
```

mul - transpose - add

Questo blocco nel diagramma è una rappresentazione sintetica di più passaggi che costituiscono la **Codifica Posizionale** (*Positional Encoding*).

- **mul (moltiplicazione)**: Corrisponde alla scalatura dell'embedding per stabilizzare l'addestramento.

```
1 ... * math.sqrt(self.embed_size)
```

- **transpose + add + transpose**: Questo è il cuore del `PositionalEncoding`. La somma (add) del segnale posizionale è incapsulata nella chiamata a `self.pos_encoder`.

```
1 src_pos = self.pos_encoder(src_emb.transpose(0, 1)).transpose(0, 1)
```

L'operazione di `add` avviene specificamente dentro la funzione `forward` della classe `PositionalEncoding`:

```
1 # 'x' e' l'embedding, 'self.pe' sono i valori posizionali
2 x = x + self.pe[:x.size(0), :]
```

Decoder Layer

I due blocchi `Decoder Layer` sono il motore del modello, dove avviene l'auto-attenzione mascherata.

Nel codice, vengono prima definiti come un singolo `decoder_layer` e poi impilati (`num_layers=2`) dentro `self.transformer_decoder`:

```
1 decoder_layer = nn.TransformerDecoderLayer(...)
2 self.transformer_decoder = nn.TransformerDecoder(decoder_layer,
3                                                 num_layers=num_layers)
```

L'intera pila di Decoder viene eseguita con una sola chiamata nella funzione `forward`:

```
1 output = self.transformer_decoder(tgt=src_pos, memory=src_pos,
2                                   tgt_mask=tgt_mask)
```

Output

`Linear`

L'ultimo blocco `Linear` serve a proiettare l'output del Transformer (dallo spazio di *embedding*) al numero di feature.

Nel codice, questo strato è definito come `self.fc_out` nel costruttore `__init__`:

```
1 self.fc_out = nn.Linear(embed_size, num_features)
```

Viene usato come ultimissimo passo nella funzione `forward`:

```
1 out = self.fc_out(output)
```

Implementazione in PyTorch

Di seguito è visualizzato il codice seguente che definisce la classe `DecoderOnlyTransformer` che implementa l'architettura descritta.

```

1  class DecoderOnlyTransformer(nn.Module):
2      def __init__(self, num_features, embed_size, num_heads, num_layers, ff_hidden_dim,
3                   dropout=0.1):
4          super(DecoderOnlyTransformer, self).__init__()
5          self.embed_size = embed_size
6          self.input_linear = nn.Linear(num_features, embed_size)
7          self.pos_encoder = PositionalEncoding(embed_size, dropout)
8          decoder_layer = nn.TransformerDecoderLayer(
9              d_model=embed_size,
10             nhead=num_heads,
11             dim_feedforward=ff_hidden_dim,
12             dropout=dropout,
13             batch_first=True
14         )
15         self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=
16                                         num_layers)
17
18         self.fc_out = nn.Linear(embed_size, num_features)
19
20     def generate_square_subsequent_mask(self, sz):
21         return torch.triu(torch.full((sz, sz), float('-inf')), diagonal=1)
22
23     def forward(self, src):
24         src_emb = self.input_linear(src) * math.sqrt(self.embed_size)
25         src_pos = self.pos_encoder(src_emb.transpose(0, 1)).transpose(0, 1)
26
27         # La maschera per il target, che in un decoder-only l'input stesso
28         tgt_mask = self.generate_square_subsequent_mask(src.size(1)).to(src.device)
29
30         # In un decoder-only, l'input (src) funge sia da target (tgt) che da memoria (
31         # memory)
32         output = self.transformer_decoder(tgt=src_pos, memory=src_pos, tgt_mask=tgt_mask)
33
34         out = self.fc_out(output)
35         return out

```

Listato 3.5: Definizione della classe DecoderOnlyTransformer in PyTorch.

3.3.4 Chronos

Chronos rappresenta un approccio innovativo al *forecasting* di serie temporali, basato su un'idea fondamentale: **trattare le sequenze numeriche come un linguaggio** e impiegare modelli linguistici (*Language Models*) per apprenderne le dinamiche e prevederne il futuro. La filosofia di Chronos è **minimalista**: invece di ricorrere ad architetture complesse e specifiche per il dominio temporale, il framework adatta modelli *Transformer* standard, come la famiglia T5, al compito di *forecasting* [20].

Questo viene realizzato attraverso un processo di **tokenizzazione** che converte i valori continui delle serie temporali in un vocabolario discreto e finito. Il modello viene poi addestrato a prevedere il "token" successivo in una sequenza, esattamente come farebbe con le parole in una frase. Il vantaggio principale di questo approccio è la sua straordinaria capacità di **generalizzazione**. I modelli Chronos, pre-addestrati su un corpus vasto e variegato di serie temporali, possono generare previsioni accurate su dati mai visti prima (**zero-shot forecasting**), senza necessità di ri-addestramento o *fine-tuning*.

Per questo lavoro di tesi, è stata utilizzata una delle implementazioni ufficiali pre-addestrate, come mostrato nel codice:

```

1 model = ChronosPipeline.from_pretrained(
2     "amazon/chronos-t5-small",
3     device_map=DEVICE,
4     torch_dtype=torch.bfloat16 if torch.cuda.is_available() else torch.float32,
5 )

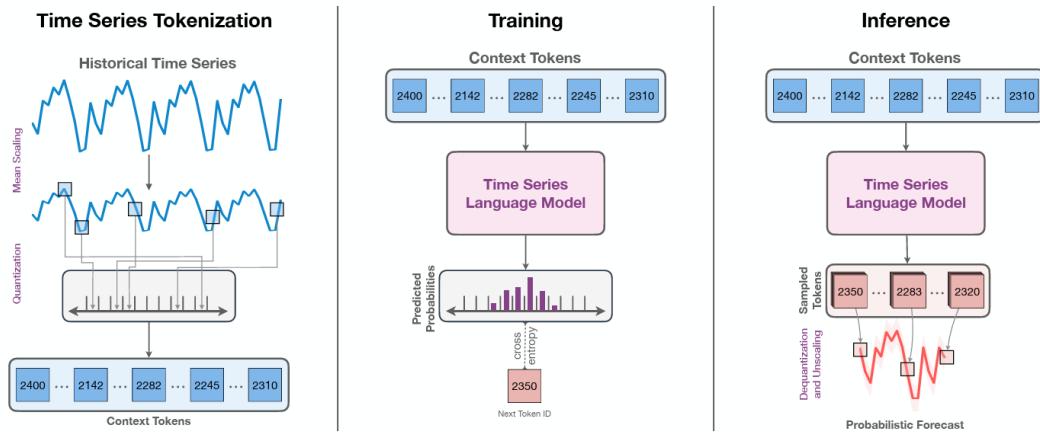
```

Listato 3.6: Definizione del modello chronos

La scelta del modello **chronos-t5-small** (46M parametri) rappresenta un ottimo compromesso tra accuratezza e requisiti computazionali, permettendo di sfruttare la potenza del framework pre-addestrato in modo efficiente per l'analisi condotta.

Architettura e Flusso Operativo di Chronos

Il funzionamento di Chronos, dalla ricezione dei dati grezzi alla generazione della previsione finale, si articola in tre fasi principali, come illustrato in Figura 3.6 del paper di riferimento (Ansari et al., 2024), oppure nella figura inserita di seguito.

**Figura 3.6:** fasi effettuate dal modello chronos per la generazione di una previsione [8]

Fase 1: Tokenizzazione - La Traduzione dei Dati in Linguaggio

Il passaggio più critico del framework è la conversione di una serie temporale da una sequenza di numeri reali a una sequenza di *token* discreti. Questo avviene in due passaggi.

- Scalatura (*Scaling*):** Per normalizzare serie con scale molto diverse, Chronos utilizza il **mean scaling**, dividendo ogni valore della serie per la media dei valori assoluti del contesto storico ($s = \frac{1}{C} \sum_{i=1}^C |x_i|$). Questa tecnica preserva i valori zero, che sono spesso semanticamente significativi. Nel nostro codice, abbiamo introdotto un passaggio di *preprocessing* aggiuntivo per stabilizzare la varianza e gestire possibili trend esponenziali, che possono essere una limitazione per Chronos:

```

1 target_series_log = np.log1p(target_series)

```

Listato 3.7: Codice utilizzato per stabilizzare e gestire i trend

Questa trasformazione, implementata con `np.log1p` per gestire correttamente anche valori pari a zero, comprime i valori su una scala più gestibile. Ciò mitiga il rischio che valori estremi (*outlier*) o un trend esponenziale possano saturare il range di quantizzazione del modello, una pratica che migliora la robustezza del *forecasting*.

- Quantizzazione (*Quantization*):** I valori scalati vengono poi mappati a un vocabolario finito di "bin" (contenitori). Ogni *bin* rappresenta un intervallo di valori. Chronos utilizza un **uniform binning** su un range predefinito, trasformando di fatto il problema di regressione in un problema di classificazione multiclass: prevedere il *bin* corretto per il *time step* successivo.

Fase 2: Il Modello Linguistico - T5

Il cuore pulsante di Chronos è un'architettura *Transformer*. Il modello `chronos-t5-small` da noi utilizzato si basa su **T5** (*Text-to-Text Transfer Transformer*) [22]. Il nome "*Text-to-Text*" riflette perfettamente il framework: l'input è una sequenza di *token* (trattata come testo) e l'output è un'altra sequenza di *token*. La sua architettura *encoder-decoder* è particolarmente adatta a questo compito.

- L'**encoder** processa l'intera sequenza di *token* del contesto storico per creare una rappresentazione latente che cattura le dinamiche della serie.
- Il **decoder** utilizza questa rappresentazione per generare, in modo autoregressivo, la sequenza di *token* futuri.

Il modello è addestrato con una **categorical cross-entropy loss**, che mira a massimizzare la probabilità del *token* corretto ad ogni passo temporale. È importante notare che il modello impara la relazione ordinale tra i *token* (ad esempio, che il *bin* 205 è vicino al 206) **implicitamente dai dati**, senza che questa informazione venga fornita a priori.

Fase 3: Inferenza - Previsione Multi-Step con Cross-Validation Temporale

Il processo di inferenza adottato nel nostro esperimento utilizza un approccio di **previsione multi-step diretta**, basato su una **strategia di validazione incrociata per serie temporali** (*TimeSeriesSplit*). Il modello Chronos viene testato su più fold, ognuno dei quali simula un diverso scenario di previsione futura.

1. **Contesto Dinamico su Ogni Fold:** All'interno di ogni fold, il modello riceve come input un contesto di lunghezza fissa, pari a `SEQ_LENGTH`, rappresentante gli ultimi valori reali del set di training. Questo contesto costituisce la base informativa da cui partire per prevedere l'intero orizzonte del fold di test (es. 12 step consecutivi nel futuro).

```
1 context = train_data[-SEQ_LENGTH:]
```

Listato 3.8: Codice utilizzato per ricavare l'input

2. **Previsione Multi-Step con Sampling Probabilistico:** A partire da questo contesto, il modello Chronos esegue una previsione multi-step diretta, generando simultaneamente `test_size` punti futuri (12 nel nostro caso), ciascuno predetto in maniera indipendente. La natura probabilistica del modello è sfruttata campionando 20 possibili traiettorie per ciascun passo.

```
1 outputs = model.predict(
2     torch.tensor(context),
3     prediction_horizon, # es. 12 step
4     num_samples=20,
5 )
```

Listato 3.9: Codice utilizzato per effettuare una previsione

3. **Aggregazione delle Previsioni tramite Mediana:** Per ogni passo predetto, si calcola la **mediana** dei campioni generati, ottenendo così una previsione puntuale robusta rispetto alle variazioni stocastiche. L'output viene poi riportato nello spazio originale applicando la trasformazione inversa del logaritmo (esponenziale meno uno).

```
1 prediction_scaled = outputs.median(dim=1).values.squeeze(0).cpu().numpy()
2 y_pred_final = np.expm1(prediction_scaled)
```

Listato 3.10: Codice utilizzato per effettuare l'aggregazione delle Previsioni tramite Mediana

4. **Raccolta e Valutazione delle Previsioni:** Le previsioni prodotte per ciascun fold vengono raccolte e confrontate con i corrispondenti valori reali del set di test, anch'essi riportati nello spazio originale. Le metriche di errore (RMSE, MAE, MSE) vengono calcolate sia a livello di fold che globalmente sull'intero processo di validazione.

L'Importanza del Pre-addestramento per la Previsione Rolling

Anche se la nostra strategia di inferenza è cambiata in un *rolling forecast*, la ragione per cui `chronos-t5-small` è efficace risiede sempre nel suo estensivo pre-addestramento. Il modello ha **"imparato il linguaggio delle serie temporali"** da un corpus di 28 dataset pubblici, per un totale di circa 890.000 serie temporali da domini eterogenei.

Questa vasta conoscenza di base, ottenuta anche grazie a tecniche di ***data augmentation*** come **TSMixup** e **KernelSynth**, è ciò che gli permette di interpretare efficacemente la finestra di contesto di 24 punti e generare una previsione accurata per lo step successivo.

In sostanza, stiamo sfruttando la sua **capacità generalista**, appresa su migliaia di esempi, per applicarla a un compito molto specifico. Questo processo massivo di addestramento è ciò che conferisce a Chronos la sua notevole performance anche in questo scenario, senza la necessità di un costoso *fine-tuning*.

3.4 Ambiente utilizzato e setup per il training

3.4.1 Training

Per l'addestramento dei nostri modelli, abbiamo impiegato una metodologia specifica per le serie temporali, nota come ***rolling forecasting cross-validation*** (o convalida incrociata a finestra mobile). Abbiamo scelto questo approccio perché simula uno scenario realistico in cui un modello viene periodicamente riaddestrato con nuovi dati per prevedere il futuro, preservando rigorosamente l'ordine cronologico degli eventi.

3.4.1.1 Ciclo di Cross-Validation a Finestra Mobile

Il cuore del nostro processo di training è un ciclo `for` che itera per un numero predefinito di **5 fold**. L'inizio di ogni iterazione è tracciato nell'output del codice:

```
--- FOLD 1/5 ---
```

All'interno di questo ciclo, abbiamo suddiviso dinamicamente il *dataset* in un *set* di *training* e uno di validazione. A differenza di una *cross-validation* standard, abbiamo implementato una finestra di *training* che si espande ad ogni passo, includendo i dati del *fold* precedente. L'output del codice conferma questa strategia:

```
--- FOLD 1/5 ---
Lunghezza training: 264, Lunghezza test: 12
--- FOLD 2/5 ---
Lunghezza training: 276, Lunghezza test: 12
```

Come si può notare, il *set* di validazione (*test*) mantiene una dimensione fissa di 12 campioni temporali futuri, mentre il *set* di *training* (*training*) si espande progressivamente. Abbiamo adottato questo metodo *rolling* per garantire che il modello impari da una cronologia di dati sempre più ampia, simulando un'applicazione reale.

3.4.1.2 Addestramento, Ottimizzazione e Valutazione

All'interno di ogni *fold* del ciclo, abbiamo svolto le seguenti operazioni chiave:

1. **Ottimizzazione del Training:** Prima di avviare l'addestramento, abbiamo definito l'ottimizzatore e uno *scheduler* per il *learning rate*, un passaggio fondamentale per garantire una convergenza efficiente del modello.

```
1   optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE,
2                           weight_decay=WEIGHT_DECAY)
3   scheduler = ReduceLROnPlateau(optimizer, 'min', factor=0.5, patience=PATIENCE
4                                 //2)
```

Listato 3.11: Codice utilizzato per definire l'ottimizzatore

Abbiamo utilizzato l'ottimizzatore ***Adam***, una scelta consolidata per i modelli di *deep learning*, che adatta il *learning rate* per ciascun parametro. Il ***weight_decay*** è stato impiegato come forma di regolarizzazione per prevenire l'*overfitting*. Lo ***scheduler ReduceLROnPlateau*** monitora la metrica di validazione e riduce il *learning rate* (***factor=0.5***) se questa non migliora per un certo numero di epoche (***patience***). Abbiamo integrato questa tecnica di *adaptive learning rate* per permettere al modello di affinare la ricerca della soluzione ottimale.

2. **Addestramento:** Successivamente, abbiamo addestrato il modello esclusivamente sui dati di *training* del *fold* corrente (***train_data***).

```
1     history = model.fit(X_train, y_train, validation_data=(X_val, y_val), ...)
```

Listato 3.12: Codice utilizzato per definire l'addestramento

3. **Previsione e Valutazione:** Completato il *training*, il modello ha effettuato le previsioni sul *set* di validazione (***val_data***). Abbiamo quindi confrontato queste previsioni con i valori reali per calcolare l'**RMSE** (*Root Mean Squared Error*), MSE e MAE, salvando il risultato per il *fold* specifico. Tali metriche verranno spiegate nel capitolo successivo.

```
1     rmse_fold = np.sqrt(mean_squared_error(y_test_final, y_pred_final))
2     mse_fold = mean_squared_error(y_test_final, y_pred_final)
3     mae_fold = mean_absolute_error(y_test_final, y_pred_final)
4     print(f" RMSE Fold {fold + 1}: {rmse_fold:.4f}")
5     print(f" MSE Fold {fold + 1}: {mse_fold:.4f}")
6     print(f" MAE Fold {fold + 1}: {mae_fold:.4f}")
```

Listato 3.13: Codice utilizzato per trovare le metriche

3.4.1.3 Calcolo delle Prestazioni Finali

Al termine dei 7 *fold*, abbiamo aggregato le metriche di errore raccolte per ottenere una valutazione finale e robusta delle prestazioni del modello.

```
1     final_rmse = np.sqrt(mean_squared_error(y_true_combined, y_pred_combined))
2     final_mae = mean_absolute_error(y_true_combined, y_pred_combined)
3     final_mse = mean_squared_error(y_true_combined, y_pred_combined)
4
5     print("\n--- Risultati Finali di Chronos su Cross-Validation ---")
6     print(f" RMSE Medio su tutti i fold: {final_rmse:.4f}")
7     print(f" MAE Medio su tutti i fold: {final_mae:.4f}")
8     print(f" MSE Medio su tutti i fold: {final_mse:.4f}")
```

Listato 3.14: Codice utilizzato per definire le metriche finali

L'**RMSE Medio** rappresenta la nostra stima più affidabile dell'errore di previsione del modello su dati futuri. L'applicazione di questa identica e rigorosa procedura a tutti e quattro i modelli ha garantito un confronto equo e basato su una solida validazione statistica.

3.4.2 Ottimizzazione degli Iperparametri con Optuna

La performance di un modello di deep learning, e in particolare di un'architettura complessa come il Transformer, è fortemente sensibile alla scelta dei suoi iperparametri. Un'inadeguata configurazione può portare a fenomeni di underfitting o overfitting, tempi di addestramento eccessivi o, in generale, a una ridotta capacità predittiva. Per affrontare questa sfida in modo sistematico ed efficiente, è stato impiegato **Optuna**, un framework di ottimizzazione iperparametrica di nuova generazione. A differenza di approcci tradizionali come la *Grid Search* o la *Random Search*, Optuna utilizza algoritmi di ricerca più sofisticati, come il *Tree-structured Parzen Estimator* (TPE), per esplorare lo spazio degli iperparametri in modo più intelligente, concentrando la ricerca nelle regioni più promettenti.

Il processo di ottimizzazione è stato incapsulato all'interno di una funzione obiettivo, denominata ***objective(trial)***, che rappresenta il nucleo dell'integrazione con Optuna. Questa

funzione definisce un singolo esperimento (*trial*) e restituisce una metrica di performance che Optuna si prefigge di minimizzare o massimizzare.

Definizione dello Spazio di Ricerca

All'interno della funzione `objective`, per ogni *trial*, viene definito lo spazio di ricerca degli iperparametri. Tramite i metodi forniti dall'oggetto *trial*, sono stati specificati gli intervalli e le tipologie per ciascun iperparametro da ottimizzare, di seguito spiegati alcuni:

- **lr**: Il *learning rate*, campionato da una distribuzione log-uniforme tra 10^{-5} e 10^{-3} tramite `trial.suggest_float("lr", 1e-5, 1e-3, log=True)`.
- **embed_size**: La dimensione dei vettori di embedding, scelta tra i valori discreti {32, 64, 128} con `trial.suggest_categorical`.
- **num_layers**: Il numero di layer per l'encoder e il decoder del Transformer, un intero tra 2 e 4 (`trial.suggest_int`).
- **nhead**: Il numero di "teste" nel meccanismo di multi-head attention, scelto tra {2, 4, 8} (`trial.suggest_categorical`).
- **dropout**: Il tasso di dropout, un valore continuo tra 0.1 e 0.5.
- **weight_decay**: Il parametro di regolarizzazione L2, campionato da una distribuzione log-uniforme tra 10^{-5} e 10^{-3} .

È stato inoltre, per il modello Encoder-Decoder e solo Decoder, introdotto un vincolo architettonurale fondamentale: la dimensione dell'embedding (`embed_size`) deve essere divisibile per il numero di teste di attenzione (`nhead`). Le combinazioni non valide vengono scartate preventivamente tramite il meccanismo di *pruning* di Optuna, sollevando un'eccezione `optuna.exceptions.TrialPruned()`. Questo approccio evita di sprecare risorse computazionali su configurazioni del modello non eseguibili.

Strategia di Valutazione Robusta

Per ottenere una stima affidabile e non viziata delle performance di una data configurazione di iperparametri, è stata adottata una strategia di validazione incrociata specifica per le serie storiche. Utilizzando la classe `TimeSeriesSplit` di Scikit-learn con 3 *split*, il dataset viene suddiviso in fold consecutivi, garantendo che il set di validazione contenga sempre dati temporalmente successivi a quelli del set di addestramento. Questo previene il *data leakage* e simula realisticamente uno scenario di previsione reale.

Per ogni *fold*, un nuovo modello Transformer viene istanziato con gli iperparametri suggeriti dal *trial* corrente. Il modello viene quindi addestrato sul set di training del fold e valutato sul relativo set di test. La metrica di valutazione scelta è il *Root Mean Squared Error* (RMSE), calcolata sui valori de-normalizzati per essere interpretabile nell'unità di misura originale. Il valore finale che la funzione `objective` restituisce a Optuna è la **media degli RMSE** calcolati sui 3 fold. Questo approccio mitiga il rischio che una buona performance sia dovuta a una particolare suddivisione fortuita dei dati.

Esecuzione dello Studio

L'intero processo di ottimizzazione viene gestito da un oggetto `study`, inizializzato con l'obiettivo di minimizzare la metrica di errore: `study = optuna.create_study(direction="minimize")`. L'ottimizzazione viene avviata tramite la chiamata `study.optimize(objective, n_trials=70)`, che esegue la funzione obiettivo per 70 volte. Al termine, lo `study` contiene i risultati di tutti i *trial* e consente di identificare facilmente la combinazione di iperparametri che ha prodotto il minor RMSE medio, la quale è stata poi selezionata per l'addestramento del modello finale.

Capitolo 4

Analisi dei risultati

4.1 Introduzione

Dopo avere analizzato le varie architetture, questo capitolo si concentra sulla fase culminante del processo di ricerca: la **valutazione rigorosa e il confronto sistematico** delle performance dei modelli implementati. Questa analisi non si limita a una mera quantificazione dell'errore, ma mira a comprendere in profondità i **punti di forza, le debolezze e le caratteristiche operative** di ciascun approccio, al fine di determinare quale architettura si dimostri più adatta al contesto applicativo specifico.

Contesto della valutazione

Il capitolo presenta i risultati ottenuti da quattro distinte architetture di rete neurale. Si parte da un modello di riferimento consolidato, la rete *Long Short-Term Memory* (LSTM). Successivamente, l'analisi si sposta verso le architetture basate sul meccanismo di attenzione, che costituiscono lo stato dell'arte attuale. Verranno esaminati due modelli *Transformer* customizzati: un'architettura *Encoder-Decoder* completa, e una più snella architettura *Decoder-only*. Infine, per stabilire un benchmark con modelli pre-addestrati su larga scala, viene valutato il modello *Chronos*, un'architettura *Transformer foundation model* sviluppata da Amazon, applicata in un contesto di *zero-shot learning*, ovvero senza alcun *fine-tuning* sul nostro dataset specifico.

Metodologia di valutazione

La metodologia di valutazione adottata è stata progettata per garantire la massima oggettività e robustezza statistica. Come già descritto nel capitolo precedente il pilastro della nostra strategia è la **validazione incrociata (cross-validation)**. Le performance di ciascun modello sono state poi misurate attraverso un set di metriche quantitative standard nel campo del machine learning: *Mean Squared Error* (MSE), *Root Mean Squared Error* (RMSE) e *Mean Absolute Error* (MAE). Sebbene una variante dell'MSE sia stata impiegata come funzione di costo durante la fase di addestramento del modello Encoder-decoder, l'analisi congiunta di tutte e tre le metriche offre una visione più completa della distribuzione e della natura degli errori di previsione.

Oltre all'analisi puramente numerica, un'enfasi significativa è posta sull'**interpretazione qualitativa dei risultati** attraverso un'ampia gamma di strumenti di visualizzazione. Per ogni modello, verranno presentati e discussi grafici specifici, tra cui l'andamento delle curve di apprendimento, il confronto visivo tra valori predetti e reali, e l'analisi dei residui. Questi strumenti grafici sono fondamentali per andare oltre il singolo valore numerico di una metrica e per comprendere il comportamento del modello, come la sua capacità di catturare picchi e stagionalità, la presenza di *bias* sistematici o la distribuzione degli errori nel tempo.

Struttura del capitolo

La struttura di questo capitolo è la seguente: si inizierà con una descrizione dettagliata della metodologia di valutazione (Sezione 4.2), definendo le metriche e le tecniche utilizzate. Seguiranno quattro sezioni distinte (4.3, 4.4, 4.5, 4.6), ciascuna dedicata all'analisi

approfondita di uno dei modelli. Infine, la Sezione 4.7 presenterà un’analisi comparativa onnicomprensiva, mettendo a confronto diretto i risultati quantitativi e qualitativi per trarre le conclusioni finali su quale modello si sia dimostrato il più performante e affidabile per il problema di *forecasting* affrontato.

4.2 Metodologia di valutazione

Una valutazione rigorosa e multidimensionale è essenziale per comprendere appieno le capacità e i limiti di un modello predittivo. In questa sezione, vengono illustrati gli strumenti e le procedure adottate per l’analisi dei risultati. La metodologia si articola su tre pilastri fondamentali: le **metriche quantitative** per la misurazione dell’errore, una **solida strategia di validazione** per garantirne la robustezza statistica, e un’**analisi qualitativa** basata su strumenti di visualizzazione per l’interpretazione del comportamento del modello.

4.2.1 Metriche di performance

La scelta delle metriche di valutazione è un passo cruciale che influenza la percezione delle performance di un modello. Sebbene esistano decine di possibili indicatori, in questo studio ci siamo concentrati su tre delle metriche più diffuse e interpretabili per i problemi di regressione e previsione di serie storiche.

1. Mean Squared Error (MSE)

L’Errore Quadratico Medio è stato il criterio guida per l’ottimizzazione dei modelli durante la fase di addestramento, fungendo da funzione di costo (*loss function*). La sua formula è:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

dove n è il numero di osservazioni, y_i è il valore reale e \hat{y}_i è il valore predetto dal modello. L’MSE misura la media degli errori al quadrato. L’operazione di elevamento al quadrato ha due effetti principali: primo, rende tutti i contributi all’errore positivi; secondo, e più importante, **penalizza in modo più che proporzionale gli errori di maggiore entità**. Un singolo errore di previsione molto grande avrà un impatto sull’MSE significativamente maggiore rispetto a molti piccoli errori. Questo lo rende una metrica particolarmente sensibile agli *outlier* e desiderabile in contesti in cui anche pochi errori gravi sono considerati molto costosi. Per i modelli *LSTM* e *Transformer Decoder-only*, l’MSE standard è stata utilizzata direttamente come funzione di costo per guidare il processo di apprendimento.

Adattamento per il Modello Transformer Encoder-Decoder: la Weighted MSE

Per il modello **Transformer Encoder-Decoder**, è stata impiegata una variante specifica della funzione di costo: una **Mean Squared Error Ponderata (Weighted MSE)**. In questa formulazione, non tutti gli errori nella sequenza di previsione sono trattati allo stesso modo. Viene assegnato un peso maggiore agli errori commessi sui punti temporali più avanzati della finestra di previsione. La logica alla base di questa scelta è duplice: in primo luogo, la previsione a lungo termine è intrinsecamente più difficile e incerta; forzare il modello a concentrarsi maggiormente su questi punti può portare a un apprendimento più robusto delle dinamiche complesse. In secondo luogo, in molti scenari applicativi, l’accuratezza sulle previsioni più lontane è di maggiore importanza strategica. Pertanto, la loss per questo modello penalizza in modo più severo gli errori commessi verso la fine dell’orizzonte di previsione, spingendo il modello a ottimizzare la sua accuratezza a lungo termine. È fondamentale sottolineare che, sebbene la *loss di training* fosse ponderata, le metriche finali di valutazione (MSE, RMSE, MAE) sono state calcolate in modo standard su tutti i modelli per garantire un confronto equo e diretto.

2. Root Mean Squared Error (RMSE)

L'Errore Quadratico Medio Radice non è altro che la radice quadrata dell'MSE:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Il vantaggio principale dell'RMSE risiede nella sua **interpretabilità**. Essendo la radice quadrata dell'MSE, l'RMSE è espresso nella stessa unità di misura della variabile target. Questo permette di comprendere immediatamente l'ordine di grandezza dell'errore medio del modello. Ad esempio, se si sta prevedendo un consumo energetico in kWh, un RMSE di 50 indica che, in media, l'errore di previsione del modello si discosta di circa 50 kWh dal valore reale. Come l'MSE, anche l'RMSE è sensibile agli errori di grandi dimensioni, ma il suo valore è più facilmente rapportabile alla scala del problema.

3. Mean Absolute Error (MAE)

L'Errore Assoluto Medio offre una prospettiva complementare sull'errore. La sua formula è:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

A differenza delle metriche quadratiche, il MAE calcola la media dei valori assoluti degli errori. Questo significa che tutti gli errori, grandi o piccoli, pesano in modo direttamente proporzionale alla loro magnitudine. Il MAE è quindi intrinsecamente **più robusto e meno sensibile agli outlier** rispetto a MSE e RMSE. Fornisce una rappresentazione chiara e diretta dell'entità media dell'errore, senza l'enfasi sui grandi scostamenti. Confrontare il valore di RMSE con quello del MAE può fornire indicazioni sulla varianza degli errori: **un RMSE significativamente più alto del MAE suggerisce la presenza di errori occasionali di grande entità** che vengono fortemente penalizzati dalla metrica quadratica.

4.2.2 Strategia di convalida

La *K-Fold Cross-Validation*, già descritta nel capitolo precedente (sezione 3.4.1.1), è una tecnica di valutazione che suddivide il dataset in K partizioni. In ciascuna delle K iterazioni, una partizione diversa viene utilizzata per la validazione, mentre le restanti $K-1$ servono per l'addestramento. Le metriche ottenute vengono poi aggregate per fornire una stima più robusta e imparziale delle performance del modello.

4.2.3 Strumenti di analisi visiva

L'analisi quantitativa, sebbene fondamentale, non è sufficiente per una comprensione completa. L'ispezione visiva dei risultati è uno strumento diagnostico insostituibile. Per questo motivo, per ogni modello sono stati generati i seguenti grafici:

- **Andamento della Loss (Training e Validation):** Questo grafico mostra l'evoluzione del valore della funzione di costo (MSE) sulle epoche di addestramento, sia per i dati di *training* che per quelli di validazione. È cruciale per diagnosticare la convergenza del modello e per identificare fenomeni di *overfitting* (quando la *loss* di *training* continua a diminuire mentre quella di validazione stagna o aumenta).
- **Grafico delle Previsioni vs. Dati Reali:** Una visualizzazione diretta che sovrappone la serie storica delle previsioni a quella dei dati reali. Questo permette una valutazione qualitativa immediata della capacità del modello di seguire il trend, la stagionalità e i picchi della serie originale.
- **Andamento dei Residui nel Tempo:** I residui (la differenza $y_i - \hat{y}_i$) vengono plottati in funzione del tempo. Un modello ideale dovrebbe produrre residui distribuiti casualmente attorno allo zero, senza pattern evidenti. La presenza di pattern (es. ciclicità, trend crescenti/decrescenti) indica che il modello non sta catturando completamente una componente della dinamica dei dati.

- **Scatter Plot Reali vs. Predetti:** Questo grafico mostra i valori reali sull'asse delle ascisse e i valori predetti sull'asse delle ordinate. In uno scenario di previsione perfetta, tutti i punti si allineerebbero sulla bisettrice del primo quadrante (la linea $y = x$). La dispersione dei punti attorno a questa linea fornisce un'idea della correlazione tra predizioni e realtà e può rivelare la presenza di *bias* sistematici (es. tendenza a sottostimare o sovrastimare i valori).
- **Errore Relativo Medio:** Calcolato come la media degli errori assoluti divisi per il valore reale corrispondente ($\frac{|y_i - \hat{y}_i|}{|y_i|}$), questo indicatore esprime l'errore in termini percentuali, rendendolo indipendente dalla scala dei dati e facilmente comprensibile.

4.3 Risultati LSTM

Il primo modello sottoposto ad analisi è la rete *Long Short-Term Memory* (LSTM), un'architettura ricorrente consolidata, ampiamente utilizzata come *benchmark* per problemi di *forecasting* su serie storiche. La sua capacità di mantenere una memoria a lungo termine la rende teoricamente adatta a catturare le dipendenze temporali complesse presenti nel dataset. Prima di alimentare il modello, è stata applicata una fase di *ingegneria delle feature*, espandendo il set di dati da 16 a 48 *feature* per arricchire l'informazione disponibile per l'apprendimento. Di seguito vengono presentati i risultati ottenuti attraverso la procedura di *cross-validation* a 5 *fold*.

Andamento della loss

L'analisi del processo di apprendimento è fondamentale per comprendere il comportamento del modello. La Figura 4.1 mostra l'andamento della funzione di costo (*Mean Squared Error*) sulle epoche di addestramento per i dati di *training* e di validazione, relativi all'ultimo *fold* della *cross-validation*.

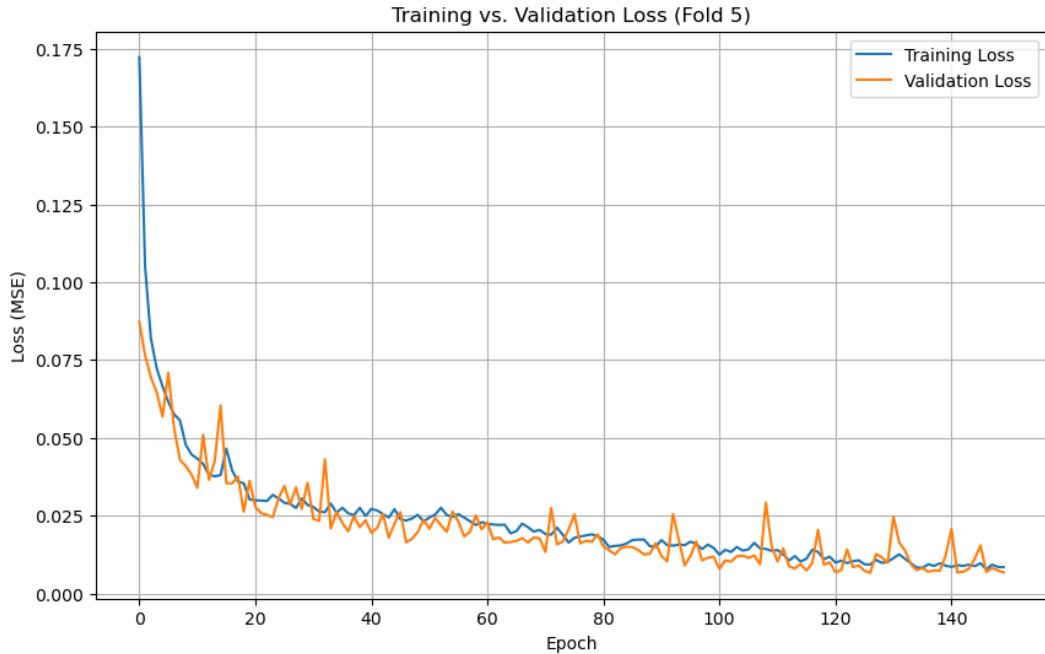


Figura 4.1: Andamento della loss di training e di validazione del modello LSTM nell'ultimo fold.

Dall'osservazione del grafico emergono due dinamiche principali. In primo luogo, **entrambe le curve mostrano una rapida e decisa diminuzione** nelle fasi iniziali del *training*, a testimonianza del fatto che il modello sta apprendendo efficacemente le correlazioni presenti nei dati. La curva di *training loss* continua a decrescere in modo monotono per tutta la durata dell'addestramento. In secondo luogo, la curva di *validation loss*, dopo aver seguito fedelmente quella di *training*, tende a stabilizzarsi e a mostrare una maggiore volatilità nelle

epoches finali. Questo leggero scostamento tra le due curve è un indicatore dell'insorgere di un **lieve overfitting**: il modello comincia a memorizzare le specificità del set di *training* a discapito della sua capacità di generalizzazione su dati mai visti. Questo comportamento suggerisce che l'uso di tecniche come l'*early stopping*, implementato nella procedura, è stato cruciale per individuare il modello con le migliori performance sul *validation set*, evitando un degrado delle prestazioni.

Analisi Qualitativa delle previsioni

Per una valutazione qualitativa della bontà delle previsioni, la Figura 4.2 sovrappone la serie storica dei valori reali (in blu) con le previsioni generate dal modello LSTM (in arancione) su tutti i *fold* di test concatenati.

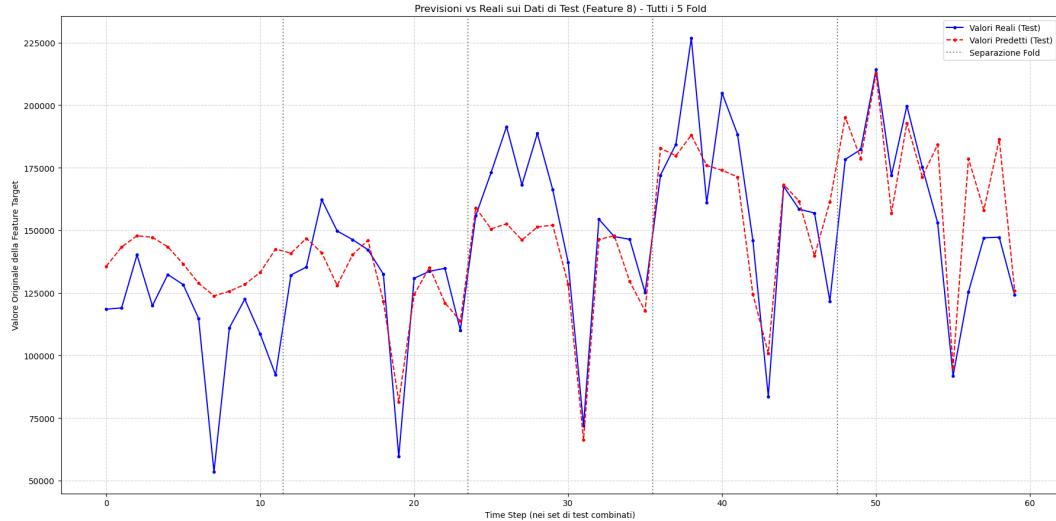


Figura 4.2: Confronto tra i valori reali e i valori predetti dal modello LSTM su tutti i fold di test.

L'ispezione visiva rivela che il modello LSTM è in grado di **catturare la dinamica generale e il trend** della serie storica. Le previsioni seguono l'andamento complessivo dei dati reali, riconoscendo i periodi di crescita e decrescita. Tuttavia, emerge con chiarezza una limitazione significativa: **il modello tende a smorzare la volatilità dei dati**. Nello specifico, si osserva una **sistematica sottostima dei picchi più elevati e una sovrastima delle valli più profonde**. Il modello, in sostanza, produce una versione "appiattita" e meno volatile della serie reale, faticando a replicare l'esatta ampiezza delle fluttuazioni più estreme. Questo comportamento è tipico di modelli che apprendono bene la tendenza centrale dei dati ma non riescono a modellizzare completamente le dinamiche non lineari che governano i movimenti di mercato più bruschi.

Analisi degli errori

Un'analisi approfondita dei residui (la differenza tra valori reali e predetti) fornisce ulteriori dettagli sulle carenze del modello. La Figura 4.3 mostra l'andamento dei residui nel tempo, mentre la Figura 4.4 presenta lo *scatter plot* dei valori reali contro quelli predetti.

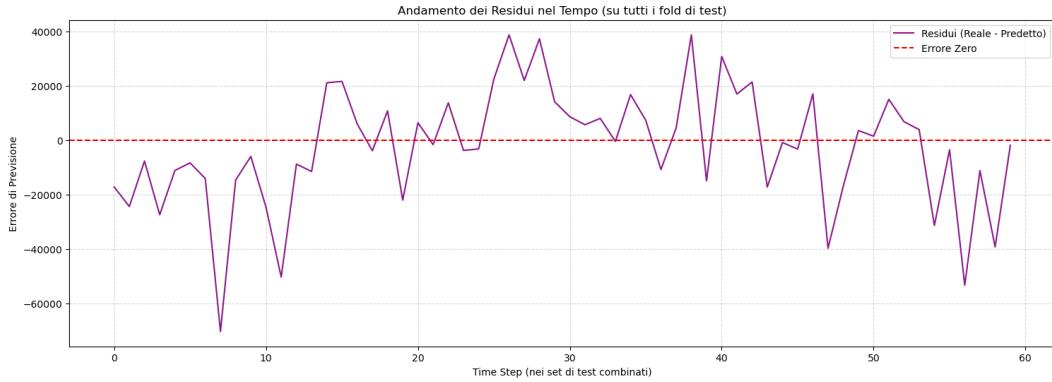


Figura 4.3: Andamento dei residui del modello LSTM nel tempo.

Il grafico dei residui (Figura 4.3) mostra che, sebbene gli errori siano mediamente centrati intorno allo zero, **la loro distribuzione non appare casuale**. Si osservano infatti fasi di relativa stabilità intervallate da periodi di elevata volatilità, un comportamento noto come *volatility clustering* o *varianza non costante*. Questo pattern suggerisce che **le prestazioni del modello non sono uniformi nel tempo**, ma tendono a peggiorare proprio nei momenti di maggiore instabilità della serie storica, ovvero quando una previsione accurata risulterebbe particolarmente importante.

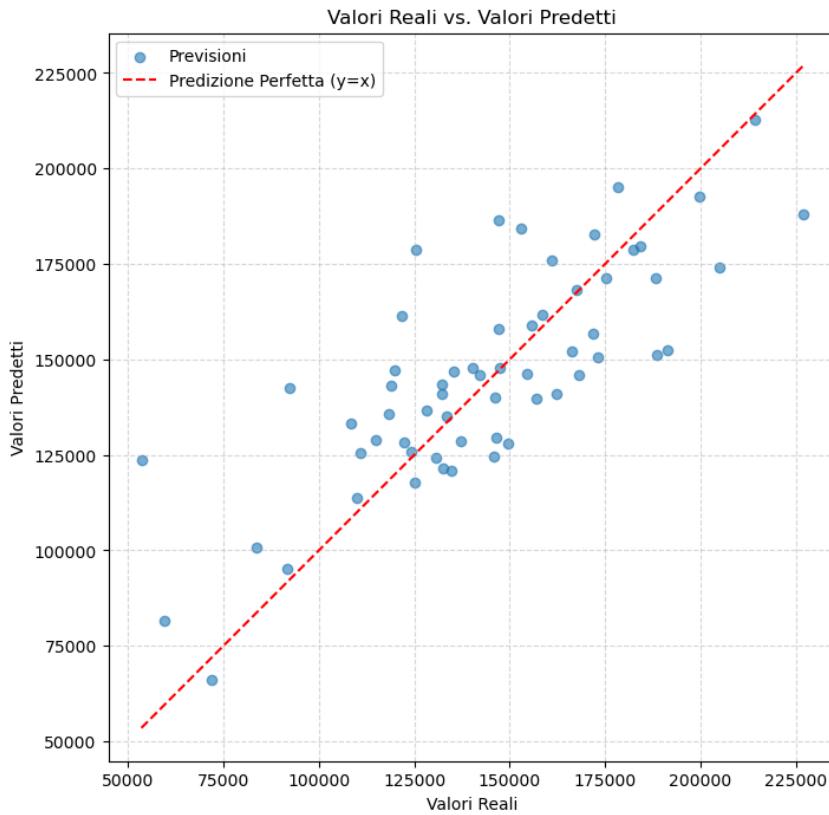


Figura 4.4: Scatter plot dei valori reali vs. i valori predetti dal modello LSTM.

Lo *scatter plot* (Figura 4.4) rafforza questa interpretazione. I punti si dispongono lungo la bisettrice $y = x$, indicando una correlazione positiva tra predizioni e realtà. Tuttavia, la nuvola di punti è piuttosto dispersa, a significare una notevole varianza dell'errore. Inoltre, si può osservare come per i valori reali più alti (asse x), i punti tendano a trovarsi al di sotto della linea di previsione perfetta. Ciò **conferma visivamente la tendenza del modello a sottostimare i picchi**, come già notato nell'analisi qualitativa.

Risultati Quantitativi

L'analisi quantitativa, basata sulle metriche di errore aggregate tramite *cross-validation*, permette di sintetizzare numericamente le performance del modello. La Tabella 4.1 riassume i valori di MSE, RMSE e MAE per ciascuno dei 5 *fold*, insieme alla media e alla deviazione standard finali.

Tabella 4.1: Risultati delle metriche di performance del modello LSTM su 5-Fold Cross-Validation.

Fold	MSE	RMSE	MAE
1	866.906.545,04	29.443,28	22.953,47
2	168.991.089,88	12.999,66	10.944,86
3	385.231.966,03	19.627,33	15.438,84
4	478.116.230,93	21.865,87	18.021,97
5	506.669.910,55	22.509,33	15.670,71
Media	481.183.148,49	21.935,89	16.605,97
Dev. Std.	253.037.561,71	5.911,61	4.375,52

I risultati aggregati mostrano un *Root Mean Squared Error* (RMSE) Medio di **21.935,89** e un *Mean Absolute Error* (MAE) Medio di **16.605,97**. L'Errore Relativo Medio calcolato sull'intero set di previsioni si attesta al **13,57%**.

Un dato significativo è l'**elevata deviazione standard**, in particolare per le metriche quadratiche (RMSE e MSE). Questo valore riflette una notevole variabilità nelle performance del modello tra i diversi *fold*, con prestazioni decisamente migliori nel Fold 2 (RMSE di circa 12.999) e peggiori nel Fold 1 (RMSE di circa 29.443). Tale instabilità conferma che la capacità predittiva dell'LSTM è fortemente dipendente dalla specifica porzione di dati su cui viene testata, allineandosi con l'osservazione di *eteroschedasticità* emersa dall'analisi dei residui. In sintesi, pur essendo un solido punto di partenza, il modello LSTM mostra chiari limiti nella gestione della volatilità e nella stabilità delle sue prestazioni.

4.4 Risultati Transformer Encoder-Decoder

In questa sezione vengono analizzate le performance del modello *Transformer Encoder-Decoder*, un'architettura stato dell'arte basata sul meccanismo di attenzione. A differenza dell'approccio ricorrente dell'LSTM, questo modello processa simultaneamente l'intera sequenza di input attraverso il suo *encoder*, creando una ricca rappresentazione contestuale. Tale rappresentazione viene poi utilizzata dal *decoder* per generare la sequenza di previsione. Come specificato nella metodologia, questo modello è stato addestrato utilizzando una funzione di costo *Weighted Mean Squared Error* (Weighted MSE) per porre maggiore enfasi sull'accuratezza delle previsioni a lungo termine.

Andamento della loss

Il processo di apprendimento del modello *Transformer* è illustrato nella Figura 4.5, che mostra le curve di *loss* di *training* e di validazione relative all'ultimo *fold* di validazione.

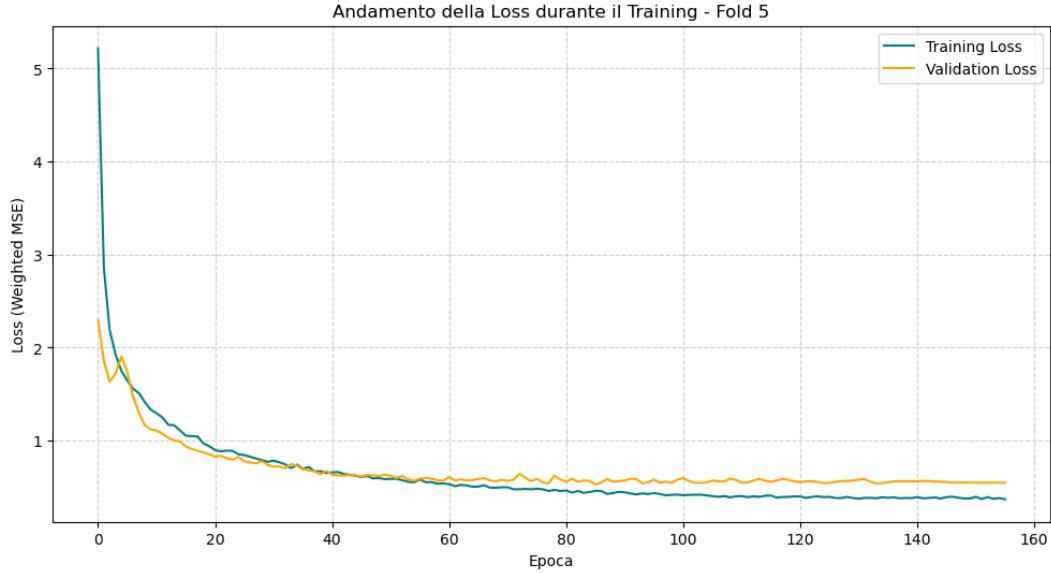


Figura 4.5: Andamento della loss di training e di validazione (Weighted MSE) del modello Transformer Encoder-Decoder nell'ultimo fold.

Il grafico mostra chiaramente che il modello apprende in modo efficace, con una diminuzione costante sia della *loss* di *training* che di quella di validazione. Il meccanismo di *early stopping* è intervenuto in modo consistente tra i vari *fold*, dimostrando la sua efficacia nel prevenire l'*overfitting*. La curva di *validation loss* segue da vicino quella di *training* senza divergenze significative, indicando che **il modello generalizza bene su dati mai visti**. Questo comportamento stabile di apprendimento suggerisce che l'architettura è ben adatta alla complessità del dataset.

Analisi Qualitativa delle previsioni

La Figura 4.6 offre un confronto visivo tra i dati reali (in blu) e le previsioni effettuate dal modello *Transformer* (in arancione) su tutti i *fold* di test.

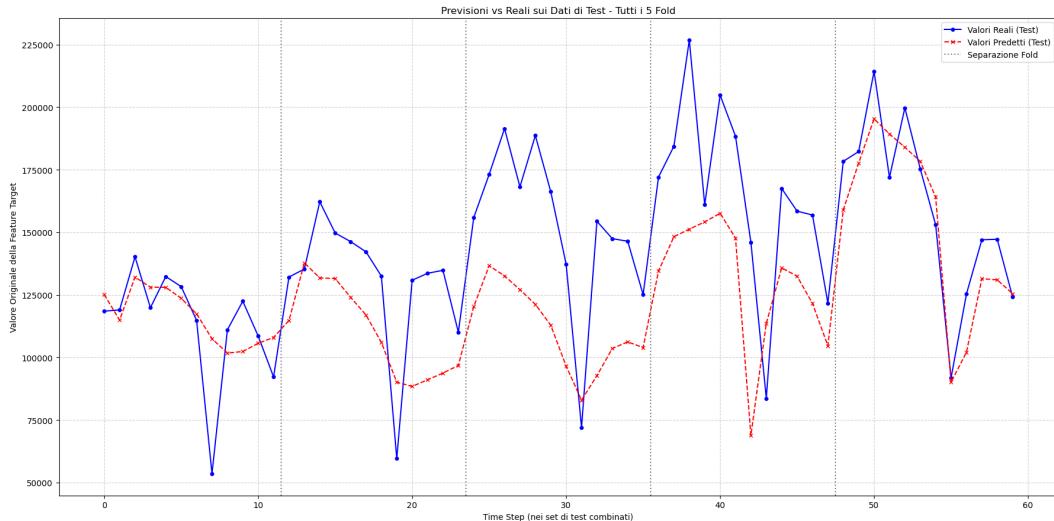


Figura 4.6: Confronto tra i valori reali e i valori predetti dal modello Transformer Encoder-Decoder su tutti i fold di test.

A un primo sguardo, il modello sembra catturare il trend generale e le principali fluttuazioni della serie storica. Tuttavia, un'analisi più attenta rivela un comportamento simile a quello dell'LSTM: **il modello tende a smussare le previsioni, sottostimando l'ampiezza dei picchi e delle valli più estremi**. Questa tendenza è particolarmente evidente nei periodi

di alta volatilità. Sebbene il modello identifichi correttamente la direzione dei principali movimenti di prezzo, **non riesce a replicarne la piena ampiezza**, risultando in una previsione meno dinamica rispetto alla serie reale.

Analisi degli errori

Per comprendere meglio i limiti del modello, sono stati analizzati i suoi residui. La Figura 4.7 mostra l'andamento dei residui nel tempo, mentre la Figura 4.8 presenta lo *scatter plot* dei valori reali rispetto a quelli predetti.

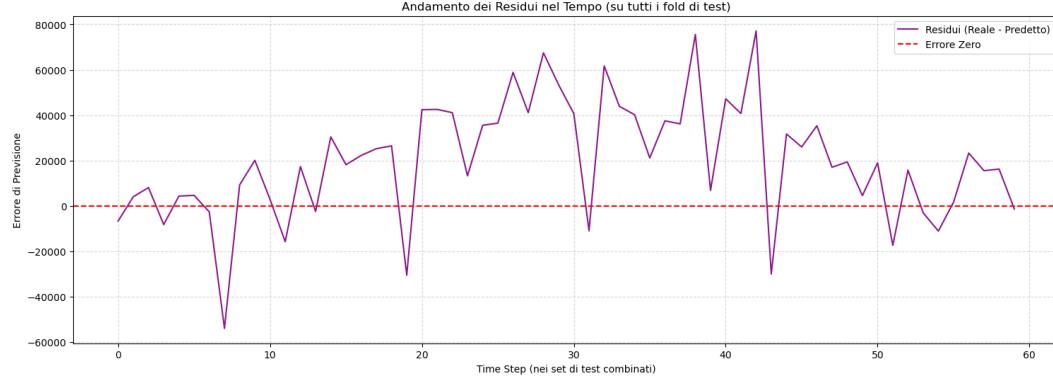


Figura 4.7: Andamento dei residui del modello Transformer Encoder-Decoder nel tempo.

Il grafico dei residui (Figura 4.7) mostra che gli errori **non sono distribuiti casualmente** attorno allo zero. Si osservano, al contrario, chiari *cluster* di alta volatilità, indicando che l'accuratezza del modello varia in modo significativo nel tempo. Questa **varianza non costante** suggerisce che il modello fatica a mantenere performance costanti, specialmente durante le fasi di mercato turbolente, dove gli errori di previsione tendono a essere maggiori.

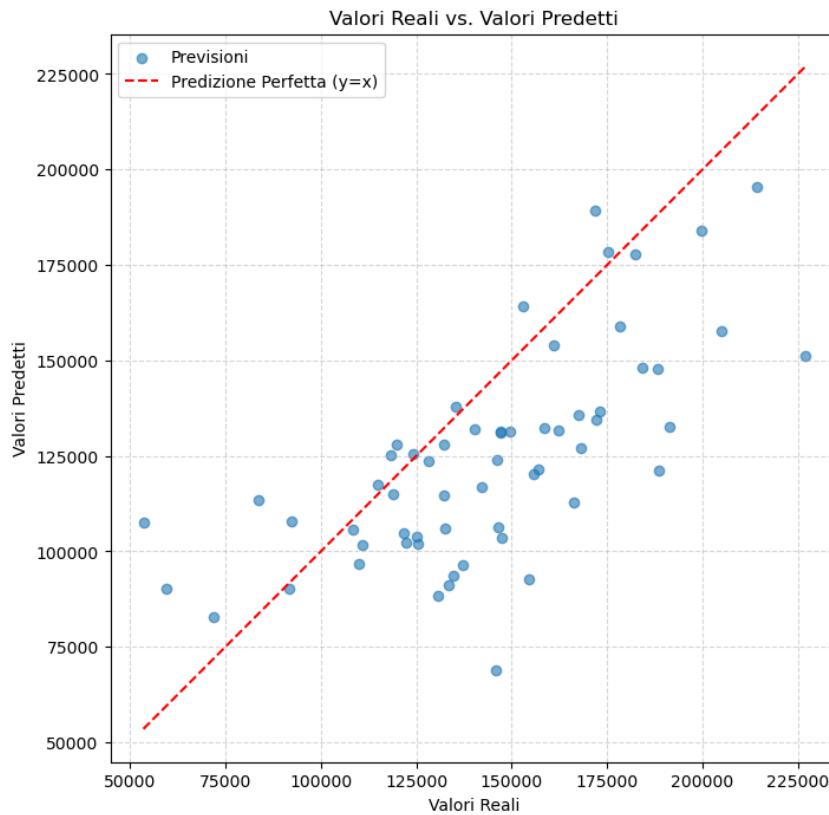


Figura 4.8: Scatter plot dei valori reali vs. i valori predetti dal modello Transformer Encoder-Decoder.

Lo *scatter plot* (Figura 4.8) conferma queste osservazioni. I punti formano una nuvola allineata lungo la linea ideale $y = x$, indicando una buona correlazione generale. Tuttavia, **la nuvola è ampia, confermando l'elevata varianza degli errori**. Come per l'LSTM, è visibile una tendenza a sottostimare i valori più alti, con molti punti che cadono al di sotto della diagonale per valori reali elevati. Ciò conferma che il modello è più conservativo nelle sue previsioni, non riuscendo a cogliere l'intera portata dei movimenti al rialzo più significativi.

Risultati Quantitativi

Le performance quantitative del modello sono riassunte nella Tabella 4.2, che riporta le metriche MSE, RMSE e MAE per ciascuno dei 5 *fold* della *cross-validation*, insieme alla loro media e deviazione standard finali.

Tabella 4.2: Risultati delle metriche di performance del modello Transformer Encoder-Decoder su 5-Fold Cross-Validation.

Fold	MSE	RMSE	MAE
1	324.831.392,00	18.023,08	11.714,96
2	816.353.792,00	28.571,91	26.008,81
3	2.058.835.456,00	45.374,39	42.604,43
4	1.867.456.384,00	43.214,08	38.419,82
5	207.618.624,00	14.408,98	12.357,65
Media	1.055.019.129,60	32.481,06	26.221,13
Dev. Std.	862.543.390,49	14.137,86	14.316,87

I risultati finali mostrano un *Root Mean Squared Error* (RMSE) Medio di **32.481,06** e un *Mean Absolute Error* (MAE) Medio di **26.221,13**. L'Errore Relativo Medio per questo modello si attesta al **18,92%**.

Un'osservazione chiave è la **deviazione standard estremamente elevata** per tutte le metriche. Questo valore evidenzia una **significativa instabilità delle performance** tra i diversi *fold*. Il modello ha performato in modo eccezionale nel Fold 5 ($\text{RMSE} \approx 14.409$) ma molto male nei Fold 3 e 4 ($\text{RMSE} > 43.000$). Questa ampia variabilità suggerisce che il modello *Transformer* è **altamente sensibile allo specifico segmento di dati** su cui viene addestrato e testato. Nonostante la sua sofisticazione architettonale, questa instabilità rappresenta una limitazione fondamentale, rendendo discutibile la sua affidabilità predittiva in uno scenario reale.

4.5 Risultati Transformer Decoder

In questa sezione viene analizzata l'architettura *Transformer Decoder-only*. A differenza del modello *Encoder-Decoder* completo, questa variante opera in modo puramente *autoregressivo*, simile concettualmente a un modello linguistico generativo. Utilizza un meccanismo di *masked self-attention* per considerare solo i punti temporali passati durante la generazione di ogni *step* futuro della previsione. Questa architettura è spesso più leggera e veloce da addestrare rispetto alla sua controparte completa. Il modello è stato ottimizzato utilizzando la funzione di costo standard *Mean Squared Error* (MSE).

Andamento della loss

La Figura 4.9 illustra il comportamento del modello durante la fase di apprendimento, mostrando le curve di *loss* di *training* e di validazione per l'ultimo *fold* della *cross-validation*.

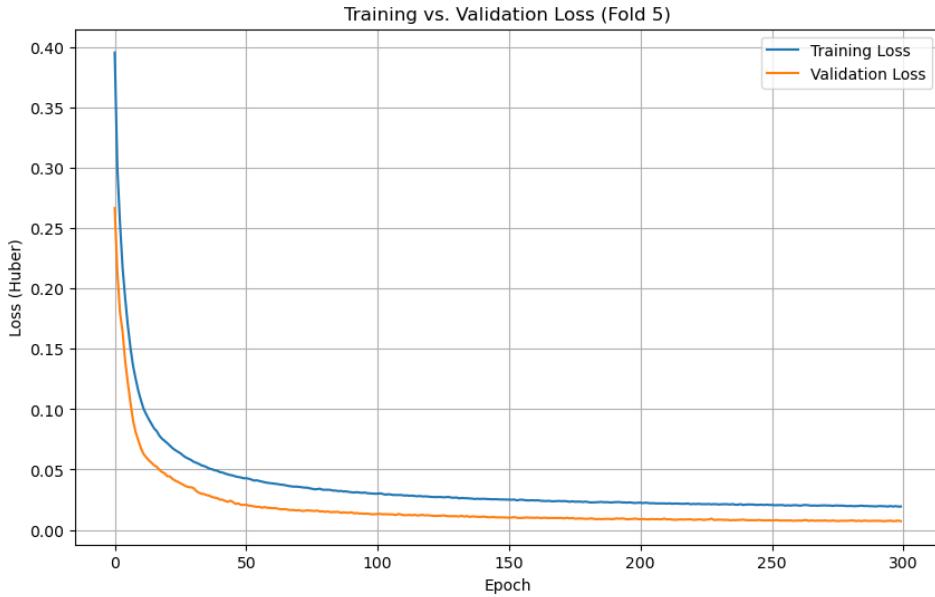


Figura 4.9: Andamento della loss di training e di validazione del modello Transformer Decoder-only nell'ultimo fold.

Il grafico mostra un processo di apprendimento **estremamente stabile ed efficiente**. Entrambe le curve di *loss* convergono rapidamente verso valori molto bassi, e la curva di validazione segue quella di *training* in modo quasi perfetto, **senza mostrare alcun segno di overfitting**. Questo parallelismo indica un'**eccellente capacità di generalizzazione** del modello. La stabilità del processo di *training* suggerisce che l'architettura *Decoder-only* è particolarmente adatta alla struttura dei dati analizzati, riuscendo a catturare le dinamiche rilevanti senza memorizzare il rumore di fondo.

Analisi Qualitativa delle previsioni

La Figura 4.10 offre un confronto visivo diretto tra i dati reali (in blu) e le previsioni generate dal modello (in arancione) per tutti i *fold* di test.

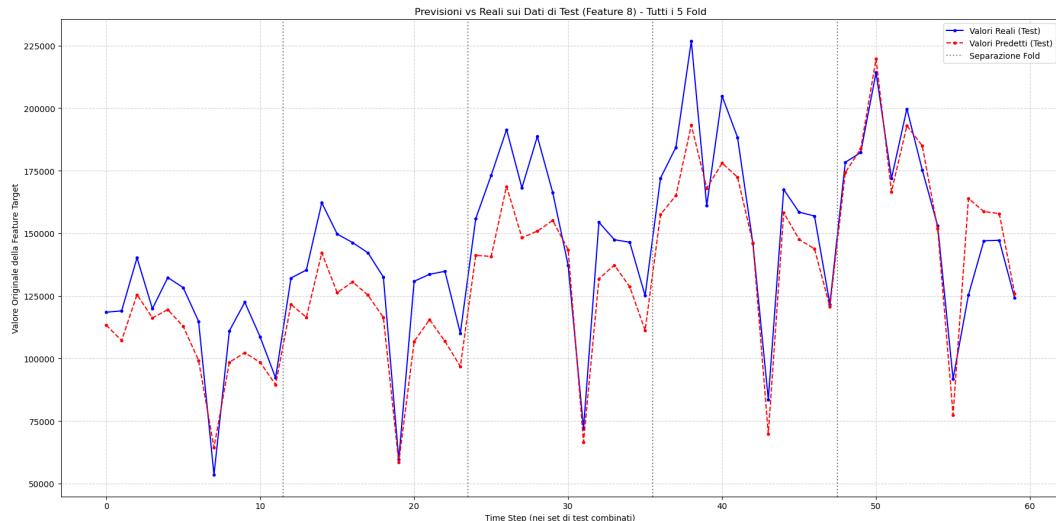


Figura 4.10: Confronto tra i valori reali e i valori predetti dal modello Transformer Decoder-only su tutti i fold di test.

L'analisi qualitativa delle previsioni segna un **netto miglioramento rispetto ai modelli precedenti**. Il grafico mostra una **straordinaria aderenza** delle previsioni ai valori reali. A differenza dei modelli LSTM e *Encoder-Decoder*, questa architettura non si limita a cat-

turare il trend generale, ma riesce a **replicare con notevole precisione l'ampiezza e la forma dei picchi e delle valli**. La tendenza a "smorzare" la volatilità è quasi del tutto assente. Questa capacità di modellizzare le fluttuazioni più estreme rappresenta un punto di forza cruciale, indicando una comprensione più profonda delle dinamiche non lineari della serie storica.

Analisi degli errori

Per confermare le osservazioni qualitative, si procede con l'analisi dei residui (Figura 4.11) e dello *scatter plot* (Figura 4.12).

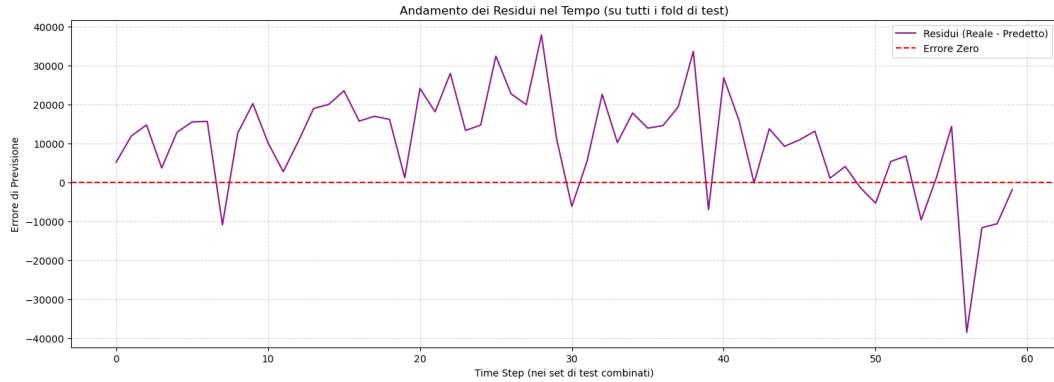


Figura 4.11: Andamento dei residui del modello Transformer Decoder-only nel tempo.

Il grafico dei residui (Figura 4.11) mostra **errori di entità significativamente inferiore** rispetto ai modelli precedenti. Sebbene permanga una leggera *varianza non costante* (la varianza degli errori non è perfettamente costante), i *cluster* di errore sono molto meno pronunciati. I residui fluttuano in una banda più stretta attorno allo zero, a conferma della maggiore precisione del modello su tutto l'arco temporale.

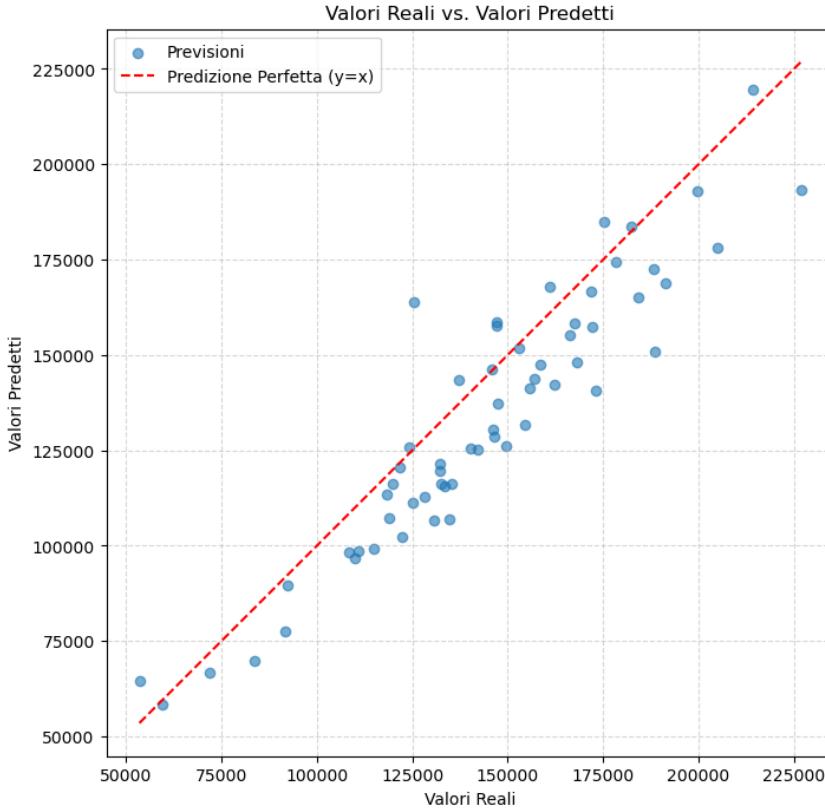


Figura 4.12: Scatter plot dei valori reali vs. i valori predetti dal modello Transformer Decoder-only.

Lo *scatter plot* (Figura 4.12) è la prova più evidente delle eccellenti performance del modello. I punti si dispongono in una **nube stretta e densa attorno alla bisettrice $y = x$** , indicando una **correlazione quasi perfetta** tra valori predetti e reali. La dispersione è minima, e non si notano troppo i *bias* sistematici come la sottostima dei valori più alti. Questo grafico conferma che il modello non solo è accurato in media, ma è anche affidabile su tutto il *range* di valori della serie.

Risultati Quantitativi

L’analisi numerica, riassunta nella Tabella 4.3, consolida le impressioni positive emerse dall’analisi qualitativa. La tabella riporta le metriche di errore per ogni *fold* e i valori aggregati finali.

Tabella 4.3: Risultati delle metriche di performance del modello Transformer Decoder-only su 5-Fold Cross-Validation.

Fold	MSE	RMSE	MAE
1	153.259.632,00	12.379,81	11.316,51
2	339.675.808,00	18.430,30	17.182,13
3	409.609.056,00	20.238,80	17.877,13
4	275.201.184,00	16.589,19	13.786,97
5	180.010.864,00	13.416,81	9.230,78
Media	271.551.308,80	16.478,81	13.878,70
Dev. Std.	107.339.163,84	3.308,21	3.710,68

I risultati quantitativi sono eccellenti. Il modello raggiunge un *Root Mean Squared Error* (RMSE) Medio di **16.478,81** e un *Mean Absolute Error* (MAE) Medio di **13.878,70**. L’Errore Relativo Medio è il più basso registrato finora, attestandosi al **9,83%**. Oltre ai valori medi nettamente migliorati, è fondamentale notare la **drastica riduzione della deviazione standard** (RMSE Std. di 3.308,21) rispetto ai modelli precedenti. Que-

sto indica che le performance del *Transformer Decoder-only* non sono solo superiori, ma anche **significativamente più stabili e consistenti** attraverso i diversi *fold* di validazione. L'architettura ha dimostrato di essere robusta, affidabile e precisa, posizionandosi come la soluzione più performante tra quelle addestrate specificamente su questo dataset.

4.6 Risultati Transformer Chronos

L'ultima architettura analizzata è *Chronos*, un *foundation model* basato su *Transformer* e pre-addestrato da Amazon su un vasto corpus di serie storiche. A differenza dei modelli precedenti, *Chronos* è stato impiegato in modalità *zero-shot*, ovvero senza alcuna fase di *training* o *fine-tuning* sul nostro dataset specifico. L'obiettivo di questo esperimento non è ottimizzare le performance, ma valutare la **capacità di generalizzazione "a freddo"** di un modello su larga scala. Di conseguenza, non essendo presente una fase di addestramento, non verrà analizzata la curva di *loss*.

Analisi Qualitativa delle previsioni

La Figura 4.13 mette a confronto le previsioni generate da *Chronos* con i dati reali. Questo grafico è essenziale per comprendere come le conoscenze generalizzate del modello si traducono sul nostro specifico problema di *forecasting*.

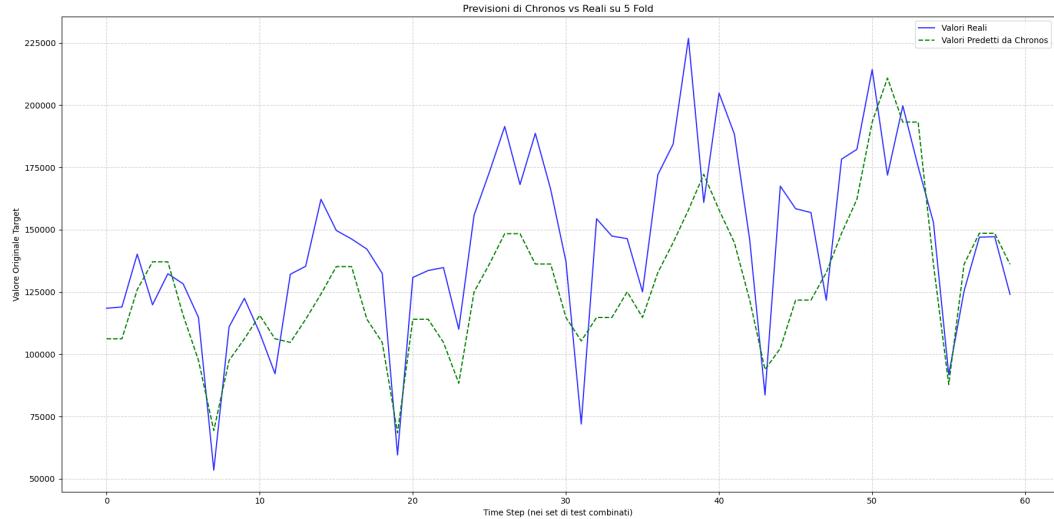


Figura 4.13: Confronto tra i valori reali e i valori predetti dal modello Chronos (zero-shot) su tutti i fold di test.

L'analisi visiva mostra che il modello *Chronos* cattura la direzionalità di base della serie storica, ma con significative imprecisioni. Le previsioni appaiono spesso **in ritardo (lag)** rispetto ai dati reali, quasi come se seguissero la serie con uno sfasamento temporale. Inoltre, si rileva una marcata tendenza a produrre una versione **smorzata e liscia** della serie originale. La volatilità, i picchi e le valli vengono appiattiti di molto, indicando che il modello non è del tutto in grado di cogliere le dinamiche specifiche e la magnitudo delle fluttuazioni del nostro dataset. Comunque, la previsione risulta non essere casuale.

Analisi degli errori

Le Figure 4.14 e 4.15 forniscono un'analisi più approfondita degli errori commessi dal modello.

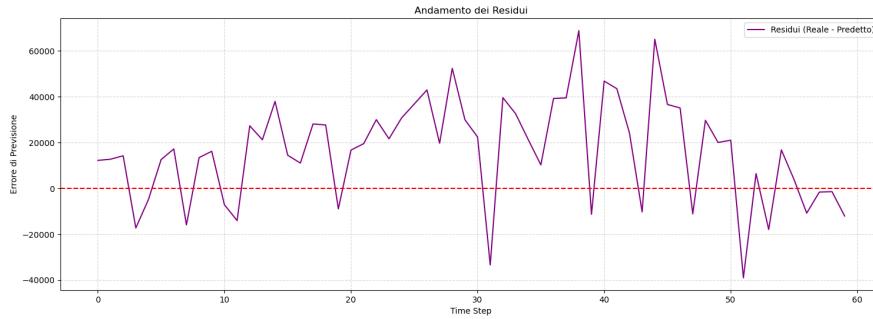


Figura 4.14: Andamento dei residui del modello Chronos nel tempo.

Il grafico dei residui (Figura 4.14) evidenzia l'entità degli errori di previsione. I residui sono di magnitudo considerevole e, soprattutto, **non sono distribuiti casualmente**. Si osservano lunghi periodi in cui il modello sovrasta o sottosta sistematicamente i valori reali, generando pattern di errore chiaramente definiti. Questa **forte autocorrelazione nei residui** è un sintomo del fatto che il modello non sta catturando componenti fondamentali della serie storica, come la stagionalità o il ciclo specifico dei dati analizzati.

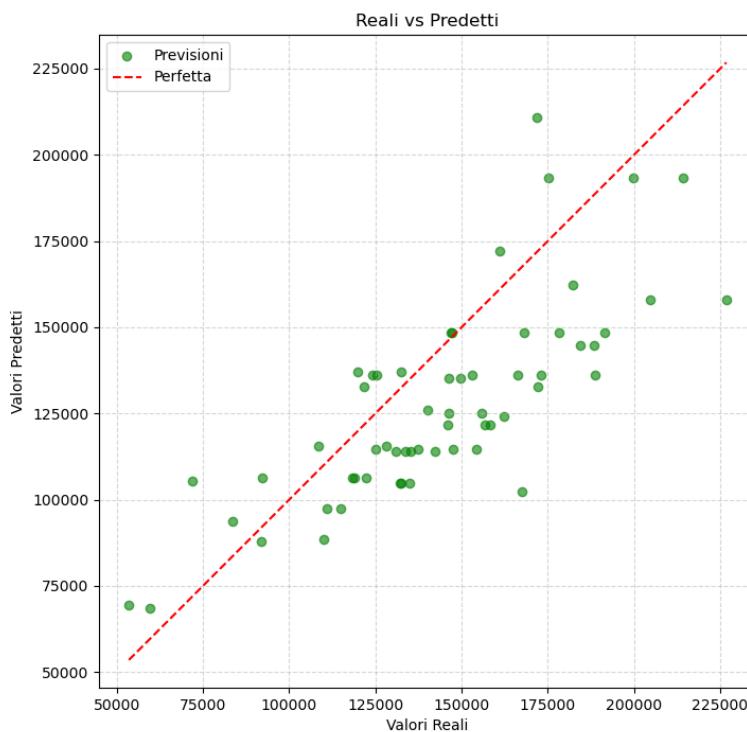


Figura 4.15: Scatter plot dei valori reali vs. i valori predetti dal modello Chronos.

Lo *scatter plot* (Figura 4.15) conferma la debolezza della performance. La nuvola di punti è **dispersa** e solo vagamente allineata sotto la linea della bisettrice $y = x$, indicando una **correlazione molto debole** tra previsioni e valori reali. La dispersione è sinonimo di un'elevata varianza dell'errore e di una bassa precisione. Questo risultato è coerente con un modello che applica una conoscenza generica a un problema specifico, riuscendo a fatica ad adattarsi alle sue peculiarità.

Risultati Quantitativi

I risultati numerici, presentati in Tabella 4.4, quantificano le osservazioni emerse dall’analisi qualitativa.

Tabella 4.4: Risultati delle metriche di performance del modello Chronos (zero-shot) su 5-Fold Cross-Validation.

Fold	MSE	RMSE	MAE
1	186.620.016,00	13.660,89	13.161,57
2	554.673.728,00	23.551,51	22.089,11
3	1.084.737.664,00	32.935,36	31.055,30
4	1.641.863.808,00	40.519,92	35.983,86
5	348.307.936,00	18.663,01	15.067,42
Media	763.240.640,00	27.626,81	23.471,45
Dev. Std.	596.552.524,39	10.850,32	9.910,95

I risultati aggregati mostrano un *Root Mean Squared Error* (RMSE) Medio di **27.626,81** e un *Mean Absolute Error* (MAE) Medio di **23.471,45**. L’Errore Relativo Medio si attesta al **16,12%**.

Il dato allarmante è l’alta **deviazione standard** (RMSE Std. di 10.850.32), che indica una performance **volatile e inaffidabile** tra i diversi *fold*. Il modello si comporta in modo passabile sul primo *fold*, ma le sue prestazioni degradano molto nel quarto, con un errore quasi triplicato. Questa instabilità dimostra che la conoscenza generalizzata di *Chronos* non è uniformemente applicabile a tutte le porzioni del nostro dataset.

4.7 Analisi Comparativa e Discussione

Dopo aver esaminato individualmente le performance di ciascuna architettura, questa sezione si propone di effettuare un’analisi comparativa diretta per identificare il modello più performante e discutere le implicazioni dei risultati ottenuti. Il confronto si basa sia sulle metriche quantitative aggregate sia sulle osservazioni qualitative emerse dall’analisi grafica.

4.7.1 Confronto Quantitativo

Per un confronto oggettivo, i risultati quantitativi medi ottenuti dalla *cross-validation* per tutti e quattro i modelli sono stati raccolti nella Tabella 4.5. Questa tabella riassuntiva permette di valutare a colpo d’occhio le performance relative di ciascun approccio in termini di accuratezza e stabilità.

Tabella 4.5: Tabella comparativa delle performance aggregate dei quattro modelli analizzati.

Modello	RMSE Medio	MAE Medio	Errore Rel. Medio	Dev. Std. (RMSE)
LSTM	21.935,89	16.605,97	13,57%	5.911,61
Transformer E-D	32.481,06	26.221,13	18,92%	14.137,86
Transformer D	16.478,81	13.878,70	9,83%	3.308,21
Chronos (Zero-Shot)	27.626,81	23.471,45	16,12%	10.850,32

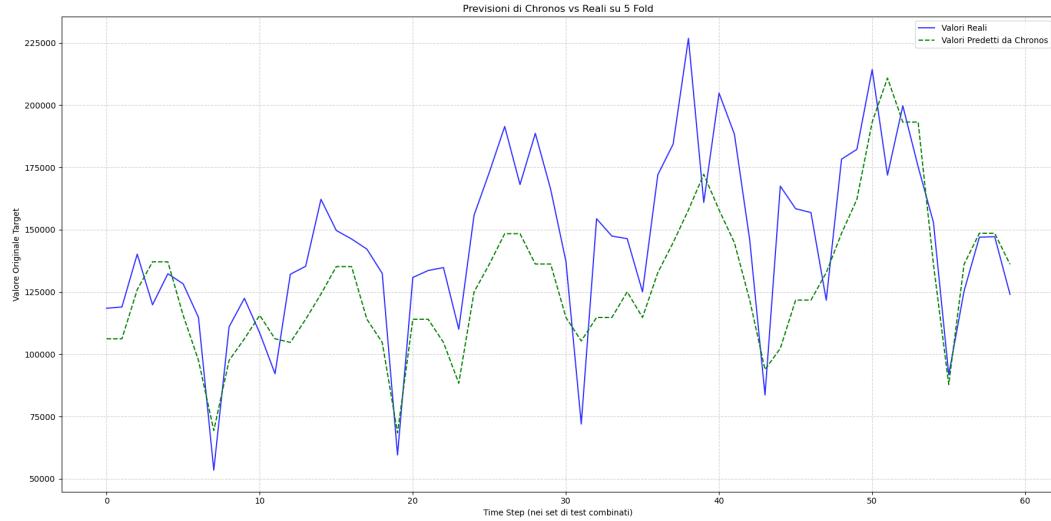


Figura 4.16: Grafico a barre comparativo delle metriche chiave (RMSE e MAE Medi) per i quattro modelli.

I dati parlano chiaro: il modello **Transformer solo Decoder** emerge come il vincitore di questo confronto. Supera tutte le altre architetture in ogni metrica di accuratezza, registrando l'RMSE, il MAE e l'Errore Relativo Medio più bassi.

Un aspetto altrettanto cruciale è la **stabilità**. Il *Transformer Decoder-only* non è solo il più preciso, ma anche il più affidabile, come dimostra la sua deviazione standard dell'RMSE, che è di gran lunga la più bassa del gruppo (3.308.21). Al contrario, il modello **Transformer Encoder-Decoder**, pur essendo architettonicamente più complesso, si è rivelato il peggiore, mostrando non solo errori medi più elevati ma, soprattutto, una **volatilità di performance molto bassa** (Dev. Std. di 14.137.86). Questo suggerisce che, per questo specifico dataset, la complessità aggiuntiva dell'*encoder* non ha portato benefici, anzi ha introdotto instabilità. Il modello **LSTM** si posiziona come un *benchmark* ragionevole, con performance intermedie, superando l'instabile *Transformer E-D* e il modello *zero-shot*, ma rimanendo significativamente distanziato dal vincitore. Infine, **Chronos**, pur essendo un *foundation model* avanzato, in modalità *zero-shot* si è dimostrato comunque adeguato nonostante non sia stato effettuato alcun fine tuning.

Capitolo 5

Conclusioni

Il presente lavoro di tesi è scaturito da una domanda fondamentale nel campo del *forecasting* di serie storiche: possono i *Transformer* rappresentare una metodologia più accurata rispetto alle tecniche tradizionalmente utilizzate, come ad esempio le LSTM (*Long Short-Term Memory*)?

Per rispondere a questo interrogativo, è stato condotto un esperimento comparativo rigoroso, mettendo a confronto non solo la consolidata architettura LSTM, ma anche due varianti di *Transformer* (*Encoder-Decoder* e *Decoder-only*) e un *foundation model* pre-addestrato (*Chronos*). È cruciale contestualizzare questa ricerca tenendo conto di una condizione operativa fondamentale: l'analisi è stata condotta su un **dataset di dimensioni relativamente ridotte**, composto da 349 campioni. Questa limitazione ha inevitabilmente influenzato il comportamento dei modelli, in particolare quelli con maggiore capacità e complessità.

Nonostante la scarsità dei dati, l'analisi ha fornito una risposta chiara e sorprendente. Il modello ***Transformer solo Decoder* si è imposto come l'architettura nettamente superiore**. Ha superato tutti gli altri concorrenti in termini di accuratezza, raggiungendo un Errore Relativo Medio del 9,83%, e soprattutto di stabilità, mostrando una varianza nelle performance tra i diversi *fold* di validazione drasticamente inferiore. Questa architettura ha dimostrato una capacità unica di modellizzare non solo il trend generale, ma anche la volatilità e l'ampiezza delle fluttuazioni, un aspetto cruciale dove gli altri modelli hanno mostrato evidenti limiti.

Quindi, alla domanda iniziale, la risposta è affermativa: **sì, un'architettura *Transformer* può essere più accurata di una LSTM**, ma con un'importante specificazione emersa da questo studio: la sua variante più snella e puramente autoregressiva si è dimostrata la più efficace in un contesto di dati limitati.

5.1 Contributi Originali

Il presente lavoro di tesi offre diversi contributi significativi al campo di studi:

1. **Confronto Empirico Diretto:** È stata condotta una comparazione sistematica di quattro diverse filosofie di modellizzazione (ricorrente, *encoder-decoder*, puramente autoregressiva e *zero-shot*) sullo stesso dataset e con la stessa metodologia di validazione.
2. **Validazione della Superiorità del *Decoder-only* (in contesti "data-scarce"):** Il risultato più rilevante è la dimostrazione che, con un dataset di dimensioni contenute, un'architettura *Transformer* più leggera (*Decoder-only*) ha superato la sua controparte più complessa (*Encoder-Decoder*). Questo *finding* è particolarmente prezioso perché suggerisce che la complessità aggiuntiva dell'*encoder*, in assenza di una grande mole di dati, non solo non porta benefici ma può addirittura introdurre instabilità e degradare le performance, probabilmente a causa di una maggiore propensione all'*overfitting*.
3. **Benchmark su *Foundation Models*:** L'inclusione di *Chronos* in modalità *zero-shot* ha evidenziato i limiti attuali dei modelli pre-addestrati quando applicati "a freddo" su dataset specifici e di piccole dimensioni, confermando che il *training* mirato rimane una strategia indispensabile per ottenere performance di alto livello in queste condizioni.

5.2 Limiti e Sviluppi Futuri

Ogni ricerca, per sua natura, apre la strada a nuove domande. È fondamentale riconoscere i limiti di questo studio per inquadrare correttamente i risultati.

Il limite principale è, senza dubbio, la **dimensione esigua del dataset** (349 campioni). Le conclusioni tratte sono valide all'interno di questo specifico contesto "*data-scarce*" e potrebbero non essere generalizzabili a scenari con grandi volumi di dati, dove modelli più complessi potrebbero avere l'opportunità di apprendere pattern più ricchi e performare meglio.

Partendo da queste considerazioni, si delineano diversi promettenti sviluppi futuri:

1. **Validazione su Larga Scala:** Il passo successivo più logico sarebbe replicare l'esperimento su un dataset di serie storiche significativamente più grande. Questo permetterebbe di verificare se la superiorità del *Transformer Decoder-only* persiste o se, con più dati a disposizione, il modello *Encoder-Decoder* riesce a sfruttare la sua maggiore capacità per colmare il divario di performance.
2. **Test di Generalizzazione:** Sarebbe interessante applicare l'architettura *Decoder-only*, risultata vincitrice, a dataset di piccole dimensioni ma provenienti da domini differenti (es. dati metereologici, biomedici) per investigare se la sua efficacia in contesti "*data-scarce*" sia una caratteristica generalizzabile.
3. **Indagine sul Fine-Tuning:** L'esperimento con *Chronos* ha lasciato una domanda aperta: come si comporterebbe il modello se venisse sottoposto a un processo di *fine-tuning*? Un confronto tra il *Transformer Decoder-only* addestrato da zero e un *foundation model* finemente sintonizzato potrebbe offrire nuovi *insight* sul *trade-off* tra specializzazione e conoscenza pre-addestrata.

In conclusione, questo lavoro di tesi ha risposto affermativamente alla domanda di ricerca, dimostrando il potenziale dei *Transformer*. Tuttavia, ha anche aggiunto una sfumatura cruciale: nel mondo reale, dove i dati non sono sempre abbondanti, la **scelta dell'architettura più semplice ed efficiente può rivelarsi non solo una necessità pratica, ma la chiave per ottenere i risultati migliori**.

Bibliografia

- [1] George E. P. Box, Gwilym M. Jenkins, Gregory C. Reinsel, and Greta M. Ljung. *Time Series Analysis: Forecasting and Control*. Wiley, 5th edition, 2015.
- [2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008. Curran Associates, Inc., 2017.
- [3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [4] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115, 2021.
- [5] Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [6] Romeo Kienzler. Introducing deep learning and long-short term memory networks.
- [7] Matteo Ferrara. Lezione 9: Transformers. Materiale didattico, slide della lezione, 2024. Corso di Laurea in Ingegneria e Scienze Informatiche, A.A. 2023/2024.
- [8] akhaliq. Chronos: Learning the language of time series.
- [9] Rapporto annuale bce 2015. Consultato nel 2025.
- [10] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [11] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.
- [12] Analisi dei dati per le imprese industriali (6). gli alberi decisionali. Consultato nel 2025.
- [13] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.
- [14] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.
- [15] What is gradient boosting? Consultato nel 2025.
- [16] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

- [17] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.
- [18] Diego Giorgini Luigi di sSotto. Come un nuovo paradigma rivoluziona il natural language processing: Chatbot sempre più bravi a comprenderci?
- [19] Mehreen Saeed. A gentle introduction to positional encoding in transformer models, part 1.
- [20] Abdul Fatir Ansari and et al. Chronos: Learning the language of time series, 2024.
- [21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- [22] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21(140):1–67, 2020.