



**Università
degli Studi
di Ferrara**

**Corso di Laurea Magistrale in Ingegneria
Informatica e dell'Automazione – percorso
Intelligenza Artificiale**

**SVILUPPO DI UN APPLICATIVO AI-BASED
PER LA PREDIZIONE DELL'USURA DEL PIANO MARTIRE
DI MACCHINE CNC PER NESTING**

Relatore:
Marcello Bonfè

Laureando:
Filippo Matteini

Correlatore:
Denis Billi

Anno Accademico: 2024/2025

“Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.”

— *Yoshua Bengio, Ian Goodfellow & Aaron Courville, Deep Learning (2016)*

“With deep learning we can solve problems that were intractable until a few years ago everything from speech recognition to medical image analysis.”

— *Geoffrey Hinton*

“The real power of deep learning is its ability to discover intricate structure in large data sets automatically.”

— *Yann LeCun*

Indice

1. Introduzione

- 1.1 Contesto industriale
- 1.2 Storia dell'azienda
- 1.3 Servizi offerti
- 1.4 Caratteristiche della macchina
- 1.5 Problema dell'usura del piano martire
- 1.6 Obiettivi del progetto formativo
- 1.7 Contributi

2. Stato dell'arte

- 2.1 Manutenzione predittiva
- 2.2 Applicazione dell'IA nel settore manifatturiero
- 2.3 Tecniche di Deep Learning

3. Metodologia

- 3.1 Raccolta dati
- 3.2 Pre-processing
 - 3.2.1 Features engineering
 - 3.2.2 Imputazione dei valori mancanti
 - 3.2.3 Encoding
 - 3.2.4 Normalizzazione
 - 3.2.5 Stratified train/val/test: splitting
 - 3.2.6 Oversampling, data augmentation e masking zero-padding
- 3.3 Algoritmo, modello e logica di programmazione
 - 3.3.1 Ciclo di vita del piano martire
 - 3.3.2 Architettura del modello, compilazione e addestramento
 - 3.3.3 Funzioni di loss e metriche
 - 3.3.4 Predizioni e funzione di inferenza dinamica

4. Esperimenti e risultati

- 4.1 Risultati e prestazioni ottenute
- 4.2 Grafici delle metriche e risultati di inferenza

5. Fine-tuning e Transfer Learning

- 5.1 Fine-tuning “regression-only” vs “full”
- 5.2 Strategie di transfer learning
- 5.3 Valutazione delle prestazioni e risultati ottenuti
- 5.4 Grafici delle metriche

6. Deployment e integrazione “online”

- 6.1 Architettura del servizio ReST Flask API ed end-points
- 6.2 Implementazione su Maestro Active: Maestro AI

7. Discussione generale del modello

- 7.1 Vantaggi e limiti del modello
- 7.2 Robustezza e generalizzabilità
- 7.3 Confronto e implicazioni pratiche

8. Conclusioni e lavori futuri

- 8.1 Sintesi dei risultati e dei contributi
- 8.2 Possibili estensioni
- 8.3 Sviluppi futuri

9. Bibliografia

10. Ringraziamenti finali e dediche

1. Introduzione

1.1 Contesto Industriale

L'intelligenza artificiale applicata all'ambito industriale è in rapida evoluzione e sta giocando un ruolo fondamentale nella competitività strategica aziendale e nel suo relativo approccio alla produzione in scala. Decidere di aprire le porte a questo "nuovo mondo" offre, chiaramente, un vantaggio non indifferente in termini di automatizzazione del processo lavorativo e leadership globale nel settore. Proprio per questa ragione, si sta iniziando a parlare sempre di più di industria 4.0, ovvero una digitalizzazione ordinaria di tutte le varie sfumature del settore secondario e meccanizzazione di tutti i processi operativi.

È possibile osservare, soprattutto, questo fenomeno nelle grandi aziende e multinazionali quali, in particolare, SCM Group S.p.A., più in dettaglio SCM... compagnia italiana - originaria della Provincia di Rimini (RN) in Emilia-Romagna, di notevoli dimensioni - specializzata nella produzione di macchine CNC a controllo numerico per la lavorazione secondaria del legno. La lavorazione secondaria è quel tassello di industria che trasforma il materiale grezzo in prodotto finito, come mobili, infissi, pavimenti e molto altro. È un settore antico ma da sempre dinamico, che si adatta alle nuove tendenze del design e alle esigenze di un mercato sempre più rigido in termini di qualità e personalizzazione. Alcuni esempi ricorrenti di prodotti derivati da tale tipologia di attività possono essere: tutti i tipi di mobili in legno, carpenteria, elementi di materiale leggero per qualunque tipo di rifinitura o base, infissi, piccole e grandi coperture di qualunque spazio e molti altri.

Le domande più ovvie con cui interrogarsi sono: quali sono i passi fondamentali da compiere in un'attività di lavorazione secondaria, come quella di SCM, e quali sono le migliori macchine da impiegare in tale business per avere successo e mantenere alti i profitti nonostante la competizione? Rispondendo alla prima domanda, si può dire: tutto ciò che riguarda la modellazione diretta del materiale grezzo, ovvero processi di: foratura, fresatura, taglio, bordatura, spianatura, sezionatura, scanalatura, profilatura, svuotatura, rifinitura della superficie e degli angoli e di modellazione di geometrie varie. Questi processi sono realizzabili, grazie ad utensili appositamente implementati sulle macchine stesse, come: trapani con punte di diverso tipo e diametro, seghe circolari, lame, frese a punta cava, tonda, piatta, triangolare ecc. Per rispondere invece alla seconda domanda, si può affermare che: siccome le operazioni e i movimenti degli utensili installati per la modellazione del legno mettono a rischio l'incolumità dell'operatore azionante, l'avvento di **macchine a controllo numerico (CNC)** è servito, in maniera assidua, per rivoluzionare questo settore, introducendo un livello di precisione, sicurezza, efficienza e flessibilità nelle lavorazioni mai visto prima.

Infatti, le CNC hanno permesso e permettono di eseguire lavorazioni con una precisione millimetrica, realizzando pezzi con tolleranze molto strette e finiture superficiali di altissima qualità. Questo è fondamentale per la produzione di mobili di design, infissi su misura e componenti per l'industria automobilistica o aerospaziale, dove la precisione è l'abilità fondamentale richiesta. Esse possono eseguire una vasta gamma di operazioni, dalla semplice segatura alla fresatura complessa e geometrica, passando per la foratura, la scanalatura, la profilatura ecc. Ciò consente di realizzare prodotti con forme articolate e personalizzate, adattandosi alle richieste specifiche di ogni cliente. Inoltre, le CNC sono in grado di lavorare in modo autonomo una volta inizializzate da un software lanciato da un operatore umano, riducendo di conseguenza, al minimo gli interventi manuali e aumentando la produttività generale. Questo fa anche in modo di ridurre drasticamente i tempi di setup e messa a punto della macchina stessa rispetto alle operazioni di lavorazione tradizionali e viene, quindi, garantita e certificata un'elevata ripetibilità dei processi,

consentendo di produrre grandi quantità di pezzi totalmente identici con la massima precisione. Infine, le CNC possono essere facilmente integrate, in sistemi meccanici - di produzione in scala - automatizzati, ottimizzando i flussi di lavorazione e riducendo i costi, nonché ottemperando, in definitiva, a tutte le esigenze richieste del mercato odierno. Tutte queste caratteristiche messe insieme, rendono tale tipologia di macchine, **un componente indispensabile e fondamentale** per tutte le aziende competenti nel settore della lavorazione di materiali leggeri, consentendo loro di rimanere al passo con i tempi e con l'andamento della competizione globale, conferendo loro un vantaggio strategico e dando loro la possibilità di evolversi verso un contesto pieno di aspettative sempre più rigide e processi lavorativi sempre più automatizzati e rapidi.

1.2 Storia dell'azienda

La storia di SCM nasce quando Nicola Gemmani e Lanfranco Aureli, esperti di fonderie e lavorazioni meccaniche, iniziano una collaborazione che nel 1952 vede nascere la loro prima macchina per la lavorazione del legno, progettata dall'ingegner Giuseppe Gemmani, figlio di Nicola. In breve tempo nascono la B2, la B3 e "L'invincibile B4" - la più celebre -, capace di quattro lavorazioni: pialla filo, pialla spessore, sega a disco e foratrice. In breve tempo, con il marchio L'Invincibile, SCM si presenta con una gamma completa di macchine classiche che le permette di conquistare tutti i mercati, raggiungendo la leadership nel mondo. Di seguito, si mostra il celebre logo aziendale, ideato dai soci dopo l'acquisizione del gruppo industriale e mostrato in questo progetto di tesi con consenso dell'azienda stessa:



Da subito SCM imposta un sistema di produzione di serie allora innovativo, utilizzando componenti e gruppi comuni a più modelli e mettendo le basi del concetto di modularizzazione. Dagli anni '60 ai fondatori si affiancano prima Adriano e poi Alfredo, figli di Lanfranco Aureli, che insieme a Giuseppe Gemmani guideranno e faranno grande l'azienda. Dal 1976, SCM sviluppa i primi centri di lavoro per massello e sistemi per il serramento; dalla metà degli anni '80 inizia una serie di acquisizioni che confermeranno la leadership aziendale a livello internazionale. Nel 1984 viene decisa l'integrazione nel gruppo di Mahros, nel 1985 viene acquisita Minimax, leader nella produzione per le macchine destinate all'hobbistica di alto livello.

Tra il 1986 e il 1987 si perfezionano tre acquisizioni destinate a lasciare il segno nel mercato: la prima, nel 1986, è quella di Gabbiani, leader nelle sezionatrici; quindi, nell'87 entrano nell'orbita del gruppo Dmc, specialista ad altissimo livello nel processo della levigatura e Morbidelli, con cui SCM entra nel settore delle macchine ed impianti per il processo di lavorazione del pannello. Nel 1992 altre due acquisizioni importanti: Routech, con cui SCM sviluppa le tecnologie per il processo di lavorazione di elementi in legno per l'edilizia, e Stefani, specialista della bordatura.

Seguendo la visione strategica iniziale di SCM e con la missione di offrire la più ampia gamma di soluzioni possibili, negli anni '90 si è costituita anche SCM Group S.p.A., una società in cui confluiscono tutte le aziende

già acquisite e che aspirava a un'ulteriore espansione e diversificazione. Nel 2002, attraverso l'acquisizione di CMS, SCM Group è entrata nel mercato delle tecnologie avanzate per la lavorazione di materiali compositi, plastica, vetro, metallo e marmo. Negli anni successivi, altre aziende sono state poi incorporate dal gruppo e, in particolare, da SCM stessa (ovvero Superfici e Valtorta, Celaschi, Sergiani, CPC, Delmac e Balestrini) per il loro prezioso know-how nella lavorazione del legno. In questo modo, è stato possibile ampliare la gamma di prodotti dell'azienda a una tale ampiezza e profondità che non aveva eguali nel settore. Per affermarsi sempre più anche nel mondo dei materiali compositi e plastici, nel luglio 2017 SCM Group ha acquisito anche il 51% della tedesca Hg Grimme e, nel 2018, il gruppo Dms - Diversified Machine Systems, una società di Colorado Springs. Oggi, questa multinazionale può fornire tecnologie avanzate per la lavorazione sia del legno che di un'ampia gamma di materiali alternativi. Inoltre, contemporaneamente, si è specializzata anche nella produzione di servizi e componenti industriali tecnologici per le macchine e i sistemi del Group, per quelli di terzi e per l'industria meccanica in generale. I sei marchi di divisione che ora appartengono al Group sono:

- SCM, settore delle tecnologie per la lavorazione del legno.
- CMS, settore delle tecnologie per la lavorazione di vetro, pietra, metallo, plastica, alluminio e compositi.
- Hiteco, settore della meccatronica.
- Steelmec, settore della lavorazione dei metalli e delle lavorazioni meccaniche.
- ES, settore dei componenti elettrici e dei quadri elettrici.
- SCMfonderie, settore della fonderia di ferro.

Inoltre, con 22 filiali distribuite in 5 continenti e una quota di export superiore al 70%, SCM Group conferma la sua presenza in tutto il mondo con la rete di distribuzione più capillare del settore. Con oltre 4000 dipendenti, ha un fatturato annuo superiore ai 700 milioni di euro e vende i suoi prodotti a giganti industriali come IKEA, Luxottica, Teuco, Boeing, Oracle, SpaceX e Samsonite. Dalla sua nascita ad oggi, la sua gestione è stata affidata alle due famiglie fondatrici, passando dai genitori ai figli in un processo di crescita continuo. Di seguito, si mostra l'immagine reperita - con consenso - direttamente dal sito di SCM Group S.p.A. delle celebri bandiere che rappresentano l'unione del cuore di SCM con le altre industrie del gruppo correlato.



1.3 Servizi offerti

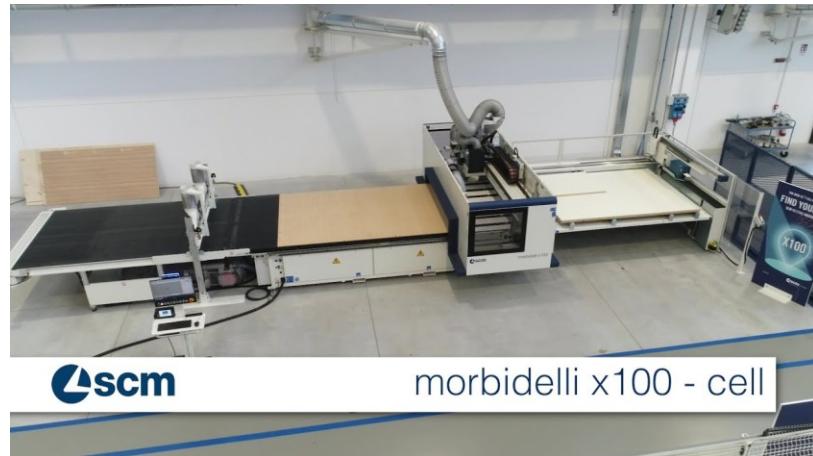
SCM offre una vasta gamma di prodotti e servizi per l'industria secondaria del legno. Questi sono dedicati al mondo dell'arredamento, dell'edilizia, dell'industria e dell'artigianato. I prodotti possono essere suddivisi in due categorie principali:

- **Macchine:** le macchine SCM coprono diverse tecnologie e includono prodotti come centri di lavoro CNC, bordatrici, seghe a pannello, soluzioni di foratura, sistemi di automazione, pialle e profilatrici, sistemi per finestre, tenonatrici e profilatrici, squadratrici e profilatrici, levigatrici e calibratori, soluzioni per il trattamento superficiale, presse, soluzioni di assemblaggio e imballaggio, macchine per la falegnameria, sistemi integrati e linee.
- **Soluzioni digitali:** le soluzioni software, i progetti e i servizi digitali di SCM sono progettati specificamente per la preparazione e l'implementazione della lavorazione del legno, nonché per il monitoraggio, la manutenzione e l'assistenza. Per ogni processo di lavorazione del legno esiste un software specifico, che va a fungere da pannello di controllo e gestionale per l'attività in questione. L'integrazione tra di essi costituisce un sistema molto complesso e articolato che prende il nome di Maestro Digital Systems. I servizi, invece, sono sviluppati per supportare il cliente in tutte le fasi del ciclo di vita della macchina. Sono altamente specializzati e coprono l'installazione e la messa in servizio in produzione, l'assistenza e la manutenzione, la formazione e la fornitura di ricambi dedicati.

Tutti i servizi forniti sono sviluppati per supportare il cliente nelle varie fasi del ciclo di vita dello stesso della macchina. Sono altamente specializzati e coprono l'installazione e la messa in servizio in produzione, l'assistenza e la manutenzione, la formazione e la fornitura di ricambi dedicati. Oltre tutto, i centri di lavoro CNC sono progettati per risolvere tutte le esigenze di lavorazione di pannelli e legno massello per l'industria dell'arredamento, per l'edilizia e per prodotti speciali. L'offerta SCM include una vasta gamma di soluzioni, suddivise in Centri di Lavoro CNC per fresatura, bordatura e foratura classica, per falegnameria, per componenti di sedie e scrivanie e per foratura, fresatura e bordatura in versione nesting. La macchina che verrà trattata, in particolare, in questo progetto di tesi è una: **Morbidelli X100-X100 CELL**, ovvero, una macchina CNC a controllo numerico che fa nesting sul piano martire di lavoro.

1.4 Caratteristiche della macchina

La macchina a controllo numerico Morbidelli X100-X100 Cell fa parte della nuova generazione dei Centri di Lavoro CNC Nesting, che risponde alle esigenze di un mercato sempre più orientato a produzioni flessibili e a lotto 1, garantendo un ottimale rapporto prezzo/prestazioni e produttività. Di seguito, un'immagine della macchina in questione, presa direttamente con consenso dal sito ufficiale di SCM Group S.p.A.



Tale centro di lavoro permette la lavorazione Nesting dei pezzi direttamente sul piano martire adattato alla macchina e, quindi, offre una soluzione perfetta che semplifica il lavoro: infatti, sono presenti diverse versioni disponibili per configurare la macchina in base alle svariate esigenze di produzione, indipendentemente dal flusso dei materiali in lavorazione. Inoltre, il piano martire offre un grande vantaggio, grazie al quale è possibile decidere in che area o quadrante del piano stesso effettuare l'operazione in questione, gestendo, pertanto, in modo strategico anche l'usura relativa del martire, di modo da massimizzarne l'uso effettivo e limitarne gli sprechi. Supporta un'ottima gestione del vuoto poiché, durante la programmazione, è possibile impostare i migliori parametri per la lavorazione da eseguire e scegliere quali aree attivare o disattivare. Il vuoto non si disperde più nelle aree in cui non è necessario e, grazie alle ventose X-POD, qualsiasi lavorazione viene svolta senza rimuovere il pannello martire. Per di più, grazie alla funzione laser incrociato, è possibile riferire sia le ventose che il pezzo da lavorare simultaneamente. Molto importante è anche la possibilità di creazione di geometrie sui pezzi da lavorare, che vengono poi riprodotti, sul materiale, con un innovativo algoritmo di taglio, denominato: Smart Cut, il quale applica sequenze e strategie di taglio ottimizzate per una migliore tenuta dei pezzi, senza incidere sulla velocità di lavorazione. Il piano martire installato sulla macchina ha, solitamente, una profondità di spessore Z che può variare da 8 a 18 mm, il quale diminuisce poi - in maniera regressiva - ad ogni operazione di rettifica effettuata, fino al punto in cui non sarà più possibile rettificarlo e si dovrà obbligatoriamente sostituirlo.

Di seguito, si elencano un po' di caratteristiche tecniche legate alla macchina appena discussa, non create da me ma prelevate direttamente con consenso dal sito di SCM Group S.p.A.:

Dati tecnici

morbidelli x100		
Area utili di lavoro		
Dimensioni X-Y	mm	2556x1296 3756x1596 3756x1896 3156x2216 4356x2216
Assi		
Velocità vettoriale assi X-Y "PRO-SPACE"	m/min	70
Passaggio pezzo in Z	mm	150
Unità di fresatura		
Potenza motore 3-4 assi (fino a)	kW (hp)	15 (20,5)
Velocità di rotazione (fino a)	g/min	24000
Utensili disponibili su magazzino (fino a)	posti	31
Unità di foratura		
Mandirini verticali indipendenti (fino a)	n.	16
Mandirini orizzontali indipendenti (fino a)	n.	8
Velocità di rotazione (fino a)	g/min	8000
Lame integrate (fino a)	n.	2
Lama integrata in X, diametro	mm	125

Come evidente, la macchina nesting in questione si occupa delle operazioni di fresatura e foratura dei pezzi passati in lavorazione, tramite vari utensili adibiti allo scopo inerente.

1.5 Problema dell'usura del piano martire

Utilizzando costantemente la macchina, l'**usura progressiva del relativo piano di lavoro è un fenomeno inevitabile**, che si verifica a causa dell'incidenza continua tra l'utensile da taglio e il materiale lavorato. Con il tempo, la superficie del piano si deteriora in maniera lineare, presentando segni di usura come graffi, solchi e irregolarità, fino alla certezza di raggiungere una condizione irreversibile in cui è necessaria la sostituzione del piano stesso per poter proseguire correttamente con la produzione.

Naturalmente, questo fenomeno porta diverse implicazioni operativamente parlando:

- **Riduzione della precisione:** Un piano di lavoro usurato compromette la precisione delle lavorazioni. Le tolleranze dimensionali e le finiture superficiali dei pezzi lavorati possono risultare inferiori agli standard richiesti, generando scarti e ri-lavorazioni.
- **Degradazione della qualità:** L'usura può causare vibrazioni e instabilità durante la lavorazione, influenzando negativamente la qualità superficiale dei pezzi. Si possono formare impurità, sbavature e imperfezioni che richiedono ulteriori lavorazioni di finitura di terze parti.
- **Aumento dei tempi di lavorazione:** Per compensare l'usura del piano, potrebbe essere necessario ridurre le velocità di taglio e di avanzamento, allungando i tempi di lavorazione e diminuendo la produttività della macchina.

- **Aumento del consumo di utensili:** L'usura del piano può accelerare l'usura degli utensili da taglio, aumentando i costi di manutenzione e i fermi macchina per la sostituzione degli utensili stessi.
- **Rischio di danneggiamento del pezzo:** In casi estremi, un piano di lavoro eccessivamente usurato può causare il danneggiamento del pezzo in lavorazione, con conseguenti fermi macchina prolungati e costi di ripristino elevati.
- **Diminuzione della durata della macchina:** L'usura del piano contribuisce all'invecchiamento prematuro della macchina, riducendone la vita utile e aumentando i costi di manutenzione nel lungo periodo.

È, quindi, di interesse, in primis della clientela ma anche dell'impresa di per sé, che venga promossa una gestione pianificata ed efficace della manutenzione del piano... fondamentale per garantire la qualità dei prodotti, la durata della macchina nel tempo, la sua longevità e la competitività dell'azienda stessa.

Proprio per questa ragione, SCM mette a disposizione, tramite il software di comando Maestro Active, varie funzionalità per ottemperare a questa esigenza... su cui, tra l'altro, è già implementata una scala di valori (da 0 a 3), che corrisponde ad un colore diverso (nullo, verde, giallo, rosso) e che indica il grado di usura corrente del piano martire installato. Di seguito, viene mostrata la legenda di tale schema extrapolato dal codice da me creato:

```
46 references
public enum SpoilboardStatus
{
    /// <summary>
    /// Non definito.
    /// </summary>
    None,

    /// <summary>
    /// Integro.
    /// </summary>
    Ok,

    /// <summary>
    /// Usurato ma ancora utilizzabile.
    /// </summary>
    Warning,

    /// <summary>
    /// Da sostituire.
    /// </summary>
    Critical
}
```

Come si può ben facilmente intendere, i gradi di usura sono associati in questo modo ai relativi colori:

- 0 = Nullo, stato del piano non definito
- 1 = Verde, piano intatto
- 2 = Giallo, piano usurato ma utilizzabile
- 3 = Rosso, piano molto usurato da sostituire

Questo sistema di controllo riassuntivo è, chiaramente, implementato su Maestro Active e disponibile al cliente, in base al quale può decidere il miglior momento, secondo le proprie abitudini operative, in cui effettuare un'operazione di manutenzione (rettifica o sostituzione) alla propria macchina, senza incorrere in problemi di tempistiche ritardanti per la produttività aziendale e seguendo una strategia ben definita, che consente di massimizzare l'operatività generale e minimizzare gli sprechi e i ritardi lavorativi.

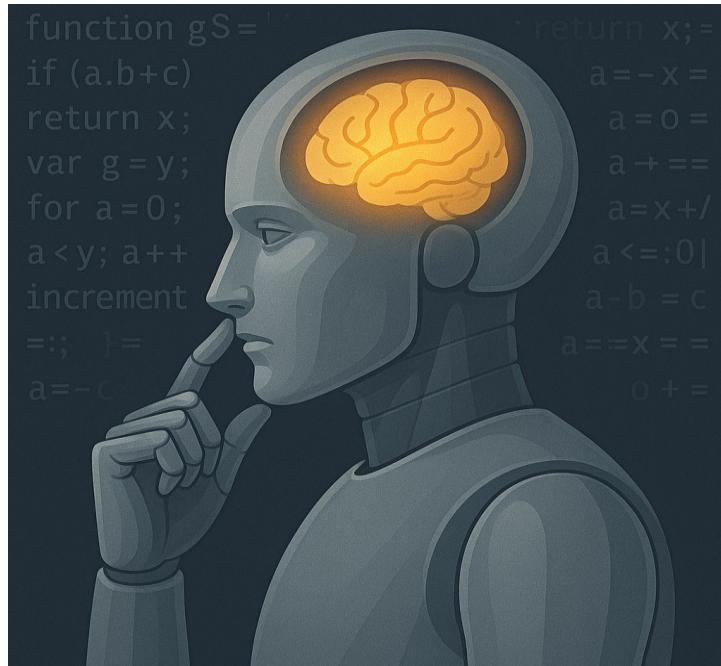
È, dunque, proprio arrivati a questo punto, che entra in gioco il mio contributo da laureando in intelligenza artificiale - basato sulla progettazione di un algoritmo predittivo - al fine di “automatizzare” questo intero processo e trasformare l’interattività euristica corrente in un modello neurale vero e proprio, che sfrutta l’addestramento su grandi quantità di dati per fornire una corretta predizione e apprendere al meglio come adattarsi alle esigenze del cliente stesso.

1.6 Obiettivi del progetto formativo

Per scendere nei dettagli, il mio ruolo in SCM, come laureando in ingegneria dell’intelligenza artificiale, è stato dunque quello di contribuire al progetto di un applicativo software AI-based per macchine a controllo numerico che fanno nesting, il quale deve occuparsi della diagnostica, dell’ottimizzazione e della previsione operativa preventiva dell’usura del piano di lavoro della macchina, tenendo in considerazione le abitudini lavorative del cliente e mettendolo al corrente, tramite notifica po-pup con preavviso su Maestro Active, su quando si presenta il momento migliore per effettuare un’operazione di manutenzione (rettifica o sostituzione) del piano martire installato. Ho iniziato il lavoro, visionando personalmente la macchina su cui avrei dovuto lavorare e studiando approfonditamente e facendo ricerca, come consigliato anche dal mio professore referente, sulle componentistiche principali su cui deve svilupparsi un algoritmo di intelligenza artificiale applicato alla previsione industriale, dopodiché ho redatto un file di testo contenente il piano di azione e l’ho suddiviso nei vari passaggi cardine da seguire per arrivare al compimento dell’incarico. Successivamente, ho iniziato ad analizzare, in locale, il database, fornитоми dai responsabili in SCM, con tutti i record relativi alle lavorazioni esercitate da un demo di macchina nesting... difatti, un esempio virtuale fittizio di macchina nesting, molto simile alla casistica reale su cui l’algoritmo è stato poi effettivamente implementato.

Di importanza cruciale è stata la composizione da zero (e il successivo pre-processing) di un dataset accettabile e abbastanza grande, popolato con i dati esaminati, da essere preso in input dall’algoritmo predittivo; e la scelta dell’architettura neurale consona all’incarico assegnatomì. Ho optato, analizzando e basandomi anche su altri casi di manutenzione predittiva industriale, per l’implementazione e l’utilizzo di una Rete Neurale Ricorrente (RNN) di tipo LSTM Bidirezionale (BiLSTM), di modo tale da poter catturare correttamente tutte le varie dipendenze temporali, precedenti e successive tra un layer di rete nascosto e l’altro e, di fatto, gestire la previsione di usura in modo regressivo tramite un output – con attivazione sigmoid - di strati densi fully connected con un neurone per ogni grado di usura, tenendo conto dell’usura stessa del piano raggiunta fino a quel momento per mezzo delle lavorazioni pregresse. L’obiettivo finale di questo progetto è stato, quindi, quello di automatizzare la procedura di avviso manutenzione in termini predittivi tramite linguaggio di programmazione Python e, integrare poi in linguaggio C# (utilizzato in azienda su tutti i software attualmente in uso) tramite API ReST per l’accesso al database aziendale, il software finito e completo all’interno del gestionale Maestro Active.

“L’immagine inserita di seguito è un’immagine vettoriale generata tramite intelligenza artificiale utilizzando ChatGPT di OpenAI.”



1.7 Contributi

I contributi forniti dall'intelligenza artificiale in questo contesto e dalla progettazione di un algoritmo predittivo sono, sicuramente, aspetti da non sottovalutare ai fini del miglioramento tecnico, inerenti alla versatilità con cui è possibile gestire le operazioni di manutenzione del martire e, quindi, rendere possibile, al cliente, la programmazione di un piano dettagliato e strategico di azione secondo le sue abitudini operative.

Tali aspetti, mirano a risolvere un problema concreto e assicurano una diminuzione delle tempistiche manutentive e una continuità nella produttività generale aziendale. Avendo a disposizione un sistema che invia notifiche al cliente basandosi su una previsione operativa si garantisce, infatti, al cliente stesso, la possibilità di conoscere, per tempo, il momento esatto in cui il piano martire andrà rettificato o sostituito, in modo tale da avere la possibilità di organizzare in maniera minuziosa e chiara il proprio lavoro, assicurando anche di ottimizzare tutti i tempi morti e pianificare l'operazione manutentiva in un orario in cui la macchina non lavorerebbe e la produzione non verrebbe, così, limitata o temporaneamente sospesa.

Sicuramente, la manutenzione predittiva industriale è un tema complesso e ancora in fase emergente, che verrà valorizzato sempre di più con la diffusione crescente dell'intelligenza artificiale e che sarà posto al centro del mirino di tutte le aziende di tale settore. Inoltre, può costituire una base solida per l'idea di progetti predittivi derivati come: la predizione in tempo reale dell'usura degli utensili della macchina e la predizione di quanti e di quali pezzi effettivi è possibile produrre in scala fino alla prossima operazione di manutenzione del pannello martire e tanti altri.

Oltre tutto, una peculiarità del modello da me implementato, è l'addestramento dinamico a cui viene sottoposto. Per essere più precisi, il modello si interfaccia direttamente con un dataframe (ovvero una struttura dati tipica delle reti neurali fornita dalla libreria Pandas di Python), che viene passato in input e generato da un altro apposito vettore di dati, il quale si aggiorna man mano che viene scritto un nuovo record nel database dal quale si attingono tutte le informazioni di interesse. In parole poche, ogni qualvolta che una

lavorazione o un'operazione di manutenzione viene lanciata, dall'operatore, viene scritto un record sul database di Maestro Active della macchina attualmente in uso; in questo modo, il modello riesce ad includere dinamicamente tutte le informazioni disponibili in tempo reale aggiornate all'ultima operazione effettuata e a far convergere le proprie conoscenze in una realtà sempre più precisa e dettagliata.

Pertanto, più esempi di training significativi vengono forniti e più il modello riesce ad apprendere correttamente le features da considerare nel training stesso e i target su cui effettuare le predizioni.

Infine, se il dataframe iniziale, a causa della grande quantità di dati generati, diventasse troppo grande per essere processato correttamente, sarebbe possibile chiamare, tramite terminale di comando, una procedura di formattazione, da me ideata, che va a eliminare i record di lavorazioni più vecchi, ordinati tramite politica FIFO (First-in, First-out), secondo una costante numerica N, che indica il numero di record effettivo da cancellare... così da mantenere il set di dati sempre uniforme e di dimensioni costanti ma, allo stesso tempo, sempre aggiornato con gli ultimi dati rilevanti acquisiti.

Per tutti questi motivi, si potrebbero riassumere alcune parole chiave per indicare il perché è importante includere un sistema intelligente di questo tipo in un contesto di lavorazione industriale. Si può parlare di contributi consistenti in termini di:

- Affidabilità
- Granularità
- Flessibilità
- Pianificazione strategica
- Risparmio di tempo e denaro
- Ottimizzazione
- Innovazione
- Evoluzione visionaria
- Limitazione di danni e problematiche
- Industria 4.0
- Automatizzazione di processi lavorativi macchinosi
- Incremento della produttività e della velocità operativa
- Minimizzazione dell'errore umano
- Qualità lavorativa migliorata per l'operatore e per l'azienda
- Apertura aziendale verso un settore in forte crescita
- Acquisizione di nuove skills e competenze
- Coniazione di nuovi posti di lavoro nell'impresa
- Formazione del personale operativo sull'intelligenza artificiale
- Investimento su personale qualificato e stakeholders più consapevoli

Come è evidente notare, esiste più di un valido contributo per includere il campo dell'intelligenza artificiale nel proprio business industriale.

2. Stato attuale dell'arte

2.1 Manutenzione predittiva

Come è già stato accennato in precedenza, la manutenzione predittiva è un approccio che mira a prevedere i guasti delle macchine prima che si verifichino, permettendo di pianificare gli interventi di manutenzione in modo ottimale compatibilmente con l'andamento della produzione. L'obiettivo è ridurre i fermi macchina non programmati, aumentare la durata dei componenti e diminuire i costi di manutenzione generali.

Chiaramente, la panoramica sulla manutenzione predittiva è molto ampia... infatti finora, non comprendeva solo l'intelligenza artificiale, bensì anche altre tecniche tradizionali che sono state utilizzate da sempre come metodi standard per far fronte a questo tipo di problematica. Alcune di queste tecniche ricorrenti sono:

- **Analisi vibrazionale:** Misurazione delle vibrazioni prodotte dalla macchina per individuare anomalie che possono indicare un imminente guasto.
- **Termografia:** Misurazione della temperatura superficiale dei componenti per individuare surriscaldamenti anomali, spesso indicativi di problemi interni.
- **Analisi dell'olio:** Analisi dell'olio lubrificante per rilevare la presenza di particelle metalliche, sfregi o di altri contaminanti che possono indicare usura o danneggiamento del piano martire o dei componenti utensili.
- **Ispezioni visive:** Ispezioni regolari delle macchine per individuare eventuali segni di usura del piano martire, corrosione o danneggiamento e applicare poi le dovute contromisure.

Tuttavia, dal momento che l'intelligenza artificiale ha effettuato il proprio ingresso nella realtà odierna, i metodi di approccio si sono ampliati notevolmente, offrendo strumentazioni più sofisticate e accurate per prevedere qualunque tipo di guasto. Alcuni di questi metodi comprendono:

- **Apprendimento automatico:**
 - **Regressione:** Utilizzata per prevedere un valore continuo, come il tempo rimanente prima di un guasto, basandosi su dati di usura reali... in parte utilizzato nella mia tesi di laurea.
 - **Classificazione:** Utilizzata per classificare lo stato di una macchina in categorie predefinite appositamente codificate (normale, anomalo, guasto), che è anche il metodo che ho scelto di utilizzare nel mio progetto di tesi.
 - **Clustering:** Utilizzato per raggruppare i dati in cluster omogenei, identificando e distinguendo pattern nascosti ricorrenti che possono indicare anomalie da pattern che indicano invece una situazione stabile.
- **Reti neurali artificiali:**
 - **Reti neurali profonde:** Sono reti con tanti livelli nascosti interconnessi tra loro, in grado di apprendere da grandi quantità di dati e di identificare pattern complessi, come quelli associati a guasti nascosti o a funzionamenti anomali.
 - **Reti neurali ricorrenti (RNN):** Utilizzate per analizzare dati sequenziali, come le vibrazioni nel tempo, l'usura di un determinato componente e identificare vantaggi, guasti, tendenze e anomalie.

- **Reti neurali convoluzionali (CNN):** Utilizzate per analizzare dati visivi, spesso in formato topologico, come lo schema di un determinato piano di lavoro e catturare features visive di interesse, positive e negative, per utilizzarle come input nel determinare qualunque tipo di operazione su cui si voglia fare previsione.
- **Algoritmi di regressione:** Algoritmi predittivi classificativi di machine learning come RandomForest o XGBoost, che servono prettamente per valutare l'andamento prestazionale di una o più caratteristiche “a perdere” di un componente della macchina stessa.
- **Reinforcement Learning:** Utilizzato sulle reti neurali per ottimizzare le decisioni di manutenzione in base a un sistema di ricompense e penalità per il sistema intelligente.
- **Elaborazione del linguaggio naturale (NLP):** Utilizzato soprattutto per analizzare i dati testuali, come i report di manutenzione o i report dei sensori installati sulle macchine, per estrarre informazioni rilevanti e identificare potenziali problemi.

Naturalmente, queste metodologie portano con sé un grande numero di vantaggi, come, ad esempio:

- **Maggiore precisione:** I modelli di AI sono in grado di identificare pattern complessi e non lineari nei dati, che possono essere difficili da individuare con metodi tradizionali ed euristicci.
- **Previsioni più accurate:** L'AI permette di effettuare previsioni più accurate e a lungo termine, consentendo di pianificare la manutenzione in modo più efficiente e sicura.
- **Automatizzazione:** Molti dei processi di analisi e previsione possono essere automatizzati, riducendo il carico di lavoro degli operatori e ridirezionando il loro focus altrove.
- **Integrazione con altri sistemi:** I modelli di AI possono essere integrati facilmente con altri sistemi aziendali, come i sistemi di gestione della produzione per ottimizzare i processi produttivi o sistemi per la previsione dell'andamento della produzione e del mercato globale.

Essendo un mondo in sviluppo costante e nato da poco, chiaramente, possono presentarsi anche alcuni svantaggi iniziali che devono assolutamente essere tenuti in considerazione nella pianificazione della strategia aziendale, ovvero:

- **Qualità dei dati:** La qualità dei dati utilizzati per addestrare i modelli di IA è fondamentale per ottenere risultati accurati... spesso però, questa qualità ambita non è facile da reperire nella totale interezza e, pertanto, può succedere di ritrovarsi a lavorare con dataset incompleti o sbilanciati con dati rumorosi.
- **Costo:** L'implementazione di soluzioni basate sull'IA può richiedere investimenti significativi in termini di hardware, software e competenze operative. Essendo una disciplina nata da poco, non è sempre semplice assumere personale con le giuste abilità per affrontare determinati problemi correlati al settore, specie in un contesto emergente.
- **Interpretabilità:** Alcuni modelli di IA, come le reti neurali profonde, possono essere macchinose nell'implementazione e generare dei risultati difficili da interpretare a livello logico, rendendo arduo comprendere come si è arrivati ad una determinata previsione, per ricostruire in modo profondo e coinciso tutti i passaggi che sono stati effettuati.
- **Difficoltà nella pianificazione:** Non è sempre inizialmente ben nitida, a livello logico, la metodologia di come rappresentare, in modo chiaro e non complesso, ciò che si vuole ottenere a livello predittivo, partendo da una base di programmazione euristica e iterativa e, tradurlo poi successivamente, in una logica applicativa consona alla giusta comprensione da parte della rete.
- **Mancanza di supporto tecnico:** Se l'azienda non ha investito adeguatamente risorse nella formazione del personale tecnico riguardo al mondo dell'intelligenza artificiale, si potrebbe

riscontrare mancanza di supporto diretto dove, coloro che possono effettivamente occuparsi di progetti su algoritmi intelligenti aziendali, ricoprirebbero una minima parte del team di sviluppo e potrebbero non essere numericamente sufficienti, a lungo termine, per mantenere attivi e operativi tali progetti nel corso del tempo... per di più, negli anni a venire, potrebbero verificarsi congedi e cambi continui di personale, che renderebbero oneroso, per i nuovi candidati, riuscire a comprendere e ad analizzare il codice già esistente, in modo lineare, se non dispongono delle adeguate competenze in materia.

È quindi fondamentale, se si vuole integrare progetti di IA nella produzione aziendale, certificare nel dipartimento tecnico una valida preparazione a riguardo e accogliere questa nuova disciplina con consapevolezza e desiderio di apprendimento costante, innovazione ed evoluzione di pari passo con la relativa sua progressione. A prescindere da tutto, l'IA offre, senza ombra di dubbio, un grande potenziale per migliorare l'efficienza e l'affidabilità dei sistemi di produzione. Combinando le tecniche standard tradizionali di manutenzione predittiva con le soluzioni basate sull'IA, infatti, le imprese possono ottenere risultati ancora migliori in termini di riduzione dei costi, aumento della produttività e miglioramento della qualità del lavoro e dei prodotti.

2.2 Applicazioni dell'IA nel settore manifatturiero

Oltre alla vera e propria manutenzione predittiva, l'intelligenza artificiale può essere applicata in tanti altri contesti del settore manifatturiero, in quanto si può parlare decisamente di una rivoluzione di massa vera e propria, che va ad offrire e a supportare una vasta gamma di operazioni e applicazioni, sponsorizzando dunque una vera e propria innovazione in qualunque contesto industriale.

Esiste una moltitudine di divisioni industriali in cui tale disciplina potrebbe sinergizzare al meglio con le architetture hardware e software già esistenti, ad esempio:

- **Motori elettrici:** Analizzando i dati relativi a vibrazioni, corrente elettrica e temperatura, l'AI può prevedere guasti imminenti come cuscinetti usurati o avvolgimenti danneggiati.
- **Trasmissioni meccaniche:** Monitorando parametri come vibrazioni, rumore e temperatura, è possibile prevedere guasti a ingranaggi, meccanismi metallici e alberi di trasmissione.
- **Sensori:** L'AI può essere utilizzata persino per monitorare lo stato di salute dei sensori stessi installati sulle macchine per la raccolta di features, garantendo, di fatti, l'accuratezza delle misure e prevenendo guasti che potrebbero compromettere la successiva qualità dei dati raccolti.
- **Andamento finanziario:** L'AI predittiva può essere utilizzata per monitorare l'andamento finanziario del mercato globale attuale di interesse ed effettuare quindi previsioni stocastiche sugli andamenti statistici futuri, di modo tale da riuscire a rimanere sempre nell'ottica competitiva del mercato in questione e sotto i riflettori come multinazionale competente.

Oltretutto, l'intelligenza artificiale è anche molto valida in un discorso prettamente di ottimizzazione dei processi produttivi... infatti, può essere utilizzata per ottimizzare direttamente diversi aspetti importanti:

- **Pianificazione della produzione:** L'IA può analizzare i dati storici di produzione, le previsioni di domanda e le capacità produttive per creare piani di produzione ottimizzati e personalizzati, minimizzando i tempi di fermo macchina e massimizzando l'utilizzo delle risorse a disposizione.
- **Controllo di qualità:** L'IA può essere utilizzata per ispezionare i prodotti finiti, identificando difetti e anomalie che potrebbero sfuggire all'occhio umano. Ad esempio, in questa attività, sono di enorme aiuto le reti neurali convoluzionali, che vengono utilizzate nell'analisi di qualunque tipologia e formato di immagini di prodotti, per individuarne difetti superficiali, caratteristiche chiave e altre componenti rilevanti. Inoltre, può gestire:
 - Analisi dei dati di processo: Analizzando i dati raccolti durante il processo produttivo, l'IA può identificare le cause alla radice di qualunque tipo di difetti e suggerire così azioni correttive
 - Test non distruttivi: L'IA può essere utilizzata per analizzare i dati provenienti da test non distruttivi, come esiti determinati da raggi X o ultrasuoni, per identificare, di conseguenza, difetti interni nella vera e propria "scatola nera" del prodotto stesso.
- **Logistica:** L'IA può ottimizzare la gestione della catena di approvvigionamento industriale, prevedendo la domanda in anticipo, ottimizzando i percorsi di trasporto e gestendo i magazzini in modo più efficiente e automatico, tramite anche l'uso di bracci robotici autonomi e piattaforme meccaniche gestite algoritmamente.
- **Robotica e meccatronica:** L'IA può essere utilizzata per migliorare la capacità dei robot di interagire con l'ambiente circostante, di apprendere nuove attività e di collaborare con gli operatori umani, al fine di portare a termine con successo qualunque tipo di task assegnato.
- **Sicurezza:** L'IA può essere utilizzata per monitorare e supportare la sicurezza sui luoghi di lavoro, identificando comportamenti a rischio, prevenendo incidenti e istruendo il personale operativo umano e robotico sui possibili rischi imminenti.
- **Sostenibilità:** L'IA può contribuire a ridurre l'impatto ambientale dell'industria manifatturiera, ottimizzando il consumo energetico, riducendo l'inquinamento e gli scarti derivati, applicando piani di azione per il riutilizzo di materiale in eccesso, il riciclo e lo smaltimento corretto dei rifiuti e promuovendo interesse per l'economia circolare.

In conclusione, l'intelligenza artificiale offre un potenziale enorme per trasformare il settore manifatturiero, rendendolo più efficiente, flessibile e sostenibile. Le applicazioni dell'IA sono in continua evoluzione e si prevede che avranno un impatto sempre maggiore sui processi produttivi futuri.

2.3 Tecniche di Deep Learning

La branca più gettonata, nell'ambito dell'intelligenza artificiale, per risolvere problemi di manutenzione predittiva è sicuramente rivolta all'utilizzo di tecniche di Deep Learning, o apprendimento profondo, che consente di esaminare attentamente tutti i dati di interesse, realizzando poi, di conseguenza, features apposite da prendere in considerazione nella costruzione dell'architettura del modello predittivo definitivo.

Il Deep Learning è quel sottocampo di ricerca dell'apprendimento automatico (*Machine Learning*) che si basa su diversi livelli di rappresentazione, corrispondenti a gerarchie di caratteristiche di fattori o concetti, dove i concetti di alto livello sono definiti sulla base di quelli di basso livello. In altre parole, per apprendimento profondo si intende un insieme di tecniche basate su reti neurali artificiali (ANN) organizzate in diversi strati,

dove ogni strato calcola i risultati, li invia in output, consegnandoli poi in input a quello successivo, affinché l'informazione venga elaborata in maniera sempre più completa e si arrivi ad ottenere un output finale definitivo sempre più preciso e dettagliato. Le reti neurali artificiali, per l'appunto, imparano continuamente utilizzando circuiti di feedback correttivi per migliorare le proprie analisi predittive dei dati. In parole povere, è possibile pensare a dati che fluiscono dal nodo di input iniziale al nodo di output finale tramite molti percorsi diversi nella rete neurale stessa. Solo un percorso, però, è quello corretto, che mappa il nodo di input verso il nodo di output giusto. Per indirizzarsi adeguatamente su questo percorso, la rete neurale utilizza un circuito di feedback che funziona nel modo seguente:

1. Ogni nodo fa un'ipotesi sul nodo successivo del percorso.
2. Controlla se l'ipotesi è corretta. I nodi assegnano valori di peso maggiori ai percorsi che conducono a ipotesi più corrette e valori di peso minori a percorsi di nodi che conducono a ipotesi errate.
3. Per il successivo punto di dati, i nodi formulano una nuova previsione utilizzando i percorsi di peso maggiore, per poi ripetere il passaggio 1.

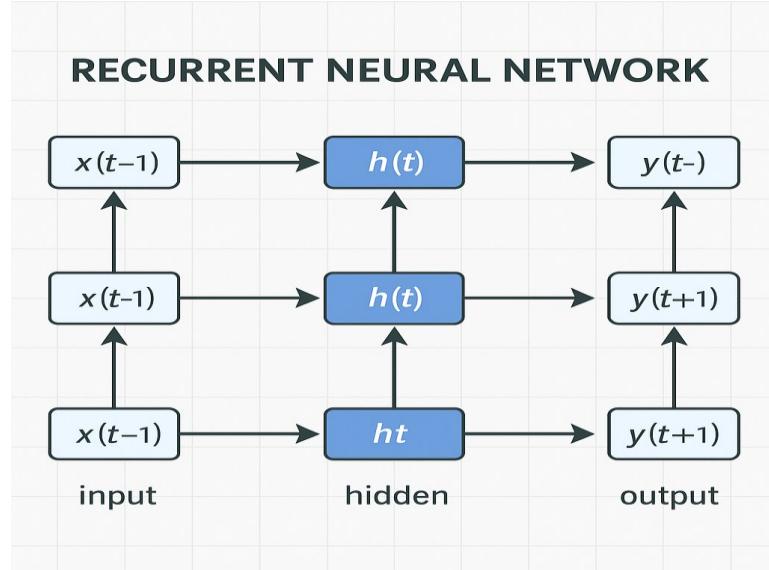
Tra le architetture di spicco nell'apprendimento profondo si annoverano le già accennate reti neurali (NN), che si possono raggruppare in: reti neurali feed-forward (FF), reti neurali convoluzionali (CNN), deep belief network (DBN), reti neurali generative (GAN) e reti neurali ricorrenti (RNN), che sono state applicate nella visione artificiale, nell'analisi di sequenze di dati correlati, nel riconoscimento automatico del discorso, nell'elaborazione del linguaggio naturale, nel riconoscimento audio e nella bioinformatica.

Per finalizzare l'obiettivo in modo corretto, ho condotto varie ricerche e approfondito l'argomento in maniera più precisa, optando dunque per l'implementazione e l'utilizzo di una specifica rete neurale ricorrente (RNN) Long Short-Term Memory (LSTM), più precisamente di una LSTM Bidirezionale (BiLSTM). Prima di entrare nel dettaglio nel cuore delle LSTM però, è necessario effettuare un piccolo preambolo riguardante le RNN nel loro insieme, per comprendere al meglio di cosa si tratti nell'effettivo.

Una **rete neurale ricorrente**, o **RNN**, è una tipologia di rete neurale artificiale che include neuroni collegati tra loro in un ciclo. Tipicamente, i valori di uscita di uno strato di un livello superiore sono utilizzati in ingresso di uno strato di livello inferiore. Questa interconnessione tra strati permette l'utilizzo di uno degli strati come memoria di stato, e consente - fornendo in ingresso una sequenza temporale di valori - di modellarne un comportamento dinamico temporale dipendente dalle informazioni ricevute agli istanti di tempo precedenti.

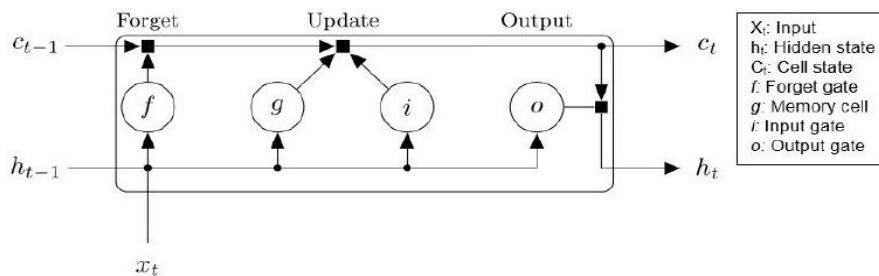
Le RNN usano le informazioni passate per migliorare le prestazioni di una rete neurale su input attuali e futuri. Contengono uno strato nascosto e loop, che consentono alla rete di memorizzare le informazioni passate nello strato nascosto stesso e di operare su sequenze. Esse possiedono due pesi: uno per il vettore dell'hidden layer e uno per gli input. Durante l'addestramento, la rete apprende i pesi sia per gli input che per lo strato nascosto. Dopo l'implementazione, l'output è basato sull'input attuale, così come sull'hidden layer, che è basato sugli input precedenti. In altri casi, tale strato è costituito da un insieme di neuroni dotato di loop di connessioni molto sparse, il quale innesca una dinamica caotica impiegata per l'addestramento di una parte successiva della rete. Ciò le rende applicabili a compiti di analisi predittiva su sequenze di dati, quali possono essere - ad esempio - il riconoscimento della grafia, il riconoscimento vocale o la classificazione di sequenze temporali di dati.

L'immagine seguente – in formato .emf - è stata generata con ChatGPT di OpenAI.



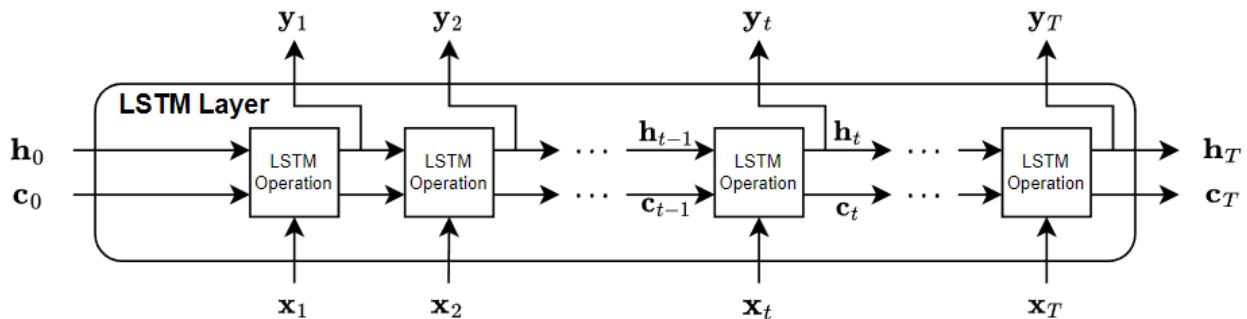
Nella pratica, tuttavia, le RNN semplici sono limitate nella loro capacità di apprendere le dipendenze a lungo termine. Infatti, vengono comunemente addestrate mediante backpropagation, mediante la quale si possono riscontrare problemi di scomparsa (**vanishing gradient**) ed esplosione del gradiente (**exploding gradient**). Questi problemi fanno sì che, i pesi della rete, diventino molto piccoli oppure molto grandi, limitando l'efficacia nelle applicazioni che richiedono un apprendimento relazionale nel lungo periodo.

Pertanto, detto ciò, è necessario utilizzare una “versione” delle RNN che sia in grado di sopperire a questa mancanza... ed è qui che entra in gioco la rete neurale ricorrente LSTM. Le reti LSTM sono una forma più specializzata dell’architettura RNN. Infatti, possiedono una struttura a strati, i quali usano porte aggiuntive per controllare quali informazioni, nello stato nascosto, vengono esportate come output e nel successivo strato nascosto ulteriore. Queste porte aggiuntive superano il problema comune delle RNN di apprendere le dipendenze a lungo termine. Di fatti, oltre allo strato nascosto nelle RNN tradizionali, l’architettura per un blocco LSTM comprende, in genere, una cella di memoria, una porta sigmoide di input, una porta sigmoide di output e una porta sigmoide di dimenticanza. Le porte aggiuntive consentono alla rete di apprendere le relazioni a lungo termine nei dati in modo più efficace e preciso. Una minore sensibilità al divario temporale rende le reti LSTM migliori delle RNN semplici per l’analisi di dati sequenziali e memorizzazione a lungo termine.



I pesi e i bias alla porta di input controllano fino a che punto un nuovo valore confluisce nella unità LSTM. Analogamente, i pesi e i bias alla porta di dimenticanza e alla porta di output controllano, rispettivamente, fino a che punto un valore rimane nell'unità e fino a che punto il valore nell'unità viene usato per calcolare l'attivazione dell'output del blocco LSTM.

Il seguente diagramma illustra il flusso di dati attraverso uno strato LSTM con più fasi temporali. Il numero di canali nell'output corrisponde al numero di unità nascoste nello strato LSTM:



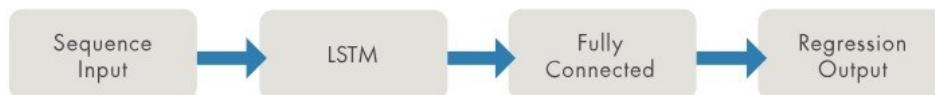
Particolare importanza è assunta dall'architettura vera e propria della LSTM. Essenzialmente, è noto che le LSTM funzionino bene con dati di sequenze e di serie storiche per attività di classificazione e regressione, come con i video, che sono sostanzialmente una sequenza di fermo-immagini. Analogamente all'utilizzo coi segnali, come nelle RNN, è utile eseguire un'estrazione dei features prima dell'alimentazione delle sequenze di immagini nello strato LSTM. Si effettua sfruttando le reti neurali convoluzionali (CNN) (ad esempio GoogleNet) per estrarre i features su ciascun frame.

Il mio argomento di tesi tratta, nello specifico, un problema di regressione classificativo e, proprio per questo, si è deciso di collegare degli strati densi fully connected al resto degli strati nascosti LSTM precedenti. Uno schema semplificato dei macro-strati della rete in questione può essere rappresentato e riassunto come nell'immagine seguente:

CLASSIFICATION

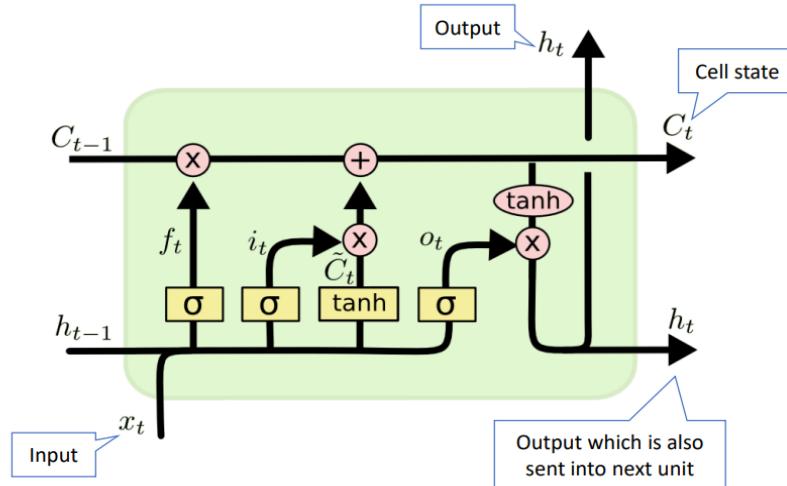


REGRESSION



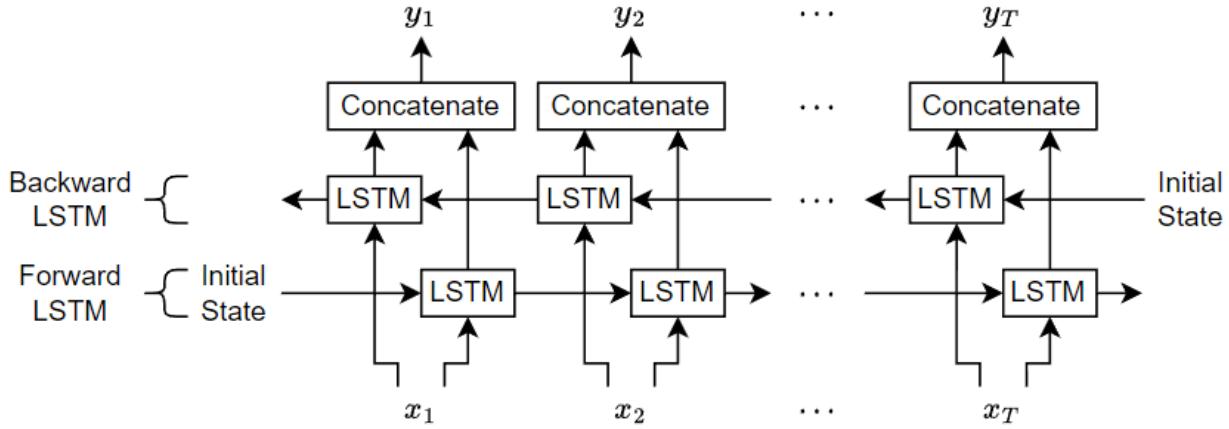
Più schematicamente, i livelli di una LSTM possono riassumersi nella sigla folkloristica F.I.G.O. e sono rappresentati in questo modo:

- Forget Gate Layer: Determina quanta parte dell'informazione memorizzata nella cella di memoria verrà dimenticata.
- Input Gate Layer: Determina quanta parte della nuova informazione verrà aggiornata nella cella di memoria.
- Gate Gate Layer (Update Gate Layer): Determina l'aggiornamento effettivo della nuova informazione dentro la cella di memoria con funzione di attivazione tanh.
- Output Gate Layer: Determina quanta informazione della cella di memoria verrà effettivamente utilizzata per calcolare l'output della cella.



Una BiLSTM invece è una variante molto diffusa della LSTM tradizionale, proprio perché viene utilizzata per apprendere le dipendenze – in entrambe le direzioni - tra fasi temporali di dati di sequenze o di serie storiche. Queste dipendenze possono essere utili quando si desidera che la rete impari dalle serie storiche complete in ogni fase temporale. Le reti BiLSTM, generalmente, consentono un addestramento aggiuntivo in quanto, i dati di input, vengono fatti passare due volte attraverso lo strato LSTM, il che può aumentare le prestazioni della rete, perché l'analisi è ulteriormente accurata anche se più lenta.

Una BiLSTM è costituita da due componenti LSTM fondamentali: LSTM in avanti e LSTM all'indietro. La LSTM in avanti funziona dalla prima fase temporale all'ultima. La LSTM all'indietro funziona dall'ultima fase temporale alla prima. Ciascuno di questi livelli genera i propri strati nascosti, che vengono poi combinati in ogni fase effettiva per formare una rappresentazione completa dei dati di input. Questa combinazione può essere ottenuta tramite concatenazione o sommatoria, consentendo al modello di sfruttare in modo efficace le informazioni provenienti da entrambe le direzioni in parallelo. Di conseguenza, le LSTM bidirezionali sono particolarmente abili nel comprendere il contesto e le sfumature nelle sequenze di dati. A livello fisico, dopo aver fatto passare i dati attraverso le due macro-aree delle LSTM, l'operazione concatena gli output lungo la dimensione del canale e come output effettivo si ha il merge dei due output relativi al canale della rete, come si può notare nella figura seguente:



Un'architettura di questo calibro conferisce numerosi vantaggi in più rispetto ai tradizionali metodi LSTM unidirezionali, soprattutto nelle problematiche in cui la dipendenza del contesto sequenziale è di importanza cruciale, rendendo le LSTM bidirezionali ampiamente utilizzate in vari domini grazie alla loro capacità di gestire dati sequenziali in modo efficace. Infatti, ad esempio, nell'elaborazione del linguaggio naturale, vengono impiegate per attività quali l'analisi del sentimento, la traduzione automatica e il riconoscimento di entità denominate. In aggiunta, considerando l'intero contesto di una frase, le LSTM bidirezionali possono discernere il significato delle parole in base ai termini circostanti, portando a interpretazioni significativamente più accurate. Inoltre, nell'analisi delle serie temporali, le LSTM bidirezionali possono prevedere i valori futuri comprendendo sia le tendenze storiche passate che i potenziali modelli futuri, rendendoli preziosi in campi come la finanza, la sanità e la manutenzione predittiva.

Uno dei principali vantaggi delle LSTM bidirezionali è la maggiore capacità di acquisire dipendenze temporali e sequenziali a lungo raggio nei dati. Le RNN tradizionali spesso riscontrano difficoltà ad affrontare questo tema, a causa di problemi noti e fastidiosi come i gradienti evanescenti, ma l'architettura di cui invece dispone LSTM mitiga tutto ciò attraverso i suoi meccanismi di gate layers. Elaborando i dati in entrambe le direzioni, le LSTM bidirezionali possono comprendere ancora meglio le relazioni tra passaggi temporali anche sequenzialmente distanti tra loro, portando così ad una maggiore precisione nelle previsioni e classificazioni. Inoltre, questa struttura è particolarmente vantaggiosa negli scenari in cui il contesto è cruciale, come comprendere il sentimento di una frase o la semantica logica dietro una query o uno script complesso. Pertanto, anche nell'ambito di manutenzione predittiva industriale, può risultare estremamente favorevole, in quanto, di prassi, è necessario catturare tutte le dipendenze temporali dei features tra un processo lavorativo e l'altro, memorizzarle, processarle e trasmetterle allo step successivo della rete, per poter implementare di conseguenza, tramite la logica opportuna, un algoritmo regressivo di calcolo inherente ad una qualunque caratteristica di cui si desideri conoscere l'impatto crescente nel tempo, come l'usura di un pannello di lavoro, l'usura di utensili e di macchinari lavorativi, l'incremento o il decremento del numero di lavorazioni o di manutenzioni in un certo lasso di tempo ecc.

Dal momento in cui esistono grandi vantaggi nel prediligere un'architettura BiLSTM, sono chiaramente presenti anche alcune sfide e, talvolta, limitazioni. Un problema significativo è la maggiore complessità computazionale associata all'elaborazione dei dati in due direzioni opposte. Ciò può comportare tempi di addestramento più lunghi e un maggiore consumo di risorse, il che potrebbe rappresentare un problema per set di dati di grandi dimensioni o applicazioni in tempo reale. Inoltre, sebbene le LSTM bidirezionali eccellono

nel catturare il contesto, potrebbero comunque riscontrare difficoltà con sequenze molto lunghe di dati, in cui le relazioni tra elementi, distanti tra loro, sono deboli. In questi casi, se si dispone di un hardware favorevole (GPU, TPU ecc.) architetture alternative e innovative, come Transformers, possono fornire prestazioni migliori. Per questa ragione, è conveniente utilizzare LSTM se il problema che si deve affrontare si sposa correttamente con la politica architetturale della rete... in quel caso le prestazioni risultano elevate e l'output potrà godere di un'ottima qualità tecnica e di un'ottima accuratezza.

L'implementazione della sola parte modellistica LSTM bidirezionale nei framework di deep learning più diffusi come TensorFlow o PyTorch è relativamente semplice. In TensorFlow, ad esempio, è possibile utilizzare il wrapper ‘tf.keras.layers.Bidirection’ attorno a un livello LSTM per creare un livello LSTM bidirezionale. Ciò consente una facile integrazione nelle architetture esistenti e facilita la sperimentazione con diverse configurazioni. È fondamentale ottimizzare gli iperparametri come il numero di unità neuronali LSTM, i tassi di loss e le dimensioni dei batch per ottenere prestazioni ottimali. Inoltre, il pre-processing appropriato dei dati di input, inclusa la normalizzazione dei dati numerici, il riempimento dei valori mancanti ecc., è essenziale per una formazione efficace.

In conclusione, siccome il campo del Deep Learning continua a evolversi costantemente, è probabile che la ricerca sulle LSTM bidirezionali esplori nuovi modi per migliorare le prestazioni generali e applicabilità tecnica. Un'area di interesse è l'integrazione dei meccanismi dei livelli di attenzione (Attention layers), che possono indirizzare il modello a focalizzarsi su parti specifiche della sequenza di input esaminando contemporaneamente le altre parti della sequenza stessa (come succede nei modelli Transformer), migliorando ulteriormente la sua capacità di catturare informazioni rilevanti. Inoltre, potrebbero emergere altri modelli ibridi che combinano LSTM bidirezionali con altre architetture, come CNN o Transformers, sfruttando i punti di forza di ciascuno per affrontare attività complesse. I continui progressi nell'hardware e negli algoritmi contribuiranno anche all'efficienza e all'efficacia delle LSTM bidirezionali nelle applicazioni del mondo reale.

3. Metodologia

3.1 Raccolta dati

È il momento di addentrarsi nel cuore vero e proprio di questo progetto di tesi, trattando esplicitamente la metodologia e i passaggi fondamentali che sono stati implementati per costruire l'architettura del modello predittivo e averne resa possibile l'effettiva realizzazione.

Innanzitutto, prima di poter essere arrivati a costruire lo scheletro vero e proprio del modello, è stata chiaramente necessaria la raccolta di una quantità importante di dati e il successivo pre-processing, in modo tale da garantire un'accuratezza di validazione consistente e avere più "materiale" su cui poter lavorare.

L'incarico è iniziato con il confronto diretto con i miei responsabili in SCM, tramite i quali ho avuto l'opportunità di visionare direttamente, in prima persona, la macchina nesting **Morbidelli X100-X100 CELL**, per avere un'idea generale di quale aspetto tecnico progettuale sarei andato a trattare nello specifico. La richiesta principale era:

"Progettare un algoritmo predittivo in forma di rete neurale che fosse in grado di fornire all'utente una predizione accurata di quante lavorazioni ancora eseguibili rimanessero da poter essere effettuate prima dell'operazione di manutenzione corrispondente e, di conseguenza, che fosse in grado di classificare anche il prossimo evento manutentivo da compiere, secondo le abitudini, operative e manutentive storiche del cliente stesso in questione apprese durante il training."

Con questa premessa, ho iniziato ad organizzare il mio piano di azione per il recupero dei dati iniziali. Il piano era così suddiviso in alcuni step principali:

- 1) Scelta del linguaggio di programmazione adeguato e dell'IDE di sviluppo più appropriato:
Col fatto che era richiesta, in modalità strettamente necessaria, l'implementazione di un algoritmo basato sull'intelligenza artificiale, ho deciso di utilizzare per il mio problema, il linguaggio di programmazione Python, poiché è un linguaggio molto versatile dal punto di vista sintattico ed è eccezionale nella progettazione di reti neurali e nello sviluppo di IA, grazie anche alla presenza di tantissime librerie dedicate, come ad esempio: TensorFlow, Keras, NumPy, Pandas, Py-Torch, Sklearn e altre. Infatti, una volta verificata la licenza d'uso e garantita la possibilità di utilizzo a livello aziendale, il mio modello è stato interamente creato con le architetture fornite da Keras nella libreria di TensorFlow. Le funzioni di Keras, di fatti, sono specializzate nello sviluppo di reti neurali in modo semplice e conciso, andando direttamente a definire - in primo luogo - la tipologia operativa del modello che si vuole implementare (funzionale, sequenziale, classificativo, regressivo ecc.) e, subito dopo, i vari layers commutativi che si vogliono aggiungere all'ossatura della rete tramite la direttiva "model.add", specificando poi tutti i vari parametri, adattati al proprio caso specifico. Come IDE di sviluppo, ho optato per l'utilizzo di Visual Studio Code di Microsoft - associato ad una compilazione ed esecuzione del software tramite terminale a riga di comando - molto flessibile e pratico nella gestione di ogni singolo linguaggio, installando i propri appositi pacchetti. A questo, è stato aggiunto

il supporto eseguibile con un ambiente virtuale appositamente installato, di modo che contenesse tutte le librerie specifiche che mi servivano.

- 2) Analisi dettagliata sul database a disposizione per individuare, tra tutte le varie colonne e tabelle, quelle inerenti al mio problema, di modo tale da effettuare una prima scrematura generale:

Questo passaggio ha comportato diverse difficoltà tecniche iniziali non indifferenti... infatti, in un'azienda così grande come SCM, solamente per la macchina nesting in questione, le colonne e tabelle del database di Maestro Active che avrebbero prodotto features significativi per il modello erano un numero importante. Tuttavia, grazie anche al mio responsabile, sono riuscito ad individuare principalmente due tabelle di interesse che, a loro volta, sarebbero state poi ulteriormente analizzate per reperire solo le colonne di dati strettamente necessarie. Le tabelle in questione sono:

- ***DbProductionEvents***: nello specifico, una tabella che contiene tutti i record relativi al programma di lavorazione (pgmx) in corso attuale indicato per la **produzione**. Le colonne scelte sono:

- *Id*: Identificativo univoco numerico del progetto di produzione attuale.
- *Start*: Colonna in formato timestamp che indica l'istante di inizio del processo di lavorazione in questione.
- *End*: Colonna in formato timestamp che indica l'istante di fine del processo di lavorazione in questione.

Entrambi i valori Start ed End sono stati utilizzati per mantenere attivo un ordine temporale tra i vari processi di lavorazione e poter generare quindi sequenze temporali specifiche tra un evento di manutenzione e l'altro.

- ***DbSpoilboardMaintenanceHistory***: tabella aggiuntiva, che contiene tutti i record inerenti alle operazioni di **manutenzione** avviate sul piano martire, registrate in ordine temporale. Le colonne prese in considerazione sono:

- *Id*: Identificativo univoco numerico del progetto di manutenzione attuale.
- *Timestamp*: Colonna in formato timestamp, che indica l'esatto istante in cui l'operazione di manutenzione correlata viene lanciata.
- *MaintenanceType*: Valore categorico che descrive la tipologia di operazione di manutenzione effettuata. Infatti, se MaintenanceType è uguale a 1, si è in procinto di un'operazione di rettifica del piano martire, se invece MaintenanceType è uguale a 0, si parla direttamente di un'operazione di sostituzione del piano stesso.
- *WearValue*: Valore numerico, molto importante, indicato in percentuale, che indica il livello di usura generale raggiunto dal piano martire della macchina, in seguito alle lavorazioni compiute, **prima** di eseguire l'operazione di manutenzione ad esso correlata.

Un'associazione logica fondamentale utilizzata, per ordinare temporalmente i dati, è stata la forte correlazione tra la colonna "Timestamp" di DbSpoilboardMaintenanceHistory e la colonna "End" di DbProductionEvents dove, sommando un gap temporale negativo o positivo prestabilito alla colonna "Timestamp", si è determinata la serie di lavorazioni in ordine di tempo che sono state effettuate prima dell'operazione relativa di manutenzione.

- 3) Ideazione del “tramite” opportuno per reperire tutti i dati sopraccitati dalle tabelle di interesse ed inserimento di essi in un’apposita struttura dati temporanea:

I dati di interesse, che sono stati poi successivamente agglomerati in un dataframe da passare come input alla rete neurale, erano inizialmente residenti su un database relazionale di tipologia SQLite, associato direttamente al software Maestro Active che risiedeva su ogni macchina distinta, a cui, naturalmente, non avevo diretto accesso in sede progettuale per via delle crittografie di sicurezza aziendali. Oltre tutto, i dati in possesso dell’azienda era comunque esigui, poiché tale tipologia di macchina non gestiva un recording completo dei dati di produzione e manutenzione in archivio. Per cui ho deciso di agire contemporaneamente con due modalità differenti:

1. *OFFLINE*: Ho studiato le dinamiche dei dati che si sarebbero dovuti prendere in considerazione durante il training del modello e ho cercato di riprodurli - in maniera simulata ma coerente con le specifiche - in quantità considerevole per sostenere un addestramento di un modello ricorrente di deep learning. Per fare ciò, insieme al mio tutor aziendale, ho analizzato l’andamento della distribuzione della produzione e della manutenzione di alcuni possibili utenti della macchina nesting in questione e ho ideato una funzione Python che andasse a generare un dataset di default, di dimensione variabile, con tutti i vincoli e caratteristiche del caso. Di seguito, il codice della mia funzione di generazione del dataset simulato:

```
def simulate_lifecycle_dataset(
    total_maintenance_records: int,
    prod_block_range=(450, 550),
    rettifica_wear_range=(0.7, 0.8),
    sostituzione_wear_range=(0.9, 1.0),
    maint_time_range_days=(2, 3),
    cycle_length_range=(25, 35),
    prod_duration_sec_range=(1, 10)
):
    records = []
    now = datetime.now()
    # calcolo un punto di partenza che copra tutti i giorni
    max_total_days = total_maintenance_records * maint_time_range_days[1] * 1.2
    previous_maint_time = now - timedelta(days=max_total_days)

    def make_maintenance(is_substitution: bool):
        nonlocal previous_maint_time
        # 1) quando succede la manutenzione
        delta_days = random.uniform(*maint_time_range_days)
        current_maint_time = previous_maint_time + timedelta(days=delta_days)
        # 2) blocchi di produzione
        prod_count = random.randint(*prod_block_range)
        for _ in range(prod_count):
            frac = random.uniform(1e-6, 1 - 1e-6)
            prod_time = previous_maint_time + (current_maint_time -
previous_maint_time) * frac
            records.append({
                "prod_time": prod_time,
                "current_maint_time": current_maint_time,
                "previous_maint_time": previous_maint_time,
                "is_substitution": is_substitution
            })
        previous_maint_time = current_maint_time
    for _ in range(total_maintenance_records):
        if random.uniform(0, 1) < sostituzione_wear_range[0]:
            make_maintenance(is_substitution=True)
        else:
            make_maintenance(is_substitution=False)
```

```

records.append({
    "Timestamp": prod_time,
    "Start": prod_time,
    "End": prod_time +
timedelta(seconds=random.uniform(*prod_duration_sec_range)),
    "EventType": 1,
    "MaintenanceType": -1,
    "WearValue": 0,
    "WorkingHours": 0,
    "TotalEvents": 1
})
# 3) evento manutentivo
mtype = 0 if is_substitution else 1
wear_rng = sostituzione_wear_range if is_substitution else
rettifica_wear_range
working_hours = (current_maint_time -
previous_maint_time).total_seconds() / 3600.0
records.append({
    "Timestamp": current_maint_time,
    "Start": 0,
    "End": 0,
    "EventType": 0,
    "MaintenanceType": mtype,
    "WearValue": round(random.uniform(*wear_rng), 3),
    "WorkingHours": round(working_hours, 2),
    "TotalEvents": prod_count
})
previous_maint_time = current_maint_time

maint_events_generated = 0
min_cycle, max_cycle = cycle_length_range

# finché non raggiungiamo il numero totale di manutenzioni richieste
while maint_events_generated < total_maintenance_records:
    remaining = total_maintenance_records - maint_events_generated
    # estraiamo la lunghezza del prossimo ciclo
    this_cycle_len = random.randint(min_cycle, max_cycle)

    if remaining >= this_cycle_len:
        # ciclo "completo": this_cycle_len-1 rettifiche + 1 sostituzione
        for i in range(this_cycle_len - 1):
            make_maintenance(is_substitution=False)
        maint_events_generated += 1

```

```

        make_maintenance(is_substitution=True)
        maint_events_generated += 1
    else:
        # ciclo "incompleto": solo le rettifiche rimaste, senza sostituzione
        for i in range(remaining):
            make_maintenance(is_substitution=False)
            maint_events_generated += 1
        break # abbiamo finito

    # mettiamo in ordine cronologico
    df = pd.DataFrame(records)
    df.sort_values(by=["Timestamp", "EventType"], ascending=[True, False],
    inplace=True)
    df.reset_index(drop=True, inplace=True)
    return df

```

Praticamente, tale funzione prende in ingresso diversi parametri:

- *Total_maintenance_record*: numero totale di eventi manutentivi (Rettifica o Sostituzione) che compongono il dataset.
- *Prod_block_range*: range numerico randomico di eventi produttivi tra un'operazione di manutenzione e la sua consecutiva/precedente che riempie gli archi temporali tra un evento manutentivo e il suo consecutivo.
- *Rettifica_wear_range & Sostituzione_wear_range*: valori di WearValue raggiunto dal piano martire al momento dell'avvenuta rettifica o sostituzione. Non utilizzato direttamente come dato deterministico nel training, ma molto utile per poter delineare una struttura realistica del dataset e dell'andamento dell'usura del piano martire stesso in funzione dei blocchi di produzione in corso (una rettifica avviene dopo una certa soglia di usura, stessa cosa per una sostituzione).
- *Maint_time_range_days*: il range massimo, scelto randomicamente, dopo quanti giorni al massimo viene effettuata un'operazione manutentiva di rettifica.
- *Cycle_length_range*: variabile molto importante, che genera un range randomico massimo di lunghezza di una serie temporale di eventi di rettifica prima di una sostituzione finale. Questo consente di avere una flessibilità notevole, poiché il modello si trova a dover apprendere eventi manutentivi a step temporali diversi tra loro e, di conseguenza, non si adatta a schemi ripetitivi e fissi, ma deve imparare realmente a prevedere il prossimo evento nella variabilità delle lunghezze.
- *Prod_duration_sec_range*: range randomico di tempo intercorso tra un'operazione di produzione e la sua successiva/precedente.

In questo modo, con una semplice iterazione, si è creato completamente da zero e in pochi secondi, un dataset intero simulato - in stile dataframe di Pandas - sulla base dei valori numerici inseriti in input alla funzione e su uno scheletro di possibili andamenti realistici di clienti con abitudini diversificate tra loro, a seconda delle esigenze specifiche. Inoltre, ho garantito anche la possibilità di poter gestire dei **cicli incompleti**, ovvero simulare delle situazioni particolari in cui non si era ancora arrivati in tempo reale alla fine del blocco produttivo in questione e, di fatto, la manutenzione successiva che scandisce i blocchi di

produzione, non era ancora stata effettuata. Si approfondirà successivamente il concetto di ciclo e sottociclo di vita del piano martire.

- **ONLINE:** Parte derivata successivamente dalla modalità “offline” di creazione di dati personalizzati. Nella pratica, ci si è immedesimati nella parte del cliente vero e proprio, facendo eseguire da terminale, lavorazioni programmate in loop sul software Maestro Active, le quali sono state poi registrate anche nel database principale e alternate alle varie manutenzioni lanciate tramite simulazione. In questo modo, è stato possibile emulare completamente il comportamento che, ipoteticamente, avrebbe potuto attuare un qualsiasi cliente X con l'utilizzo della macchina e, così facendo, l'intelligenza artificiale avrebbe appreso da tali comportamenti.

Alle colonne già esistenti nella logica di Maestro Active, ho aggiunto delle colonne ulteriori determinate a posteriori, come si può osservare nell'output seguente mostrato a terminale:

Dati simulati - Tail:					
	Timestamp	Start	\		
501312	2022-06-01 21:22:39.242996	2022-06-01 21:22:39.242996			
501313	2022-06-01 21:28:23.382257	2022-06-01 21:28:23.382257			
501314	2022-06-01 21:30:31.242696	2022-06-01 21:30:31.242696			
501315	2022-06-01 21:41:05.429909	2022-06-01 21:41:05.429909			
501316	2022-06-01 21:42:10.768376	0			
	End	EventType	MaintenanceType	WearValue	\
501312	2022-06-01 21:22:48.392192	1	-1	0.000	
501313	2022-06-01 21:28:25.966167	1	-1	0.000	
501314	2022-06-01 21:30:37.471376	1	-1	0.000	
501315	2022-06-01 21:41:07.337448	1	-1	0.000	
501316	0	0	1	0.741	
	WorkingHours	TotalEvents			
501312	0.00	1			
501313	0.00	1			
501314	0.00	1			
501315	0.00	1			
501316	57.94	501			

Ho determinato le colonne:

- TotalEvents: numero cumulativo di eventi di produzione tra un'operazione manutentiva e la consecutiva.
- WorkingHours: numero cumulativo dei Timestamp, convertiti in ore, di produzione tra un'operazione manutentiva e la consecutiva.
- EventType: variabile binaria che se settata a 0 indica un evento manutentivo e se settata a 1 indica un evento produttivo. In questo modo si riesce fin da subito a scremare il dataframe tra le due macro-tipologie di evento riscontrabile nel problema.

Questi dati extra generati hanno assunto rilevante importanza per poter effettuare la corretta distinzione tra evento produttivo ed evento manutentivo e, soprattutto, per dare la possibilità concreta di poter determinare il numero di eventi di produzione ancora intraprendibili prima di ogni evento manutentivo, analogamente

anche, alle relative ore di lavoro ancora rimaste prima di dover forzatamente effettuare una presunta manutenzione.

Di grande aiuto sono stati, senza dubbio, i vari datetime di ogni singolo evento (produzione e manutenzione), poiché si è sfruttato l'ordine cronologico per ricreare uno storico ordinato di tutti i vari eventi, su cui è stata poi implementata la logica che descriverò successivamente. Di fatto, l'idea principale è stata la seguente:

"Tutti i progetti di lavorazione eseguiti, in ordine temporale, prima dell'evento di manutenzione, possono essere categorizzati in vari blocchi, scanditi dall'evento di manutenzione stesso. Ciò significa che, tramite il "Timestamp" di ogni manutenzione, si potrà ottenere la suddivisione precisa di ogni blocco di lavorazioni ordinato temporalmente suddiviso dalle varie operazioni manutentive stesse."

Chiaramente, come affermato precedentemente, i dati di interesse provenivano da tabelle diverse nel database reale, per cui, a livello semantico e pratico, con i miei responsabili ho deciso di non dipendere direttamente dal database stesso, ma mantenere comunque coerentemente la logica delle due tabelle distinte per facilitare il debug e i vari test da convalidare a monte. Così facendo, abbiamo deciso di gestire i dati in ingresso al mio script con una struttura vettoriale di estensione .json. In questo modo, il mio codice non si sarebbe dovuto mai interfacciare con il database e si sarebbero evitati tutti i protocolli di comunicazione interni, gestendo direttamente l'invio dei dati di interesse al mio script per mezzo di un server ReST appositamente sviluppato, di cui parlerò però successivamente. Per ora, ci basti sapere che la lista .json di ingresso conteneva esattamente tutti i campi desiderati per la costruzione del DataFrame unico, ovvero una struttura dati specifica, messa a disposizione dalla libreria Pandas di Python. Per la parte offline invece, il tutto era già sostenuto dalla funzione di simulazione del dataset, la quale si occupava di generare i dati e ordinarli automaticamente secondo la logica richiesta. Mentre, per quanto riguarda la parte online e, quindi, l'accesso al contenuto .json, era necessario amalgamare i dati grezzi, della lista di ingresso, nella logica effettiva del problema. Proprio per questo, ho ideato la funzione seguente:

```
def merge_data_list(events: list) -> pd.DataFrame:
    # separa prima gli eventi produttivi
    prod_events = [e for e in events if e.get("EventType")==1]
    # qui sfruttiamo esattamente la tua riga
    df_prod = pd.DataFrame(prod_events, columns=['Start', 'End'])
    df_prod['Timestamp'] = pd.to_datetime(df_prod['End'], errors='coerce')
    df_prod['EventType'] = 1
    df_prod['TotalEvents'] = df_prod.get('TotalEvents', 1) \
    if prod_events and "TotalEvents" not in df_prod.columns else
        df_prod['TotalEvents']
    df_prod = df_prod[['Timestamp', 'EventType', 'TotalEvents']]
    # poi i manutentivi
    spoil_events = [e for e in events if e.get("EventType")==0]
    df_spoil = pd.DataFrame(spoil_events,
                            columns=['Timestamp', 'MaintenanceType', 'WearValue'])
```

```

df_spoil['Timestamp'] = pd.to_datetime(df_spoil['Timestamp'],
errors='coerce')
df_spoil['EventType'] = 0
df_spoil['TotalEvents'] = 0
df_spoil=df_spoil[['Timestamp','EventType','MaintenanceType','WearValue','TotalEvents']]
df = pd.concat([df_prod, df_spoil], ignore_index=True)
df = df.sort_values(['Timestamp','EventType'], ascending=[True,False])\
    .reset_index(drop=True)
return df

```

Tale funzione mantiene distinti, logicamente e semanticamente, i concetti di Produzione e Manutenzione, generando inizialmente due dataframe separati (utili per effettuare determinati test sui dati o debug mirato prima dell'unione in struttura unica) che vengono poi amalgamati e ordinati, in ordine crescente, prendendo in considerazione il campo chiave “Timestamp”. Per comodità, siccome il campo “Timestamp” è utilizzato nell'effettivo solo dalla tabella DbSpoilboardMaintenanceHistory, si è associato a DbProductionEvents - che al suo interno non presenta tale colonna in origine - “End” = “Timestamp”. In questo modo, la colonna “Timestamp” finale viene completamente e interamente popolata:

- Con valore “Timestamp” originale per i record relativi alle manutenzioni
- Con valore della colonna “End” per i record relativi alle produzioni

A livello logico, la scelta è stata perfettamente giustificata dal fatto che, sicuramente, una data di fine di un’operazione di produzione, viene sicuramente collocata temporalmente prima rispetto al “Timestamp” effettivo di un’operazione di manutenzione... per cui, in questo modo, si è ottenuto un dataframe unico di blocchi di produzione scanditi dal relativo prossimo evento di manutenzione imminente. Di seguito, uno screenshot, catturato dal mio terminale, di come appare la tail del dataset completamente unificato (in questo caso però originato tramite funzione di simulazione) e pronto per essere pre-processato:

Dati simulati - Tail:						
		Timestamp		Start	\	
499287	2022-05-24	13:24:17.148957	2022-05-24	13:24:17.148957		
499288	2022-05-24	13:33:05.489806	2022-05-24	13:33:05.489806		
499289	2022-05-24	13:37:28.780203	2022-05-24	13:37:28.780203		
499290	2022-05-24	13:48:29.964972	2022-05-24	13:48:29.964972		
499291	2022-05-24	13:49:16.966815		0		
		End	EventType	MaintenanceType	WearValue	\
499287	2022-05-24	13:24:22.919310		1	-1	0.000
499288	2022-05-24	13:33:09.539791		1	-1	0.000
499289	2022-05-24	13:37:36.699979		1	-1	0.000
499290	2022-05-24	13:48:39.768055		1	-1	0.000
499291		0		0	1	0.742
		WorkingHours	TotalEvents			
499287		0.00	1			
499288		0.00	1			
499289		0.00	1			
499290		0.00	1			
499291		68.19	453			

Come si può ben notare dall'immagine, i dati relativi alle produzioni avranno “End” = “Timestamp” e, allo stesso modo, valori inerenti alla tabella DbSpoilboardMaintenanceHistory a valori di default (-1, 0, oppure 1 a seconda del senso logico della colonna) per tutte le colonne che non sono rilevanti nell'ambito della produzione stessa. Infatti, TotalEvents = 1 per tutti gli eventi di produzione, indicando che ogni produzione vale solamente 1 TotalEvents; WorkingHours e WearValue = 0, in quanto sono record relativi solo alla manutenzione e MaintenanceType = -1 per avere un valore di default all'infuori della logica di 0 e 1 delle due classi possibili per MaintenanceType stesso. Analogamente, lo stesso avviene per i record manutentivi, che hanno i valori di “Start” ed “End” (inerenti soltanto ai record di produzione) uguali a 0, poiché l'unico DateTime significativo è proprio “Timestamp”.

3.2 Pre-processing

3.2.1 Features Engineering

Successivamente, si è passati alla fase di pre-processing di questa struttura unificata... la quale, secondo la mia opinione, è stata la fase più tecnicamente complessa di tutto il progetto. Non è stato subito chiaro come e quali tecniche utilizzare per pre-processare correttamente il dataframe ottenuto, in modo tale da avere una discreta linearità nei dati da introdurre nella rete e, perciò, si è dovuta fare una ricerca documentativa online e brainstorming collettivo tra colleghi e professori, per arrivare ad un dunque che potesse essere sensato. Sicuramente da una parte, dopo aver ordinato i dati secondo "Timestamp" per attribuire un senso alle sequenze temporali, si dovevano seguire le linee guida standard del pre-processing di un dataframe, ovvero l'imputazione di tutti i valori mancanti con un valore di default, la normalizzazione e la deviazione standard delle variabili numeriche, lo scaling delle variabili di tipo DateTime, l'applicazione di Oversampling o Undersampling se il dataset fosse stato sbilanciato su una classe in particolare, infine, la divisione del dataset in features X, ovvero le caratteristiche che avrebbero aiutato la rete a predire il target, e il target stesso Y, cioè la variabile o valore che la rete avrebbe dovuto prevedere e gestire durante le varie epoche di addestramento sui dati di validazione e, successivamente, su dati di test mai visti prima durante l'addestramento stesso. D'altra parte, però, era assolutamente necessario gestire tutto il discorso legato al **features engineering** poiché, in un problema così densamente complesso a livello logico, le informazioni apprese dai dati iniziali e dalle features correnti non erano sufficienti ad un robusto training della rete.

I dati collezionati fino a questo momento erano, per l'appunto, tutti quelli appartenenti al dataframe iniziale e nient'altro; per cui, quale strategia si poteva mettere in atto per garantire un'indicazione più precisa e reperire maggiori dettagli?

Senza ombra di dubbio, era imperativo partire dagli elementi in possesso e generare, tramite formule matematiche e associazioni logiche, nuove informazioni rilevanti.

Il primo step importante è stato l'implementazione di una logica che andasse a raggruppare separatamente le operazioni di rettifica e di sostituzione, in modo tale da riuscire a determinare, poi, tutti le features derivanti e

gestire, quindi, i due concetti in modo parallelo. Per fare ciò, ho realizzato due maschere di supporto, determinate da EventType, per gestire i due concetti principali introdotti: Produzione e Manutenzione e, grazie all'utilizzo di esse, ho realizzato due sottogruppi principali distinti, dove ho raccolto temporaneamente entrambe le tipologie di eventi:

```
# Maschere per distinguere record di manutenzione e produzione
mask_maint = df['EventType'] == 0 # record manutentivi
mask_prod = df['EventType'] == 1 # record produttivi
# --- Calcolo dei relativi gruppi ---
# Nei record manutentivi: calcolo dei gruppi cumulativi in base al tipo
df.loc[mask_maint, 'Group_Rett'] = (df.loc[mask_maint, 'MaintenanceType'] == 1).cumsum()
df.loc[mask_maint, 'Group_Sost'] = (df.loc[mask_maint, 'MaintenanceType'] == 0).cumsum()
# Nei record produttivi, i gruppi devono essere -1 perché no manutenzione
df.loc[mask_prod, 'Group_Rett'] = -1
df.loc[mask_prod, 'Group_Sost'] = -1
```

Questo schema logico e l'effettiva creazione di Group_Rett e Group_Sost è stato fondamentale nella semplificazione del calcolo di ogni feature successiva appartenente ai due gruppi distinti.

Innanzitutto, a partire dal “Timestamp” delle varie manutenzioni a disposizione e, grazie alla funzione di pandas: **df.loc**, che consente di manipolare il dataframe originario, ho deciso di calcolare due features molto importanti che sono state determinanti nella risoluzione del problema. Esse erano:

- **T_Rettifica**: arco temporale in ore che intercorre tra due operazioni di rettifica consecutive tra loro.
- **T_Sostituzione**: arco temporale in ore che intercorre tra due operazioni di sostituzione consecutive tra loro.

```
# --- Calcolo dei tempi ---
# Per i record manutentivi, calcola T_Rettifica e T_Sostituzione separatamente
mask_rett = mask_maint & (df['MaintenanceType'] == 1)
mask_sost = mask_maint & (df['MaintenanceType'] == 0)
# Calcola T_Rettifica: differenza tra il Timestamp della rettifica e quello della precedente rettifica
df.loc[mask_rett, 'T_Rettifica']=df.loc[mask_rett, 'Timestamp'].diff().dt.total_seconds() / 3600
# Se il primo record del gruppo non ha diff, usa la differenza rispetto all'inizio del ciclo (t0)
df.loc[mask_rett, 'T_Rettifica']=df.loc[mask_rett, 'T_Rettifica'].fillna((df.loc[mask_rett, 'Timestamp'] - t0).dt.total_seconds() / 3600)
# Calcola T_Sostituzione per i record di sostituzione
df.loc[mask_sost, 'T_Sostituzione']=df.loc[mask_sost, 'Timestamp'].diff().dt.total_seconds() / 3600
```

```

df.loc[mask_sost, 'T_Sostituzione']=df.loc[mask_sost, 'T_Sostituzione'].fillna((df.loc[mask_sost, 'Timestamp'] - t0).dt.total_seconds() / 3600)
# Nei record manutentivi, azzera il tempo dell'altro tipo
df.loc[mask_rett, 'T_Sostituzione'] = 0
df.loc[mask_sost, 'T_Rettifica'] = 0
# Nei record produttivi, forza entrambi i tempi a 0
df.loc[mask_prod, ['T_Rettifica', 'T_Sostituzione']] = 0

```

Dopodiché, è stato fondamentale andare a gestire le **ore effettive di lavoro** tra un tipo di operazione di manutenzione e l'altra, a partire dall'utilizzo della colonna “**WorkingHours**”. La relazione principale da tenere in considerazione per il mio problema era la seguente proporzione:

Ore di lavoro / N. lavorazioni eseguite <=> Manutenzioni Effettuate

La quale significava esattamente che le manutenzioni effettuate non erano altro che il **rappporto** tra le ore di lavoro totali e il numero esatto di lavorazioni eseguite. Quindi, come si può ben dedurre, le colonne TotalEvents e WorkingHours sono state di importanza vitale nel dataframe di input.

Da questo punto in poi, l'ingegnerizzazione delle features è stata effettuata in maniera distinta tra i due macro-gruppi, per mezzo delle colonne create precedentemente. Siccome si era a conoscenza del totale delle lavorazioni eseguite tra un'operazione di manutenzione qualsiasi e l'altra, era utile essere al corrente anche di quante lavorazioni potessero intercorrere tra una rettifica e l'altra e, analogamente, di quante lavorazioni potessero intercorrere tra una sostituzione e l'altra. Questi dati sono stati sintetizzati tramite l'ausilio della colonna **TotalEvents** come:

```

# --- Calcolo dei Tot_Events ---
    # Per i record manutentivi di rettifica:
    # Tot_Events_Rett = massimo di TotalEvents nel gruppo di rettifica
    df.loc[mask_maint & (df['MaintenanceType'] == 1), 'Tot_Events_Rett'] = \
        df.loc[mask_maint & (df['MaintenanceType'] == 1)].groupby('Group_Rett')\
            ['TotalEvents'].transform('max')
    # Per i record manutentivi di sostituzione:
    # Tot_Events_Sost = il valore di CumulativeEvents (cioè il totale delle
    # lavorazioni accumulate fino alla sostituzione)
    df.loc[mask_maint & (df['MaintenanceType'] == 0), 'Tot_Events_Sost'] = \
        df.loc[mask_maint & (df['MaintenanceType'] == 0)].groupby('Group_Sost')\
            ['CumulativeEvents'].transform('max')
    # Settiamo rispettivamente a 0 i valori di Sostituzione nella Rettifica e
    # viceversa
    df.loc[mask_maint & (df['MaintenanceType'] == 1), 'Tot_Events_Sost'] = 0
    df.loc[mask_maint & (df['MaintenanceType'] == 0), 'Tot_Events_Rett'] = 0
    # Nei record produttivi, i Tot_Events relativi alla manutenzione devono
    # essere 0
    df.loc[mask_prod, ['Tot_Events_Rett', 'Tot_Events_Sost']] = 0

```

Dove si è gestito sempre il concetto di maschera e anche l'imputazione dei valori mancanti possibili. Analogamente, sono stati determinati, per mezzo della colonna **WorkingHours**, le relative tempistiche di ognuna delle due categorie manutentive nei due tempi specifici: **T_Rettifica** e **T_Sostituzione**, ovvero rispettivamente il tempo che intercorre tra un'operazione di rettifica e la sua consecutiva e il tempo che intercorre tra un'operazione di sostituzione e la sua consecutiva.

```
# --- Calcolo dei tempi ---
# Per i record manutentivi, calcola T_Rettifica e T_Sostituzione separatamente
    mask_rett = mask_maint & (df['MaintenanceType'] == 1)
    mask_sost = mask_maint & (df['MaintenanceType'] == 0)
# Calcola T_Rettifica: differenza tra il Timestamp della rettifica e quello della precedente rettifica
    df.loc[mask_rett, 'T_Rettifica']=df.loc[mask_rett, 'Timestamp'].diff().dt.total_seconds() / 3600
# Se il primo record del gruppo non ha diff, usa la differenza rispetto all'inizio del ciclo (t0)
    df.loc[mask_rett, 'T_Rettifica'] = df.loc[mask_rett, 'T_Rettifica'].fillna(
        (df.loc[mask_rett, 'Timestamp'] - t0).dt.total_seconds() / 3600
    )
    # Calcola T_Sostituzione per i record di sostituzione
    df.loc[mask_sost, 'T_Sostituzione']=df.loc[mask_sost, 'Timestamp'].diff().dt.total_seconds() / 3600
    df.loc[mask_sost, 'T_Sostituzione']=df.loc[mask_sost, 'T_Sostituzione'].fillna((df.loc[mask_sost, 'Timestamp'] - t0).dt.total_seconds() / 3600)
```

3.2.2 Imputazione dei valori mancanti

Le reti neurali e, più precisamente, i dataframe su cui esse lavorano, necessitano di varie operazioni di pulizia e riempimento dei valori mancanti o indefiniti (NaN) per poter essere processati in maniera consona. Questo perché è essenziale avere a disposizione valori finiti che l'architettura neurale possa prendere in considerazione come quantità misurabili e deducibili. Per tale motivo, dopo aver ingegnerizzato le features inerenti al problema, è stato imprescindibile formulare una funzione di imputazione dei valori mancanti, che andasse a sopperire a tale ostacolo. Il codice che ho scritto:

```
def missing_values(df):
    df['Timestamp'] = df['Timestamp'].astype(str)
    numerical_cols = ['WearValue', 'TotalEvents', 'WorkingHours']
    temporal_cols = ['Timestamp']
    categorical_cols = ['MaintenanceType']
    # definizione degli imputers da usare
    imputer_mean = SimpleImputer(strategy='mean')
    imputer_frequent = SimpleImputer(strategy='most_frequent')
```

```

imputer_constant = SimpleImputer(strategy='constant', fill_value=-1)
# numerical_cols
for col in numerical_cols:
    imputer_mean.fit(df[[col]])
    df.loc[:, col] = imputer_mean.transform(df[[col]])
# categorical_cols
for col in categorical_cols:
    imputer_constant.fit(df[[col]])
    df.loc[:, col] = imputer_constant.transform(df[[col]])
# temporal_cols
for col in temporal_cols:
    imputer_frequent.fit(df[[col]])
    df.loc[:, col] = imputer_frequent.transform(df[[col]])
return df

```

Ho quindi creato la funzione “missing_values”, la quale si è occupata di prendere in ingresso l’intero dataframe grezzo e ne ha imputato tutti i valori mancanti in ogni colonna. Come logica di imputazione, ho scelto di utilizzare quella messa a punto dalla libreria “scikit-learn” di Python, la quale fornisce uno strumento molto utile, per questo tipo di problematiche, denominato SimpleImputer:

Classe `sklearn.impute.SimpleImputer(*, missing_values=nan, strategy='mean', fill_value=None, copy=True, add_indicator=False, keep_empty_features=False)`

Esso corrisponde ad una funzione di imputazione uni-variata per completare i valori mancanti con semplici strategie. Sostituisce i presenti valori mancanti utilizzando una statistica descrittiva (ad es. media, mediana o elemento più frequente) lungo ogni colonna, o utilizzando un valore costante da definire manualmente a priori.

I parametri che la medesima funzione vuole in ingresso sono principalmente due:

a) **missing_values** : `int, float, str, np.nan, None o pandas. NA, predefinito=np.nan`

Segnaposto per i valori mancanti. Rappresenta tutte le occorrenze da imputare. Per i DataFrame Pandas con nullable integer dtype con valori mancanti, può essere impostato su

`.missing_values missing_values np.nan pd.NA`

b) **strategy: str o Callable, default='mean'**

La strategia effettiva dell'imputazione.

- Se "**mean**", si sostituiscono i valori mancanti usando la media dei valori di ogni colonna. Può essere utilizzato solo con dati numerici.
- Se "**median**", sostituisce i valori mancanti usando la mediana lungo i valori di ogni colonna. Può essere utilizzato solo con dati numerici.

- Se "**most frequent**", si sostituiscono i valori mancanti utilizzando il più frequente tra essi, considerando tutta la colonna. Può essere utilizzato con stringhe o dati numerici. Se sono presenti più valori di questo tipo, viene restituito solo il più piccolo tra tutti.
- Se "**constant**", si sostituiscono i valori mancanti con "fill_value=k". Dove "k" può essere utilizzato con stringhe o dati numerici e assegnato quindi ad un valore fisso hard-coded.

Nel mio caso specifico, ho riempito i valori mancanti utilizzato tre tipologie di imputazione:

1. “Mean” per gestire tutti i valori numerici nelle relative colonne omonime.
2. “Constant” per i valori categorici, utilizzando, come fill_value, il valore ‘-1’ poiché, se la variabile di MaintenanceType non fosse stata presente, sarebbe significato che sicuramente non c’era stata alcuna operazione di manutenzione e quindi il valore sarebbe stato settato a ‘-1’ per indicarlo.
3. “Most_frequent” per i valori DateTime, convertendoli dapprima in formato stringa (poiché non si può, per ragioni tecniche dell’algoritmo, applicare l’imputer sul formato originale Timestamp) per generare l’imputer effettivo e ri-convertirli successivamente in formato Timestamp per la loro utilità come date ordinarie nel dataframe stesso.

3.2.3 Encoding

Dopo essere riuscito correttamente a gestire il riempimento dei valori mancanti, è venuto il momento di dirigere il nucleo vero e proprio di un’attività di pre-processing di un dataset, ovvero l'**encoding** delle colonne numeriche, categoriche e temporali presenti; per poi ricorrere, successivamente, alla suddivisione del dataframe in Features X e Target Y e nel relativo splittaggio ulteriore in Training Set (da cui si è ricavato anche il Validation Set da utilizzare in addestramento) e Test Set, del tipo: X_Train, X_Test, Y_Train, Y_Test. I passaggi relativi alla codifica delle colonne del dataframe a disposizione sono stati:

- **Conversione e ordinamento:** La colonna “Timestamp” è stata ri-convertita nuovamente in formato DateTime, per poterla utilizzare nel features engineering e come parametro guida nella creazione di sequenze paddate (derivanti dal dataframe) di lunghezza uniforme da utilizzare nell’input. Inoltre, il dataframe è stato ri-ordinato (resetando anche i propri indici in modo crescente), tramite la funzione di “sort”, secondo la medesima colonna, poiché vi era l’occorrenza di dare alla luce sequenze temporali, basate sulle operazioni di manutenzione eseguite, per gestire tale dipendenza a livello neurale.

```
df['Timestamp'] = pd.to_datetime(df['Timestamp'], errors='coerce')
# Ordinamento DataFrame in base a Timestamp per creare sequenza temporale
df = df.sort_values(by='Timestamp').reset_index(drop=True)
```

- **Gestione delle intrinsecità delle categorie manutentive:** Dato che il dataframe è stato organizzato con una base logica duale sulle rispettive classi manutentive Rettifica e Sostituzione, anche le features derivate sono state, naturalmente, raggruppate in base ad esse. Proprio per questo, si è

dovuta utilizzare una logica di azzeramento dei record, inerenti alle manutenzioni, nei pressi delle lavorazioni e, analogamente, di quelli inerenti alle produzioni nei pressi delle manutenzioni. Questo per il semplice motivo che, anche se il dataframe è stato unificato, la logica semantica è rimasta distinta al suo interno. Lo stesso identico discorso è valso anche per la distinzione tra le due tipologie di evento interne al dataset: EventType = 0: Manutenzione, EventType = 1: Produzione. Di seguito, il codice con cui ho gestito questo aspetto:

```
# Nei record manutentivi, azzerà il tempo dell'altro tipo
df.loc[mask_rett, 'T_Sostituzione'] = 0
df.loc[mask_sost, 'T_Rettifica'] = 0
# Nei record produttivi, forza entrambi i tempi a 0
df.loc[mask_prod, ['T_Rettifica', 'T_Sostituzione']] = 0
# Nei record manutentivi, forzaggio di Start ed End a 0 perché non esistono
df.loc[mask_maint, ['Start', 'End']] = pd.NaT
# Nei record produttivi, tutti i campi relativi alla manutenzione (eccetto
gli id di ciclo) devono essere 0
df.loc[mask_prod, ['WearValue', 'WorkingHours', 'T_Rettifica',
                   'T_Sostituzione', 'Tot_Events_Rett', 'Tot_Events_Sost']] = 0
# Il MaintenanceType nei record produttivi è -1
df.loc[mask_prod, 'MaintenanceType'] = -1
```

3.2.4 Normalizzazione

Per fare in modo che qualunque architettura di Deep Learning impari correttamente a gestire i valori numerici, è quasi sempre opportuno effettuare una normalizzazione o una standardizzazione delle colonne numeriche. Questo procedimento serve per traslare tutti i valori in un range ben definito – solitamente tra 0 e 1 – ed evitare che durante la computazione del gradiente in addestramento, la rete veda troppi outlier e valori fuori scala. Le tecniche utilizzate principalmente sono due e non necessariamente devono essere utilizzate insieme:

- **Standardizzazione:** si applica utilizzando la funzione StandardScaler() di Python. Pone ogni variabile numerica su una distribuzione con media = 0 e deviazione standard = 1. Mantiene la forma della distribuzione gaussiana ed è utile quando gli algoritmi assumono, spesso, dati centrati attorno allo zero. È inoltre resistente ad outlier moderati perché la scala è basata su deviazione standard.
- **Normalizzazione:** si applica utilizzando la funzione MinMaxScaler() di Python. Riconduce i valori di una feature in un intervallo prestabilito, tipicamente tra 0 e 1 o tra -1 e 1. Assicura che tutte le features siano normalizzate nello stesso range. Questo è fondamentale per algoritmi con funzioni di attivazioni che saturano i vari intervalli. Molto sensibile però agli outlier, infatti un valore anomalo può spostare l'intervallo stesso e soffocare gli altri punti.

Quando usare l'una o l'altra tecnica dipende considerevolmente dalla presenza di outlier nei dati di ingresso e dall'algoritmo che si vuole utilizzare per risolvere il problema in questione. Io, per questo problema, ho deciso di utilizzare la standardizzazione sulle features di input con la funzione di StandardScaler() e una normalizzazione con MinMaxScaler() sui target di output, di modo tale da renderli uniformi durante la

predizione da parte della rete, ed evitare scalini o salti fuori luogo. Seguendo questo approccio, ho riscontrato buoni risultati in fase di addestramento ed inferenza finale. La funzione con cui ho realizzato il tutto è la seguente:

```
numeric_feature_indices = [2, 3, 4, 5, 7, 8, 9, 10, 11]
def normalize_fn(features, target):
    features = tf.cast(features, tf.float32)
    batch_size = tf.shape(features)[0]

    # normalizzazione delle feature numeriche
    features_numeric = tf.gather(features, numeric_feature_indices, axis=1)
    mean = tf.cast(scaler_numeric.mean_, tf.float32)
    std = tf.cast(scaler_numeric.scale_, tf.float32)
    norm_numeric = (features_numeric - mean) / (std + 1e-8)

    # ricostruisco il tensore completo
    row_idx = tf.range(batch_size)
    col_idx = tf.constant(numeric_feature_indices, dtype=tf.int32)
    row_t = tf.reshape(tf.tile(tf.expand_dims(row_idx, 1), [1, tf.shape(col_idx)[0]]), [-1])
    col_t = tf.tile(col_idx, [batch_size])
    indices = tf.stack([row_t, col_t], axis=1)
    feats_norm = tf.tensor_scatter_nd_update(features,
                                              indices,
                                              tf.reshape(norm_numeric, [-1]))
    # target: non tocchiamo i primi due canali (già in [0,1]) né il canale
    # classificativo
    return feats_norm, target
```

Che viene chiamata poi da un'altra funzione di **custom_map**, la quale va a gestire anche la normalizzazione dei target stessi:

```
def custom_map_fn(x, y):
    # Normalizza features numeriche e target numerici
    x_norm, y_norm = normalize_fn(x, y)

    # Per la parte regressiva (many-to-many) prendo tutti i timestep
    y_rett = y_norm[:, 0:1]
    y_sost = y_norm[:, 1:2]

    # Per la parte classificativa (many-to-one) prendo solo l'ultimo timestep
```

```

    # Se il target è -1, verrà comunque passato (e la loss personalizzata lo
    ignorerà)
    y_class = y_norm[-1:, 2:3]
    return x_norm, (y_rett, y_sost, y_class)

```

Chiaramente, solo le features numeriche vanno normalizzate o standardizzate; per questo è opportuno identificarle e raccoglierne gli indici in una lista, numeric_feature_indices, che viene passata alla funzione di normalizzazione stessa, per chiarire su chi deve essere applicata tale trasformazione. Infine, nel main della pipeline, il tutto viene rappresentato così:

```

numeric_idx = numeric_feature_indices # es. [2,3,4,5,7,8,9,10,11]
all_features = np.concatenate(seq_x_tr, axis=0)      # shape (N, D)
all_numeric_tr = all_features[:, numeric_idx]        # solo colonne numeriche
scaler_numeric = StandardScaler().fit(all_numeric_tr)
with open("scaler_numeric.pkl", "wb") as f:
    pickle.dump(scaler_numeric, f)

```

Si effettua una normalizzazione delle features tramite StandardScaler(), si concatenano tra loro e si restituiscono nella pipeline, salvandole anche in un apposito file .pkl per importarle successivamente nel procedimento di fine-tuning e riutilizzare lo stesso scaler sui nuovi dati.

3.2.5 Stratified training/validation/test: splitting

Dopo aver normalizzato correttamente le features e i target numerici, nell'addestramento di un modello di Deep Learning, è determinante suddividere il dataset iniziale in tre set distinti: Training, Validation e Test. Il Training Set è quello comprendente più dati ed è utilizzato durante l'addestramento vero e proprio della rete; il Validation Set è ricavato da una piccola porzione del set di training e serve per verificare e validare le prestazioni del modello, ad ogni epoca di iterazione, su dati mai visti prima in addestramento; il Test Set è, infine, un set piccolissimo, sempre ricavato dal set di training (solitamente un 10% della dimensione totale) che raccoglie anch'esso dati mai visti prima, su cui si testa la capacità del modello di generalizzare in maniera consona. La configurazione che ho utilizzato per gestire lo splittaggio è stata:

- *Training set*: 80% del dataset totale.
- *Validation set*: 10% del dataset totale.
- *Test set*: 10% del dataset totale.

In tale maniera, con una suddivisione 80/10/10, sono riuscito a conferire alla rete i dati necessari per l'addestramento, salvaguardando - allo stesso tempo - anche la capacità di generalizzare, da parte del modello, sui set mai visti durante il training. Tuttavia, siccome il problema di manutenzione predittiva affrontato è stato, nella maggioranza dei casi, un problema sbilanciato verso la classe minoritaria: ci si è trovati ad avere, molto spesso, situazioni in cui N° Rettifiche >> N° Sostituzioni e, talvolta a livello di set splitting, non era possibile garantire un'equa distribuzione delle etichette nel Validation e Test Set. Questo ha portato il modello a predire sempre la classe maggioritaria o non disporre di abbastanza dati di entrambe le classi nel

set di validazione per poter rappresentare correttamente le metriche e l'andamento delle performance. Proprio per tale ragione, ho deciso di implementare un approccio di **stratificazione dei set** con fallback, di modo tale che, per ogni sequenza, in ogni set fosse presente almeno un esempio di ciascuna classe esistente e, se per caso non fossero più state disponibili etichette di un tipo di classe in particolare (ad esempio: è presente solo classe 1 ma non la classe 0 e viceversa), si sarebbe deviata, momentaneamente, una di esse dal training set al validation set solo a tempo di validazione. Tale approccio ha consentito di ottenere una prestazione nettamente superiore in termini di AUC e di validazione sulle due classi, che vedremo poi successivamente.

Chiaramente, questo procedimento va attuato solamente se i dati sono davvero sbilanciati, altrimenti si potrebbe incorrere in overfitting... per cui, è stato molto utile con la problematica attuale. Inoltre, Python mette a disposizione una funzione molto utile chiamata: "StratifiedShuffleSplit" che si occupa automaticamente di "stratificare" i set che gli vengono passati. Di seguito, il codice implementato:

```
# _____
# 1-2-3) STRATIFIED SPLIT SUI SOTTOCICLI (quindi per ogni record "portavoce"
del sottociclo => sequenza temporale)
# _____
sub_ids      = np.array([info[1] for info in meta_info])
seq_labels = np.array([y_seq[-1,2] for y_seq in sequences_y])

# etichetta univoca per ciascun subciclo
uniq_subs, first_idxs = np.unique(sub_ids, return_index=True)
sub_labels = seq_labels[first_idxs]
sub_counts = Counter(sub_labels)
# 1) TRAIN vs (VAL+TEST) a livello di sub-ciclo
if sub_counts[0] >= 2 and sub_counts[1] >= 2:
    sss1 = StratifiedShuffleSplit(n_splits=1, test_size=0.20, random_state=42)
    train_sub_ix, temp_sub_ix = next(sss1.split(uniq_subs, sub_labels))
    train_subs = uniq_subs[train_sub_ix]
    temp_subs  = uniq_subs[temp_sub_ix]
    temp_labels = sub_labels[temp_sub_ix]
else:
    # se non stratifico, faccio un semplice random split dei subcicli
    train_subs, temp_subs = train_test_split(
        uniq_subs, test_size=0.20, random_state=42
    )
    # ricavo anche le etichette corrispondenti
    temp_labels  = np.array([sub_labels[list(uniq_subs).index(s)] for s in temp_subs])
# 2) VAL vs TEST sul temp
temp_counts = Counter(temp_labels.astype(int))
if temp_counts[0] >= 2 and temp_counts[1] >= 2:
    # stratifico solo se ho almeno 2 esempi di ciascuna classe
```

```

sss2 = StratifiedShuffleSplit(n_splits=1, test_size=0.50, random_state=42)
val_ix, test_ix = next(sss2.split(temp_subs, temp_labels))
val_subs = temp_subs[val_ix]
test_subs = temp_subs[test_ix]
else:
    # altrimenti split semplice a metà
    half = len(temp_subs) // 2
    val_subs = temp_subs[:half]
    test_subs = temp_subs[half:]
# 3) indice delle sequenze per ciascun set
train_idx = [i for i, sid in enumerate(sub_ids) if sid in train_subs]
val_idx = [i for i, sid in enumerate(sub_ids) if sid in val_subs]
test_idx = [i for i, sid in enumerate(sub_ids) if sid in test_subs]
# 4) costruisco i set X/Y/meta
X_tr = [sequences_x[i] for i in train_idx]
Y_tr = [sequences_y[i] for i in train_idx]
meta_tr= [meta_info[i] for i in train_idx]

X_va = [sequences_x[i] for i in val_idx]
Y_va = [sequences_y[i] for i in val_idx]
meta_va= [meta_info[i] for i in val_idx]

X_te = [sequences_x[i] for i in test_idx]
Y_te = [sequences_y[i] for i in test_idx]
meta_te= [meta_info[i] for i in test_idx]

```

Tramite questo lungo codice ho realizzato, dapprima, la stratificazione del training set, mettendo anche un fallback con if/else, con cui, se le classi per esercitare lo stratify non fossero attualmente disponibili, allora si procederebbe direttamente con lo splitting classico in Training, Validation e Test in forma randomica tramite una percentuale di split. Dopodichè, ho applicato lo stesso procedimento, allo stesso modo, anche al Validation e Test set – dove, in realtà, è molto più necessario a causa dello sbilanciamento – conferendo sempre un fallback condizionale nel caso le classi necessarie non fossero state presenti. Nella pratica, il trucco per non mescolare le features e le label è lavorare sugli **indici** degli esempi, cioè su un array [0, 1, 2, ..., N-1], dove N è il numero totale di campioni. Si estrae prima il Test Set con il medesimo stratagemma, calcolando quanta percentuale di ogni distribuzione di classe inserire all'interno. Si mescola il tutto e si imposta un random seed fisso per la riproducibilità. Si estende questo processo anche al Validation e al Training Set conservando gli indici e gestendo la medesima logica. Il risultato: due vettori di indici, idx_trainval e idx_test che **contengono la stessa proporzione di classi** presente in y. Così facendo, tutti i set sono opportunamente stratificati con la stessa riproducibilità. Infine, ho impostato un ulteriore fallback manuale per il Training Set di modo che, se la stratificazione del Validation e Test avesse tolto troppe classi di un solo tipo dal Training, allora si sarebbe deviata una di esse direttamente dal Validation Set per consentire un adeguato Training.

Il codice utilizzato è il seguente:

```

# --- 5) Fallback manuale se TRAIN non ha entrambe le classi ---
class_labels_tr = np.array([y[-1,2] for y in Y_tr])
print("Prima del fallback → distribuzione train:", Counter(class_labels_tr))
if meta_va and len(np.unique(class_labels_tr)) < 2:
    # classe mancante
    needed = 0 if 0 not in class_labels_tr else 1
    # scorri i sottocicli di VAL
    for sub in set(meta_va):
        idxs = [i for i,m in enumerate(meta_va) if m == sub]
        labs = [Y_va[i][-1,2] for i in idxs]
        if needed in labs:
            # sposta tutto quel sottociclo da VAL a TRAIN
            for i in sorted(idxs, reverse=True):
                X_tr.append(X_va.pop(i))
                Y_tr.append(Y_va.pop(i))
                meta_tr.append(meta_va.pop(i))
            break
    else:
        print("No fallback: train già bilanciato o meta_va vuoto")

    # riconrollo
    class_labels_tr_new = np.array([y[-1,2] for y in Y_tr])
    print("Dopo il fallback → distribuzione train:", Counter(class_labels_tr_new))

    # controllo che ora ci siano entrambe le classi
    assert len(np.unique(class_labels_tr_new)) == 2, \
        "ERRORE: nel train manca ancora una classe!"
    # ora il vero train è:
    seq_x_tr, seq_y_tr = X_tr, Y_tr
    # mi serve per calcolare la distribuzione dei pesi originali delle classi
    seq_y_tr_orig = list(seq_y_tr)

```

Questo intero processo è stato lungo e complesso, ma ha garantito un accurato addestramento del ramo classificatore del modello.

3.2.6 Oversampling, data augmentation e masking zero-padding

L'ultimo passaggio del pre-processing è stato uno dei più importanti in assoluto: bilanciare adeguatamente le classi manutentive in possesso con oversampling del set di training; arricchire il set di training, tramite data augmentation, con varianti di dati sintetici derivati da quelli reali in modalità "on-the-fly" per mezzo del plugin tensorflow generator; uniformare la lunghezza totale delle sequenze di input derivate dai vari set, con zero-padding, opportunamente poi, associato ad una maschera per gestire correttamente il classificatore. Procediamo con ordine:

1. **Oversampling**: è un procedimento molto utile quando si ha a che fare con dataset sbilanciati verso una delle classi in possesso. Purtroppo, nel contesto della manutenzione predittiva, questo avviene molto spesso, soprattutto in una situazione in cui si mette a confronto un'operazione veloce come la Rettifica (molto frequente) con un'operazione dispendiosa e lenta come la Sostituzione (molto più rara). Per cui, trovarsi di fronte a queste situazioni a livello industriale, è prettamente la normalità. Ho seguito un approccio alternativo al classico SMOTE o ADASYN che si applica in questo contesto. Siccome utilizzo sequenze temporali come dato di input al modello neurale, ho deciso di applicare oversampling sequenziale direttamente su di esse. Di seguito, il codice di come l'ho implementato:

```
def oversample_classification(sequences_x, sequences_y):  
    # Indici delle sequenze con target validi e non validi per classificazione  
    valid_indices = [i for i, seq in enumerate(sequences_y) if seq[-1, 2] != -1]  
    invalid_indices = [i for i, seq in enumerate(sequences_y) if seq[-1, 2] == -1]  
  
    if len(valid_indices) == 0:  
        # Nessun esempio valido: ritorna tutto così com'è  
        return sequences_x, sequences_y  
  
    # Estrazione dei target di classificazione validi  
    valid_targets = np.array([sequences_y[i][-1, 2] for i in valid_indices])  
    unique, counts = np.unique(valid_targets, return_counts=True)  
  
    if len(unique) < 2:  
        # Se c'è solo una classe valida, niente oversampling  
        return sequences_x, sequences_y  
  
    # Determinare classe minoritaria e maggioritaria  
    minority_class = unique[np.argmin(counts)]  
    majority_class = unique[np.argmax(counts)]  
    n_min = np.sum(valid_targets == minority_class)  
    n_maj = np.sum(valid_targets == majority_class)  
    extra = n_maj - n_min  
    # Se extra == 0, le classi sono già bilanciate  
    if extra == 0:
```

```

        return sequences_x, sequences_y
    # Selezione degli indici (all'interno di valid_indices) per la classe
    minoritaria
    valid_minority = [i for i in valid_indices if sequences_y[i][-1, 2] ==
                      minority_class]
    # Duplicazione a caso fino ad aggiungere "extra" esempi per il bilanciamento
    extra_indices = np.random.choice(valid_minority, extra, replace=True)

    # Costruzione della nuova lista di indici:
    # - Tutti gli indici originali (sia validi che non validi)
    # - Più gli extra duplicati per la classificazione
    new_indices = list(range(len(sequences_x))) + list(extra_indices)
    np.random.shuffle(new_indices)

    sequences_x_os = [sequences_x[i] for i in new_indices]
    sequences_y_os = [sequences_y[i] for i in new_indices]

    return sequences_x_os, sequences_y_os

```

Agendo sulle sequenze, ho immediatamente isolato le due casistiche principali:

- **Presenza** di “-1” in NextMaintenanceType delle sequenze.
- **Non presenza** di “-1” in NextMaintenanceType delle sequenze.

Chiaramente, era necessario fare oversampling solamente sui record con indice, di NextMaintenanceType, dei record “portavoce”: discriminanti fondamentali immediatamente precedenti alla manutenzione imminente. Ho gestito il tutto in maniera dinamica, di modo tale che, anche se inverosimilmente e ipoteticamente in uno scenario futuro, la classe minoritaria fosse stata la Rettifica, allora sarebbe stata etichettata come minoritaria e bilanciata con la Sostituzione. Infine, il tutto è stato concatenato e restituito sottoforma di sequenze temporali oversamplete sui valori delle etichette.

2. **Data Augmentation:** Altra tecnica fondamentale nel pre-processing dei dati. Consente di arricchire il dataset con varianti sintetiche dei dati, derivati da quelli già presenti. Serve, principalmente, per conferire variabilità al dataset stesso, di modo tale che il modello neurale riesca ad esaminare, durante l’addestramento, più variazioni possibili del set di training in ingresso e, di conseguenza, non adattarsi ai dati di input, ma imparare bene a generalizzare su di essi. Ho deciso di utilizzare varie tecniche, che ho sinergizzato tra di loro in un’unica funzione e l’ho passata alla creazione dei dataset on-the-fly. Le metodologie sfruttate sono quelle più compatibili con il concetto di sequenza di dati e rete neurale ricorrente, ovvero:

- *Magnitude Warping*: particolarmente utile con le sequenze temporali. Rende il modello più robusto a variazioni di ampiezza o intensità del segnale. Infatti, **altera localmente l’ampiezza del segnale stesso**, in modo non lineare e controllato, generando esempi derivati distorti ma realistici. Nella pratica, si moltiplica il segnale originale per una curva di fattori di scala continua

(quindi valori numerici) che varia lentamente lungo il tempo. Chiaramente, nel mio caso, questo procedimento è stato applicato direttamente alle sequenze temporali di input:

```
def magnitude_warping(seq, sigma=0.2, knot=4):
    timesteps, features = seq.shape
    if timesteps < 2:
        return seq
    knot = min(knot, timesteps)
    knot_positions = np.linspace(0, timesteps - 1, num=knot)
    random_factors = np.random.normal(loc=1.0, scale=sigma, size=(knot,
features))
    orig_steps = np.arange(timesteps)
    warping_curve = np.zeros((timesteps, features))
    for i in range(features):
        cs = CubicSpline(knot_positions, random_factors[:, i])
        warping_curve[:, i] = cs(orig_steps)
    return seq * warping_curve

def magnitude_warping_sequence(sequences, sigma=0.2, knot=4):
    augmented_sequences = []
    for seq in sequences:
        warped_seq = magnitude_warping(seq, sigma=sigma, knot=knot)
        augmented_sequences.append(warped_seq)
    return augmented_sequences
```

- *Time Warping*: altra tecnica utilizzata con le serie temporali dove, invece di alterare l'ampiezza del segnale, si **modifica la velocità** dei sotto-segmenti del segnale ricevuto (la vera e propria scala temporale). Rende il modello più robusto a variazioni di timing e velocità nel pattern del segnale, applicando una mappatura non lineare del tempo; quindi, allungando o comprimendo localmente tratti del segnale senza però alterarne l'ampiezza effettiva. Nella pratica, si definisce una funzione monotona crescente, che mappa il tempo originale “ t ” nel nuovo tempo τ .

È stata applicata sulle sequenze temporali nel seguente modo:

```
def time_warp_sequence(seq, time_idx, warp_range=(0.8, 1.2)):
    factor = np.random.uniform(warp_range[0], warp_range[1])
    seq_warped = seq.copy()
    seq_warped[:, time_idx] = seq[:, time_idx] * factor
    return seq_warped

def time_warping(sequences, time_idx, warp_range=(0.8, 1.2)):
    return [time_warp_sequence(seq, time_idx, warp_range) for seq in sequences]
```

- *Jitter*: è, probabilmente, la tecnica di data augmentation più diffusa per serie temporali. Consiste nell'aggiungere rumore gaussiano ai campioni del segnale, per rendere il modello più robusto a piccole variazioni casuali nei dati lungo l'andamento temporale. Simula il rumore di misura o di piccole fluttuazioni nella sequenza di dati, evitando che il modello si adatti troppo a pattern molto precisi e statici. Ad ogni valore del segnale originale si somma un piccolo contributo di disturbo, solitamente estratto da una distribuzione normale centrata in zero. Il tutto rende il modello più robusto agli outlier e alle piccole imperfezioni che si possono riscontrare lungo i timestep delle sequenze stesse. Nell'esempio seguente ho utilizzato un valore di rumore molto piccolo che ho trovato ottimale ($\sigma=0.03$).

```
def jitter(seq, sigma=0.03):
    noise = np.random.normal(loc=0.0, scale=sigma, size=seq.shape)
    return seq + noise

def jittering_sequence(sequences, sigma=0.03):
    augmented_sequences = []
    for seq in sequences:
        jittered_seq = jitter(seq, sigma)
        augmented_sequences.append(jittered_seq)
    return augmented_sequences
```

Per non eccedere nella quantità di rumore immesso all'interno del dataset, ho deciso di gestire, l'immissione di data augmentation on-the-fly, tramite una funzione che andasse ad inserire le funzioni sopracitate in maniera randomica, con una probabilità di attivazione di 0.2 per ognuna. Facendo così, si è evitato il rischio di introdurre troppa distorsione all'interno delle sequenze con conseguente possibilità di deviare il gradiente dal corretto percorso. L'augmentazione è fondamentale, ma non deve essere mai talmente esagerata da coprire totalmente il segnale grezzo, altrimenti le performance peggiorano e l'apprendimento rimane in stallo. La metodologia utilizzata è stata la seguente:

```
def augment_x(x_np):
    choice = np.random.choice(['jitter', 'magnitude', 'time', 'none'],
p=[0.3,0.3,0.3,0.1])
    if choice == 'jitter':
        return jitter(x_np, sigma=0.03).astype(np.float32)
    elif choice == 'magnitude':
        return magnitude_warping(x_np, sigma=0.1, knot=4).astype(np.float32)
    elif choice == 'time':
        return time_warp_sequence(x_np,time_idx=0,
warp_range=(0.95,1.05)).astype(np.float32)
    else:
```

```

        return x_np.astype(np.float32)
def tf_augment(x, y):
    do_aug = tf.random.uniform([]) < 0.5
    x_aug = tf.cond(
        do_aug,
        lambda: tf.numpy_function(augment_x, [x], tf.float32),
        lambda: x
    )
    x_aug.set_shape(x.shape)
    return x_aug, y

```

Così facendo, il programma sceglie randomicamente tra una delle tecniche di data augmentation e la applica ai dati per il processamento imminente. Inoltre, ho generato un fallback - che si attiva con probabilità del 50% - con cui l'augmentazione può essere iniettata oppure no. Questo procedimento serve sempre per non andare a saturare il modello con troppe varianti, di dati sintetici, tutte insieme.

3. **Masking Zero-Padding:** procedimento essenziale per uniformare la lunghezza di tutte le sequenze temporali di ingresso. La rete neurale LSTM Bidirezionale lavora molto bene con sequenze di lunghezza costante; proprio per questo motivo, l'addestramento sarebbe impossibile se venissero introdotte sequenze di lunghezza variabile tra di loro. Per sopperire a questa problematica, ho utilizzato un metodo molto comune in Deep Learning che è lo **zero-padding**. Tale tecnica consiste nell'aggiungere liste di zeri in testa o in coda alla sequenza originaria corrispondente, tenendo in conto della lunghezza massima tra tutte le sequenze a disposizione. In questo modo, ogni sequenza originaria viene popolata con zeri (zero-padding) fino a che non si raggiunge la lunghezza massima possibile presente nel dataset (max-padding). La modalità di iniezione dello zero-padding che ho utilizzato è "pre", ovvero in testa alla sequenza, poiché, i record "portavoce" finali delle sequenze, dovevano assolutamente rimanere intatti, per garantire una corretta discriminazione delle successive classi manutentive. Il codice utilizzato è il seguente:

```

def pad_dynamic_sequences(sequences_x, sequences_y):
    max_length = max(seq.shape[0] for seq in sequences_x)
    num_features = sequences_y[0].shape[1]
    x_padded = pad_sequences(
        sequences_x,
        padding='pre', # preservo il record "portavoce" finale
        dtype='float32',
        maxlen=max_length
    )
    y_padded_list = []
    for y in sequences_y:
        pad_len = max_length - y.shape[0]
        if pad_len > 0:
            pad_block = np.ones((pad_len, num_features), dtype=np.float32)*-1.0
            y_full = np.vstack([pad_block, y])
        else:

```

```

        y_full = y
        y_padded_list.append(y_full)
y_padded = np.stack(y_padded_list, axis=0)
return x_padded, y_padded

```

Mentre, per quanto riguarda la funzione di mascheramento associata alla logica di padding, ho elaborato:

```

def mask_padding(x, y):
    if isinstance(y, (tuple, list)):
        y_dict = {
            "remaining_rett": y[0],
            "remaining_sost": y[1],
            "next_maint_type": y[2]
        }
    else:
        y_dict = y
    # mask per la regressione
    mask_r = tf.not_equal(y_dict["remaining_rett"], -1)
    mask_s = tf.not_equal(y_dict["remaining_sost"], -1)
    # ---- normalizziamo next_maint_type ----
    nm = y_dict["next_maint_type"]
    # nm può arrivare con shape [batch,1] oppure [batch,1,1]
    # lo "appiattiamo" in [batch]
    cls = tf.reshape(nm, [-1])
    # mask classificazione
    mask_c = tf.not_equal(cls, -1)      # [batch]
    # rimettiamo y_dict["next_maint_type"] in shape [batch,1]
    y_dict["next_maint_type"] = tf.expand_dims(cls, -1)
    # sample-weights: diamo a ognuno il suo peso, shape [batch,1]
    sw = {
        "remaining_rett": tf.cast(mask_r, tf.float32),
        "remaining_sost": tf.cast(mask_s, tf.float32),
        "next_maint_type": tf.cast(tf.expand_dims(mask_c, -1), tf.float32)
    }
    return x, y_dict, sw

```

Con questa funzione, tutti i valori placeholder, dei timestep produttivi, per NextMaintenanceType di valore “-1” vengono adeguatamente mascherati e non considerati dal classificatore durante l’addestramento. Tale aspetto è cruciale, in quanto la discriminazione delle classi deve avvenire esclusivamente sui valori [0, 1] inerenti a Sostituzione e Rettifica. Infatti, come abbiamo affermato precedentemente, i record produttivi appartenenti alla medesima sequenza del record “portavoce” avranno, come target classificativo, lo stesso che tale record “portavoce” possiede (many-to-one).

3.3 Algoritmo, modello e logica di programmazione

3.3.1 Ciclo di vita del piano martire

Dopo aver codificato correttamente i dati ottenuti e prima di poter aver effettuato la normalizzazione dei valori numerici, è stato necessario implementare interamente da zero un algoritmo che permetesse la corretta segmentazione del dataframe, seguendo una logica opportuna, al fine di creare sequenze temporali da introdurre nella rete neurale LSTM Bidirezionale. Si è deciso, pertanto, di utilizzare il concetto reale di “ciclo di vita” e “sottociclo interno” del piano martire che, più nello specifico rispettivamente: il primo identifica la durata della vita di un piano di lavoro a partire dal momento in cui viene installato sulla macchina (quindi nuovo di zecca), fino al punto in cui esso viene sostituito perché è troppo usurato e, il secondo invece, identifica la durata di un sottociclo di produzioni fino al prossimo evento manutentivo ... difatti, per il primo, si parla di N rettifiche in mezzo a due sostituzioni consecutive tra loro mentre, per il secondo, si parla di singoli eventi produttivi ordinati temporalmente tra due eventi di manutenzione consecutivi tra loro. Per cui, si è potuto gestire il tutto come:

- **T1:** Tempo in cui il piano martire è stato installato e quindi è nuovo. Per cui è stato possibile trattare ciò come una sostituzione pregressa prima che il piano attuale venisse inserito.
- **T2:** Tempo in cui il piano martire attuale viene sostituito con il suo successore. Quindi, tale istante, è stato trattato come una sostituzione futura del piano attuale installato.

Pertanto, il ciclo di vita del piano martire è rappresentato proprio dalla differenza temporale tra l’istante T2 e l’istante T1:

$$\text{Ciclo di vita} = \mathbf{T2 - T1}$$

Analogamente, un discorso simile si è applicato anche al concetto di sottociclo di produzione, infatti:

- **TS1:** Tempo in cui è stata effettuata una manutenzione pregressa; quindi, solitamente sarà un’operazione di rettifica, tranne per il momento iniziale del ciclo di vita stesso (che è, per l’appunto, una sostituzione) e anche per la sua fine (sempre una sostituzione). Infatti, l’inizio e la fine del ciclo di vita coincidono anche rispettivamente con l’inizio del primo sottociclo e la fine dell’ultimo sottociclo interno.
- **TS2:** Tempo in cui il piano martire attuale subisce una manutenzione successiva, consecutiva alla precedente. Quindi, tale istante, è stato trattato, solitamente, come una rettifica futura del piano attuale installato. Se, invece, si è nell’ultimo sottociclo, tale istante di tempo coincide esattamente con la sostituzione del piano.

Per cui, anche in questa situazione, il sottociclo è rappresentato dalla differenza temporale tra i due:

$$\text{Sottociclo} = \mathbf{TS2 - TS1}$$

Come abbiamo già potuto affermare: all’interno di un ciclo di vita esistono N sottocicli di rettifica per M programmi di lavorazione, rapportati però ad una sola sostituzione con questa formula matematica:

$$\sum_{i=1}^N M_i \cdot N_i$$

Il concetto intrinseco di ciclo è stato riassunto così a livello di codice:

```
def segment_cycles(df):
    df = df.copy()
    cycle = 0
    cycle_ids = []
    cycle_start = []
    current_cycle_start = None # Qui salveremo il to corrente
    for idx, row in df.iterrows():
        if row['EventType'] == 0 and row.get('MaintenanceType', -1) == 0:
            current_cycle_start = row['Timestamp']
            cycle_ids.append(cycle)
            cycle_start.append(current_cycle_start)
            cycle += 1
        else:
            cycle_ids.append(cycle)
            if current_cycle_start is None:
                current_cycle_start = row['Timestamp']
            cycle_start.append(current_cycle_start)

    df['cycle_id'] = cycle_ids
    df['Cycle_Start'] = cycle_start
    return df
```

E, analogamente quello di sottociclo:

```
def segment_subcycles(df):
    df = df.copy()
    all_subcycle_ids = []
    all_subcycle_start = []
    # Raggruppa per ciclo
    for cycle, group in df.groupby('cycle_id'):
        subcycle = 0
        current_subcycle_start = None
        sub_ids = []
        sub_starts = []
```

```

# Ordina i record nel gruppo per timestamp
group = group.sort_values("Timestamp")
for idx, row in group.iterrows():
    if row['EventType'] == 0:
        if current_subcycle_start is None:
            current_subcycle_start = row['Timestamp']
        sub_ids.append(subcycle)
        sub_starts.append(current_subcycle_start)
        current_subcycle_start = row['Timestamp']
        subcycle += 1
    else:
        if current_subcycle_start is None:
            current_subcycle_start = row['Timestamp']
        sub_ids.append(subcycle)
        sub_starts.append(current_subcycle_start)
    # Conserva i risultati per il gruppo
    all_subcycle_ids.extend(sub_ids)
    all_subcycle_start.extend(sub_starts)
df['subcycle_id'] = all_subcycle_ids
df['SubCycle_Start'] = all_subcycle_start
return df

```

Con queste righe di codice sono riuscito ad ordinare temporalmente i vari record presenti nel dataset, grazie al parametro “Timestamp” in comune a tutte le tipologie di evento. Ho ulteriormente creato colonne aggiuntive per gestire correttamente la logica di divisione in cicli e sottocicli: **Subcycle_id** e **Cycle_id**.

Ognuno è il rispettivo ID di ogni tipologia di segmentazione logica che ho attuato sul dataframe. Successivamente, è stato fondamentale definire due contatori ciclici principali:

- **CumulativeEvents**: contatore del mapping completo di tutto il ciclo di vita del piano, quindi a partire dalla sostituzione pregressa alla sostituzione prossima. In questo modo, è stato possibile tracciare ogni singolo evento produttivo che fosse stato lanciato dall'inizio alla fine.
- **SubCycle_Events**: contatore di eventi produttivi in un sottociclo di rettifica. Molto simile a TotalEvents, ma con uno scopo diverso nell'insieme della logica implementativa. È, fondamentalmente, un contatore di supporto per non andare ad inquinare il valore di TotalEvents con i calcoli dei valori attesi.

Ho realizzato tutto questo con la seguente funzione:

```

def compute_counters(df):
    df = df.sort_values("Timestamp").reset_index(drop=True)
    cumulative = 0
    subcycle = 0
    cumulative_list = []
    subcycle_list = []

```

```

for i, row in df.iterrows():
    if row['EventType'] == 1:
        cumulative += 1
        subcycle += 1
        cumulative_list.append(cumulative)
        subcycle_list.append(subcycle)
    else:
        cumulative_list.append(cumulative)
        subcycle_list.append(0)
        # Se è una sostituzione (MaintenanceType == 0), reset totale
        if row.get('MaintenanceType', -1) == 0:
            cumulative = 0
            subcycle = 0
df['CumulativeEvents'] = cumulative_list
df['SubCycle_Events'] = subcycle_list
return df

```

In questo modo, i contatori vengono resettati al valore massimo quando si incontra un'operazione di sostituzione e i vari cicli ripartono. Mentre, ad ogni operazione di produzione, i contatori vanno avanti di uno perché registrano una nuova lavorazione nel programma. Questa modalità ha permesso di effettuare una mappatura completa dei valori numerici di interesse, per poterli utilizzare poi, nella composizione delle sequenze temporali di input alla rete neurale. Una volta conclusi tutti gli aspetti di segmentazione e calcolo, è stato necessario procedere con il raggruppamento vero e proprio in cicli e sottocicli di vita, generando quindi le sequenze già opportunamente segmentate. Per mezzo degli ID corrispondenti, è stata effettuata la suddivisione vera e propria, tramite il codice seguente:

```

def create_representative_sequences(df, features, target, factor=1.0):
    sequences_x = []
    sequences_y = []
    meta_info = []
    id_seq = 1
    for cycle, group_cycle in df.groupby('cycle_id'):
        for subcycle, group_sub in group_cycle.groupby('subcycle_id'):
            group_sub = group_sub.sort_values("Timestamp").reset_index(drop=True)
            working_events = group_sub[group_sub['EventType'] == 1]
            if working_events.empty:
                continue
            seq_x = working_events[features].values
            seq_y = working_events[target].values
            seq_length = seq_x.shape[0]
            max_sub = working_events['SubCycle_Events'].max()
            max_cum = working_events['CumulativeEvents'].max()
            if factor < 1.0:
                portion_length = max(1, int(seq_length * factor))

```

```

        seq_x = seq_x[-portion_length:]
        seq_y = seq_y[-portion_length:]
        sequences_x.append(seq_x)
        sequences_y.append(seq_y)
        meta_info.append((cycle, subcycle, seq_length, id_seq, max_sub, max_cum))
        id_seq += 1
    return sequences_x, sequences_y, meta_info

```

Questa è stata la funzione vera e propria di creazione delle sequenze rappresentative, formanti il dataframe, da prendere in considerazione come dati di input. Ho deciso, inoltre, di includere una variabile “factor” per decidere quanta porzione - di ogni sequenza – utilizzare. Strategia particolarmente utile con esempi di dataset molto grandi, che evita una computazione troppo onerosa su hardware CPU. Ho inizializzato, dunque, due array dimensionali vuoti, rispettivamente per le features X e i target Y e ho concatenato due cicli “for” tra di loro per andare a sequenziare il dataset nella modalità desiderata. Specifica attenzione va al fatto che le features e i target sono già stati originati prima di chiamare questa funzione, la quale poi restituisce anche metadati inerenti al dataframe... particolarmente utili per gestire i vari collegamenti successivi a tempo di inferenza finale. Infatti, per ogni sequenza, si hanno informazioni su:

- Cycle: Ciclo di vita attuale.
- Subcycle: Sottociclo di rettifica attuale.
- Seq_length: Lunghezza della sequenza in questione.
- Id_seq: Identificativo della sequenza corrispondente.
- Max_sub: Valore massimo di lavorazioni raggiunto dal sottociclo corrispondente.
- Max_cum: Valore massimo di lavorazioni raggiunto dal ciclo corrispondente.

Si è delinata la quantità massima di eventi produttivi dei sottocicli prendendo il massimo valore di SubCycle_Events e salvandolo in “Max_sub” e, analogamente, la quantità massima di eventi produttivi dei cicli prendendo il massimo valore di CumulativeEvents e salvandolo in “Max_cum”; per poi popolare, tramite essi, i metadati corrispondenti. A questo punto, è stato opportuno definire anche quali sono state le features e i target da prendere in considerazione, che venivano realmente passati alla funzione di creazione di sequenze:

```

features = ["Timestamp", "EventType", "CumulativeEvents", "SubCycle_Events",
            "TotalEvents", "WorkingHours", "cycle_id", "WearValue",
            "Tot_Events_Rett", "Tot_Events_Sost", "T_Rettifica", "T_Sostituzione"]
target = ["Remaining_Rett", "Remaining_Sost", "NextMaintenanceType"]

```

I target sono, per l'appunto, i valori che la rete deve determinare e predire, minimizzando la propria funzione obiettivo (loss function). Più la funzione di perdita viene minimizzata, maggiore saranno le possibilità - da parte della rete - di apprendere correttamente. Nello specifico:

- **Remaining_rett**: Valore regressivo di eventi produttivi rimanenti prima della prossima rettifica.
- **Remaining_sost**: Valore regressivo di eventi produttivi rimanenti prima della prossima sostituzione.
- **NextMaintenanceType**: Variabile binaria classificativa che indica il prossimo evento manutentivo da compiere.

Pertanto, mi sono trovato dinnanzi a un problema multi-output regressivo e classificativo con tre target da considerare simultaneamente. Per poter ottemperare a tale compito, ho realizzato la seguente funzione:

```
def prepare_targets_combined(df, incomplete_cycle_target=-1):
    df = df.copy()
    # 1) assegno gli indici di produzione per subcycle e cycle
    df['Production_Index_Subcycle'] = np.nan
    df['Production_Index_Cycle'] = np.nan
    for (c, s), grp in df.groupby(['cycle_id', 'subcycle_id']):
        idx = 0
        for i, row in grp.iterrows():
            if row['EventType'] == 1:
                df.at[i, 'Production_Index_Subcycle'] = idx
                idx += 1
    for c, grp in df.groupby('cycle_id'):
        idx = 0
        for i, row in grp.iterrows():
            if row['EventType'] == 1:
                df.at[i, 'Production_Index_Cycle'] = idx
                idx += 1
    # 2) calcola T_sub e T_cycle
    df['T_sub']=df.groupby(['cycle_id','subcycle_id'])['Production_Index_Subcycle']\
    .transform('max').fillna(0) + 1
    df['T_cycle'] = df.groupby('cycle_id')['Production_Index_Cycle'] \
        .transform('max').fillna(0) + 1

    # se un sottociclo non ha ancora manuten-tivi (EventType==0),
    # uso il numero reale di eventi (TotalEvents) anziché T_sub “fittizio”,
    # e se un ciclo non ha ancora sostituzione (MaintenanceType==0),
    # uso CumulativeEvents anziché T_cycle “fittizio”.
    # 1) individua subcicli _senza_ alcun evento manutentivo
    sub_has_maint = df.groupby(['cycle_id','subcycle_id'])['EventType']\
        .transform(lambda x: (x==0).any())
    mask_sub_incomplete = ~sub_has_maint
    # override T_sub con TotalEvents
    df.loc[mask_sub_incomplete, 'T_sub'] = df.loc[mask_sub_incomplete, 'TotalEvents']

    # 2) individua cicli _senza_ alcuna sostituzione (incompleti)
    cyc_has_sub = df.groupby('cycle_id')['MaintenanceType']\
        .transform(lambda x: (x==0).any())
    mask_cycle_incomplete = ~cyc_has_sub
    # override T_cycle con CumulativeEvents
```

```

df.loc[mask_cycle_incomplete, 'T_cycle'] = df.loc[mask_cycle_incomplete,
'CumulativeEvents']

# 3) countdown normalizzati (solo su EventType==1)
prod = df['EventType'] == 1
df.loc[prod, 'Remaining_Rett'] = (
    (df.loc[prod,'T_sub'] - df.loc[prod,'Production_Index_Subcycle'] - 1)
    / df.loc[prod,'T_sub']
)
df.loc[prod, 'Remaining_Sost'] = (
    (df.loc[prod,'T_cycle'] - df.loc[prod,'Production_Index_Cycle'] - 1)
    / df.loc[prod,'T_cycle']
)
df['NextMaintenanceType'] = incomplete_cycle_target
for (c, s), grp in df.groupby(['cycle_id','subcycle_id']):
    prod_idxs = grp.index[grp.EventType == 1].tolist()
    if not prod_idxs:
        continue
    last = prod_idxs[-1]
    # se esiste un record manutentivo di sostituzione → 0, altrimenti 1
    if ((grp.EventType == 0) & (grp.MaintenanceType == 0)).any():
        df.at[last, 'NextMaintenanceType'] = 0
    else:
        df.at[last, 'NextMaintenanceType'] = 1

return df

```

Questo snippet di codice ha assunto fondamentale importanza, perché è stato il responsabile della corretta implementazione della logica algoritmica del problema. Vediamolo più nel dettaglio. Innanzitutto, si identificano due scenari distinti e possibili:

- *Cicli/sottocicli completi*
- *Cicli/sottocicli incompleti*

I primi sono quelli di cui abbiamo discusso finora, ovvero blocchi di eventi produttivi che vengono alternati da operazioni di manutenzione e che, quindi, si concludono sempre con un evento manutentivo finale. I secondi sono quelli in cui l'evento manutentivo finale non è ancora arrivato o magari sono stati interrotti prima di raggiungerlo. Questo significa che, se a livello di ciclo grande e generale, la sostituzione finale non è ancora stata raggiunta, ci si trova in un ciclo aperto; mentre, anche a livello di sottociclo interno funziona esattamente allo stesso modo, se anche qui non è ancora stata raggiunta l'operazione di manutenzione corrispondente. Pertanto, l'algoritmo deve essere in grado di gestire questa particolare situazione, andando a mappare l'evento produttivo come unità minima di misura del piano martire e gestendo la predizione dei target in base ad esse. Le domande ulteriori sono:

- Come è possibile definire correttamente i tre target?
- Quali sono i criteri che ne consentono il corretto calcolo in modo dinamico?

Per rispondere correttamente ad entrambe le domande, bisogna dividere il discorso in parte regressiva e classificativa

del

problema.

Per quanto riguarda la regressione, si è consapevoli che è necessario calcolare delle quantità che, poco alla volta, decrescono da una quantità massima iniziale fino ad arrivare ad una quantità minima finale. Se supponiamo di avere dei valori normalizzati tra 0 e 1: il valore iniziale sarà molto vicino all'1 e il valore finale sarà molto vicino allo 0. Per questo si parla di MinMaxScaler e di un vero e proprio contatore regressivo decrescente che, ad ogni iterazione, viene decrementato di una quantità 'K' inherente alle abitudini del cliente. In altre parole, è stato necessario calcolare dei valori attesi storici di eventi produttivi rispettivi di ciclo e sottociclo e utilizzarli come unità di misura per andare a calcolare tutti i vari eventi successivi. In questo modo, se un ciclo o sottociclo fosse stato incompleto, si sarebbero assunti i valori medi: **Expected_cycle_value** ed **Expected_subcycle_value** come medie storiche delle produzioni dei cicli e sottocicli precedenti e, di conseguenza, tramite altri due contatori: **True_cycle_value** e **True_subcycle_value** sarebbe stato possibile calcolare esattamente la differenza, tra le due categorie di contatori ad ogni timestep, fornendo così un'approssimazione dinamica di quanti Remaining_rett e Remaining_sost fossero ancora svolgibili prima della manutenzione

corrispondente.

Infatti:

- **Remaining_rett = Expected_subcycle_value – True_subcycle_value**
- **Remaining_sost = Expected_cycle_value – True_cycle_value**

Con questi contatori dinamici è stato possibile calcolare quanti eventi di produzione rimanenti mancavano alla prossima manutenzione ad ogni timestep di ogni sequenza in modalità many-to-many (architettura di rete in cui ogni timestep della rete ha un output), poiché ci si è basati sulla differenza del valore storico simultaneamente sottratto al valore reale, attualmente raggiunto dal ciclo e sottociclo correlato. Ho realizzato questa logica con la seguente funzione:

```
def compute_expected_values(df, dedupe: bool = True):
    exp = load_expected()
    exp.setdefault("processed_cycle_ids", [])
    exp.setdefault("processed_subcycle_ids", [])

    # cicli completi
    complete_cycles = df[(df['EventType']==0) & (df['MaintenanceType']==0)]
    cycle_totals = complete_cycles.groupby('cycle_id')['CumulativeEvents'].max()

    if dedupe:
        new_cycles = [cid for cid in cycle_totals.index
                      if cid not in exp["processed_cycle_ids"]]
    else:
        new_cycles = list(cycle_totals.index)

    if new_cycles:
```

```

exp["sum_cycle_totals"] += cycle_totals.loc[new_cycles].sum()
exp["count_cycle_totals"] += len(new_cycles)
exp["processed_cycle_ids"].extend(new_cycles)
# sottocicli completi
complete_subs = df[(df['EventType']==0) & (df['MaintenanceType']!=1)]
sub_totals=complete_subs.groupby(['cycle_id','subcycle_id'])['TotalEvents'].max()
for (cid, sid), tot in sub_totals.items():
    key = f"{cid}-{sid}"
    is_new = (not dedupe) or (key not in exp["processed_subcycle_ids"])
    if is_new:
        exp["sum_subcycle_totals"] += tot
        exp["count_subcycle_totals"] += 1
        exp["processed_subcycle_ids"].append(key)
save_expected(exp)
expected_cycle_total = round(
    exp["sum_cycle_totals"] / exp["count_cycle_totals"]
    if exp["count_cycle_totals"]>0 else 0
)
expected_subcycle_total = round(
    exp["sum_subcycle_totals"] / exp["count_subcycle_totals"]
    if exp["count_subcycle_totals"]>0 else 0
)
print(f"Expected Cycle Total: {expected_cycle_total}")
print(f"Expected Subcycle Total: {expected_subcycle_total}")

return expected_cycle_total, expected_subcycle_total

```

Tali valori vengono creati ad ogni nuovo avvio, basandosi sulla media storica attuale dei dati del cliente (in questo caso simulati) e di tutto il dataset a disposizione. Dopodichè, essi vengono salvati in un file .json, il quale viene caricato, successivamente, nella procedura di fine-tuning, per tenere conto delle medie storiche accumulate nel dataset di default e nell'addestramento globale iniziale, prima di effettuare il vero e proprio raffinamento specifico.

Analogamente, ho anche deciso di tenere traccia anche dei valori temporali medi orari di quanto ci mette una lavorazione ad essere eseguita, in media, in un ciclo di sostituzione e in un sottociclo di rettifica. Questo ha consentito di poter ottenere - oltre al numero di lavorazioni rimanenti - anche il numero di giorni di produzione ancora disponibili prima di effettuare la manutenzione corrispondente. Si noti che non è un tipo di dato direttamente predetto dalla rete, ma calcolato a posteriori per mezzo dei valori di lavorazioni rimanenti moltiplicati per la media oraria di svolgimento di ognuna, secondo la tipologia di manutenzione corrispondente. La funzione realizzata è la seguente:

```

def calcola_default_avg_hours(df, maintenance_type, col_time, col_events):
    # Filtra i record per cui il tipo di manutenzione e l'intervallo sono validi.
    df_filtered = df[(df['MaintenanceType'] == maintenance_type) & (df[col_time] > 0)]
    # Raggruppa per cycle_id; seleziona solo i gruppi con almeno 2 record.
    df_completi = df_filtered.groupby('cycle_id').filter(lambda g: g.shape[0] >= 2)
    if not df_completi.empty:
        df_completi = df_completi.copy()
        df_completi['avg_hours_per_event']=df_completi[col_time]/df_completi[col_events]
        return df_completi['avg_hours_per_event'].mean()
    else:
        return None

DEFAULT_EXPECTED = {
    "sum_cycle_totals":      0,
    "count_cycle_totals":    0,
    "sum_subcycle_totals":   0,
    "count_subcycle_totals": 0,
    "processed_cycle_ids":  [],
    "processed_subcycle_ids": []
}

```

Dove poi, nella pipeline del main del programma:

```

# Seleziono i record "validi" in cui l'intervallo (T_Rettifica o T_Sostituzione) è > 0
valid_rett = df_rett_orig[df_rett_orig['T_Rettifica'] > 0]
valid_sost = df_sost_orig[df_sost_orig['T_Sostituzione'] > 0]

default_avg_hours_rett=calcola_default_avg_hours(df_manutenzione,
maintenance_type=1, col_time="T_Rettifica", col_events="TotalEvents")
if default_avg_hours_rett is None:
    default_avg_hours_rett = None # se non si hanno ancora dati nello storico il valore di default è non definito

default_avg_hours_sost=calcola_default_avg_hours(df_manutenzione,
maintenance_type=0, col_time="T_Sostituzione", col_events="CumulativeEvents")
# Se non abbiamo dati sufficienti, posso decidere di impostare un fallback (oppure passarlo via parametro)
if default_avg_hours_sost is None:

```

```

    default_avg_hours_sost = None # se non si hanno ancora dati nello storico
il valore di default è non definito
    # --- Per le RETTIFICHE ---
    if valid_rett.empty:
        tot_events_rett_storico_medio = 0
        avg_hours_rett = 0
    else:
        if valid_rett.shape[0] < 2:
            tot_events_rett_storico_medio = valid_rett.iloc[0]['TotalEvents']
            avg_hours_rett = default_avg_hours_rett
        else:
            valid_rett = valid_rett.copy()
            valid_rett['avg_hours_per_event'] = valid_rett['T_Rettifica'] /
            valid_rett['TotalEvents']
            tot_events_rett_storico_medio = valid_rett['TotalEvents'].mean()
            avg_hours_rett = valid_rett['avg_hours_per_event'].mean()

    # --- Per le SOSTITUZIONI ---
    if valid_sost.empty:
        tot_events_sost_storico_medio = 0
        avg_hours_sost = 0
    else:
        if valid_sost.shape[0] < 2:
            tot_events_sost_storico_medio = valid_sost.iloc[0]['CumulativeEvents']
            avg_hours_sost = default_avg_hours_sost
        else:
            valid_sost = valid_sost.copy()
            valid_sost['avg_hours_per_event'] = valid_sost['T_Sostituzione'] /
            valid_sost['CumulativeEvents']
            tot_events_sost_storico_medio = valid_sost['CumulativeEvents'].mean()
            avg_hours_sost = valid_sost['avg_hours_per_event'].mean()

```

Discorso diverso si è applicato, invece, alla classificazione, poiché la logica di tale problema era leggermente differente. Infatti, l'evento manutentivo, normalmente, era posto in fondo alla sequenza di appartenenza e questo implicava una situazione in cui, ogni timestep della medesima sequenza, inferisse su uno stesso target con modalità many-to-one (architettura di rete in cui ogni timestep di una medesima sequenza punta ad uno stesso risultato). Praticamente, il record significativo per gestire correttamente la manutenzione successiva era proprio il record di indice “i-1” rispettivamente all’evento di manutenzione “i”, proprio perché, come evento subito successivo ad esso, era presente la manutenzione stessa corrispondente. Così, ho deciso di utilizzare tale record – precedentemente citato - “**portavoce**”, per inferire direttamente tutta la sequenza di appartenenza sullo stesso target classificativo successivo in modalità many-to-one. Quindi, sostanzialmente, ogni sequenza poteva contare sul proprio record portavoce personale, che andava a riflettere il relativo record successivo su tutti i timestep della sequenza, facendo, pertanto, in modo che ogni singolo record della

sequenza stessa avesse, come target classificativo, la medesima manutenzione. Detto ciò, ogni sequenza temporale veniva generata direttamente da tale record, il quale associava a tutti gli altri record, aventi il medesimo “Subcycle_Id”, lo stesso target “NextMaintenanceType”.

Fatto ciò, ci si è trovati davanti ai due metodi di iterazione, da parte della rete, citati precedentemente, ma finalmente concretizzati:

- **Many-to-many:** ogni timestep del ciclo di vita esprime un output regressivo dei due target: Remaining_rett e Remaining_sost, che inferiscono dinamicamente su quante lavorazioni sono ancora attuabili per manutenzione correlata.
- **Many-to-one:** tutti i timestep relativi ad una stessa sequenza temporale di appartenenza (quindi allo stesso sottociclo) inferiscono lo stesso output classificativo NextMaintenanceType di fine sequenza, dettato dal record “portavoce” corrispondente.

Di conseguenza, l’output a video di tutto questo procedimento di pre-processing e definizione dei target è il seguente:

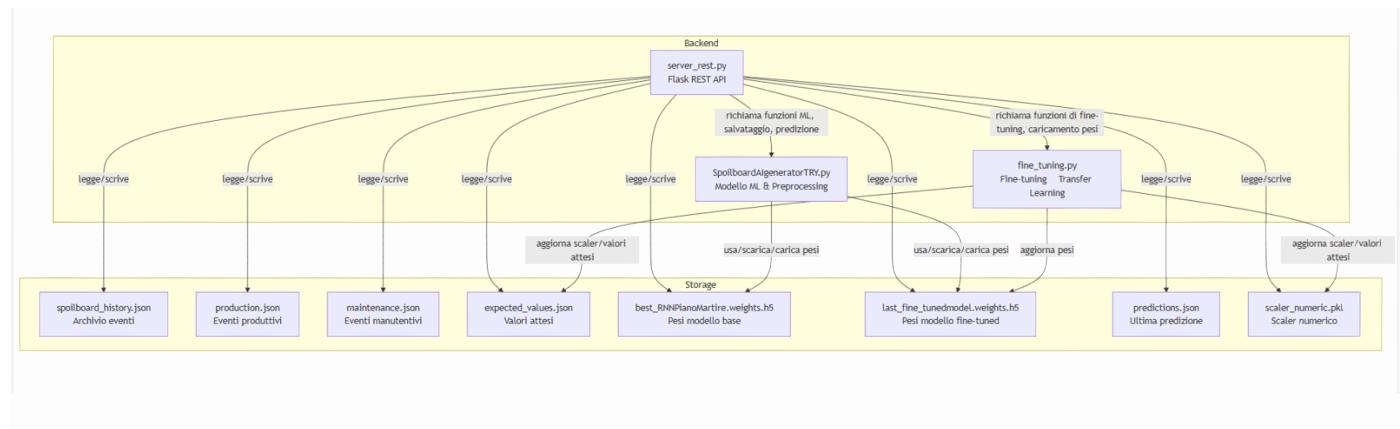
```
Expected Cycle Total: 14485
Expected Subcycle Total: 495
Statistiche Remaining_Rett:
count    499951.000000
mean      0.499000
std       0.288675
min      0.000000
25%      0.249004
50%      0.498998
75%      0.748992
max      0.998182
Name: Remaining_Rett, dtype: float64
Statistiche Remaining_Sost:
count    499951.000000
mean      0.498479
std       0.289525
min      0.000000
25%      0.247726
50%      0.498474
75%      0.749212
max      0.999944
Name: Remaining_Sost, dtype: float64
Distribuzione NextMaintenanceType:
NextMaintenanceType
-1      499951
 1       960
 0        40
```

Come si può notare da questa immagine, i valori iniziali Expected_* rispecchiano l’andamento storico del dataset popolato inizialmente. La distribuzione dei due target regressivi è normalizzata con MinMaxScaler() tra 0 e 1 e la distribuzione del target classificativo avviene sul record portavoce per ogni sequenza. Infatti, in questa specifica situazione, si hanno mille sequenze (una per ogni record portavoce), suddivise in 960 rettifiche e 40 sostituzioni. Quindi, ci sono all’incirca 40 cicli di vita completi, con 960 sottocicli di rettifica

divisi tra le 40 sostituzioni con un range randomico di cycle_length_range da 15 a 35, dettato dalla funzione di simulazione iniziale del dataset:

```
def simulate_lifecycle_dataset(
    total_maintenance_records: int,
    prod_block_range=(450, 550),
    rettifica_wear_range=(0.60, 0.70),
    sostituzione_wear_range=(0.70, 0.80),
    maint_time_range_days=(2, 3),
    cycle_length_range=(15, 35),
    prod_duration_sec_range=(1, 10)
):
```

La classe “-1” è, come dicevo precedentemente, un placeholder che ho utilizzato per gestire l’output di NextMaintenanceType nei pressi di ogni timestep di produzione che non sia il record “portavoce” (il quale, proprio nella colonna target di NextMaintenanceType, punta al valore effettivo della classe della manutenzione successiva). Naturalmente, se si sta agendo in modalità many-to-one, significa che, ogni timestep della medesima sequenza, possiede lo stesso target dell’ultimo record di tale sequenza; quindi, non ha senso includerlo nella mappatura e, per non destabilizzare l’addestramento e i calcoli che ne derivano, è buona prassi effettuare una mascheratura che non lo vada a tenere in considerazione. Quindi, di fatti, i record delle produzioni totali – non “portavoce” - che si sono compiute in questo scenario specifico sono circa: 499951. Di seguito, una mappa concettuale, in formato .emf, di tutta la logica algoritmica utilizzata nel mio problema, generata con IA:



3.3.2 Architettura del modello, compilazione e addestramento

Ora passiamo al cuore vero e proprio del progetto, ovvero quello che ha riguardato prettamente la parte di costruzione dell'architettura del modello, la sua compilazione e il relativo addestramento. Questa parte di codice è stata, probabilmente, la più significativa e, anche quella che ho modificato fino all'ultimo, per testare costantemente nuovi approcci, nuovi layer e cercare di potenziare le performance della rete al meglio delle mie possibilità. Come affermato precedentemente, ho utilizzato un modello di rete neurale ricorrente con due layer convoluzionali integrati, per la precisione: Long-short Term Memory Bidirezionale (BiLSTM) multi-output con adapter layer, meccanismo di self-attention integrato e due layer convoluzionali prima delle teste regressive per migliorare l'efficienza di calcolo. Di seguito, il codice completo del mio modello neurale:

```
def modelling_combined(num_features, units=128):
    inputs = Input(shape=(None, num_features))
    masked = Masking(mask_value=0.0, name='masking')(inputs)

    # —— 3 blocchi Bi-LSTM + LayerNorm + Dropout ——
    x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(masked)
    x = LayerNormalization()(x)
    x = Dropout(0.2)(x)
    x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(x)
    x = LayerNormalization()(x)
    x = Dropout(0.2)(x)
    x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(x)
    x = LayerNormalization()(x)
    x = Dropout(0.2)(x)

    # rompo la maschera per i layer CNN
    x_no_mask = Lambda(lambda t: t,
                        mask=lambda inputs, mask: None,
                        name="drop_mask")(x)

    # recupero maschera iniziale per andarla a mettere dopo i Conv1D layers
    mask_float = Lambda(
        lambda m: tf.cast(tf.expand_dims(m, -1), x_no_mask.dtype),
        name="mask_float"
    )(masked._keras_mask)

    # —— AttentionLayer ——
    context_vector, attention_scores = AttentionLayer(name="attention")(
        x,
```

```

        mask=masked._keras_mask
    )
# Semplifico: riduco subito a [batch, D]
squeezed = Lambda(lambda z: tf.squeeze(z, axis=1),
name="context_squeeze")(context_vector)
squeezed = GaussianNoise(0.05)(squeezed)

# → uso blocco Conv1D 3x3 + residual connections per boostare regressione
RETT senza aumentare complessità della rete
# —— blocco RETT ——
conv_r = Conv1D(64, 3, padding='same', activation=None)(x_no_mask)
conv_r = Activation('relu')(conv_r)
conv_r = LayerNormalization()(conv_r)
conv_r = Dropout(0.1)(conv_r)
proj_r = Conv1D(64, 1, padding='same', activation=None)(x_no_mask)
proj_r = LayerNormalization()(proj_r)
# residual + relu
r1_unmasked = Add()([conv_r, proj_r])
r1Activated = Activation('relu')(r1_unmasked)
# qui riapplico la mask: tutti i padding (0) rimangono 0
r1 = Multiply(name="r1_masked")([r1Activated, mask_float])

# —— Ramo regressivo (Many-to-Many) ——
# ramo remaining_rett
rett = MyTimeDistributed(Dense(64, activation='relu',
kernel_regularizer=l2(0.0001)))(r1)
rett = LayerNormalization()(rett)
rett = Dropout(0.15)(rett)
out_rett = MyTimeDistributed(Dense(1, activation='linear'),
name="remaining_rett_lin")(rett)
output_rett = Lambda(lambda x: tf.clip_by_value(x, 0.0, 1.0),
name="remaining_rett")(out_rett)

# → uso blocco Conv1D 3x3 + residual connections per boostare regressione
SOST senza aumentare complessità della rete
# —— blocco SOST ——
conv_s = Conv1D(64, 3, padding='same', activation=None)(x_no_mask)
conv_s = Activation('relu')(conv_s)
conv_s = LayerNormalization()(conv_s)
conv_s = Dropout(0.1)(conv_s)
proj_s = Conv1D(64, 1, padding='same', activation=None)(x_no_mask)
proj_s = LayerNormalization()(proj_s)
# residual + relu

```

```

s1_unmasked = Add()([conv_s, proj_s])
s1Activated = Activation('relu')(s1_unmasked)
# riapplico la stessa mask
s1 = Multiply(name="s1_masked")([s1Activated, mask_float])

# ramo remaining_sost
sost = MyTimeDistributed(Dense(64, activation='relu',
kernel_regularizer=l2(0.0001)))(s1)
sost = LayerNormalization()(sost)
sost = Dropout(0.15)(sost)
output_sost = MyTimeDistributed(Dense(1, activation='sigmoid'),
name="remaining_sost")(sost)

# — Ramo classificativo (Many-to-One) usa il context_vector per
MultiHeadAttention —
# Layer Fully Connected
class_dense = Dense(128, activation='relu',
kernel_regularizer=l2(0.0001))(squeezed)
class_dense = LayerNormalization()(class_dense)
class_dense = Dropout(0.1)(class_dense)

class_dense = Dense(64, activation='relu',
kernel_regularizer=l2(0.0001))(class_dense)
class_dense = LayerNormalization()(class_dense)
class_dense = Dropout(0.1)(class_dense)
class_dense = Dense(32, activation='relu',
kernel_regularizer=l2(0.0001))(class_dense)
class_dense = LayerNormalization()(class_dense)
class_dense = Dropout(0.1)(class_dense)

# — Adapter leggero con gating per evitare "catastrophic forgetting" nel
fine-tuning —
# 1) recupero dimensione corrente di class_dense
orig_dim = K.int_shape(class_dense)[-1] # es. 32
# 2) "cuore" dell'adapter: scendo a un quarto (o anche un ottavo) della
dimensione
hidden_dim = max(4, orig_dim // 8)
adapter_hidden = Dense(hidden_dim,
activation='relu',
name='adapter_hidden')(class_dense)
# 3) risalgo subito a orig_dim (senza ulteriori normalizzazioni intermedie)
adapter_out = Dense(orig_dim,
activation=None, # lasciamo il lineare, basta poi la LN

```

```

        name='adapter_out'))(adapter_hidden)
adapter_drop = Dropout(0.05, name='adapter_dropout')(adapter_out)
# 4) somma residua (più leggera di un gating scalare)
residual = Add(name='adapter_residual')([class_dense, adapter_drop])
# 5) una sola LayerNormalization finale per stabilizzare
normed = LayerNormalization(name='adapter_ln')(residual)
# 6) testa di output per la classificazione
output_class = Dense(1,
                      activation='sigmoid',
                      name='next_maint_type')(normed)

# —— Maschera eventi di manutenzione per le custom losses ——
idx_event = features.index("EventType")
ev_type    = Lambda(lambda z: z[:, :, idx_event])(inputs) # [batch, T]
mask_maint = Lambda(lambda z: K.equal(z, 0))(ev_type) # True dove
EventType==0

model = Model(
    inputs=inputs,
    outputs={
        "remaining_rett": output_rett,
        "remaining_sost": output_sost,
        "next_maint_type": output_class
    }
)

# —— Loss globali di consistency / reset ——
_ = ConsistencyLossLayer(0.1)([output_rett, output_sost])
_ = ResetLossLayer(0.1)([output_rett, #pred
                        output_sost,
                        mask_maint])
return model

```

Come si può ben notare, il codice è molto articolato e complesso; quindi, lo analizzerò un passo alla volta. Inizialmente, ho gestito la shape delle features di input e ho creato un layer di Masking per andare ad isolare i valori riempitivi derivanti dallo zero-padding, di modo tale che non andassero ad inquinare negativamente le prestazioni durante l'addestramento. Quindi, sostanzialmente, tale layer è servito a processare correttamente tutte le sequenze di lunghezza variabile in ingresso alla rete. Tale maschera è stata, poi chiaramente, propagata a tutti i layer del modello, tranne per quelli CNN-1D dove essa viene smantellata, perché i layer CNN-1D non sono in grado di gestire la nozione di “sequenze con masking”, a causa dell’operazione direttamente sulla finestra fissa; pertanto, tramite una funzione Lambda, ho rimosso la maschera appena prima di passare i dati a Conv1D, evitando qualunque errore di incompatibilità. Successivamente a ciò, ho

gradualmente introdotto tre layer Bi-LSTM con 128 neuroni l'uno e un regolarizzatore L2 = 0.0001, opportunamente normalizzati tramite LayerNormalization (il tipo di BatchNormalization applicato alle sequenze ricorsive) e regolarizzati con un Dropout di 0.2, in questo modo:

```
# — 3 blocchi Bi-LSTM + LayerNorm + Dropout —
x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(masked)
x = LayerNormalization()(x)
x = Dropout(0.2)(x)

x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(x)
x = LayerNormalization()(x)
x = Dropout(0.2)(x)

x = Bidirectional(LSTM(units, return_sequences=True,
kernel_regularizer=l2(0.0001)))(x)
x = LayerNormalization()(x)
x = Dropout(0.2)(x)
```

Questi tre layer iniziali sono stati condivisi sia dal branch regressivo che da quello classificativo della rete neurale e, quindi, è servito prettamente per analizzare le features dei dati in maniera condivisa e poi specializzarsi successivamente nei layer seguenti. In seguito, ho generato il layer per la disattivazione della maschera sui layer Conv1D e il layer di recupero di tale maschera per poter riapplicarla sui layer successivi alla convoluzione. Era fondamentale ripristinare correttamente la maschera sui layer ricorrenti successivi, poiché il problema, che la rete si trovava a modellare, era fortemente dipendente dalla struttura stessa della sequenza. Quindi, è stato essenziale non corrompere i risultati ottenuti fino a quel momento con zeri di padding contestualmente superflui.

```
# rompo la maschera per i layer CNN
x_no_mask = Lambda(lambda t: t,
                    mask=lambda inputs, mask: None,
                    name="drop_mask")(x)

# recupero maschera iniziale per andarla a mettere dopo i Conv1D layers
mask_float = Lambda(
    lambda m: tf.cast(tf.expand_dims(m, -1), x_no_mask.dtype),
    name="mask_float"
)(masked._keras_mask)
```

Arrivati a questo punto, ho deciso di utilizzare un vettore di contesto generato col meccanismo di Attention, creando, di fatto, un **Attention Layer**, il quale è un meccanismo, nei modelli di Deep Learning, che permette

di pesare dinamicamente parti diverse di un input (o di una rappresentazione intermedia) in funzione della rilevanza rispetto ad un dato contesto o “query”. In poche parole, invece di trattare tutti i timestep o tutte le posizioni con la stessa importanza, l’Attention Layer impara a focalizzarsi sulle porzioni più utili per il compito in corso, conferendo loro un peso maggiore. Infatti, nei modelli sequenziali (RNN, LSTM ecc), l’informazione passa in uno stato nascosto di dimensione fissa. Quando la sequenza è lunga, i primi elementi possono andare a perdere e diventare poco influenti sullo stato finale. L’attention risolve esattamente questo problema perché:

- **Raccoglie** le rappresentazioni di **tutti** i time-step.
- **Calcola** per ciascuno un **punteggio di rilevanza** rispetto a una “query” (ad es. lo stato finale, un vettore di contesto, un altro token).
- **Normalizza** questi punteggi (tipicamente con attivazione softmax) per ottenere **pesi** tra 0 e 1 che sommano a 1.
- **Costruisce** un **vettore di contesto** come combinazione pesata di tutte le rappresentazioni.

Ciò che ne risulta, assume questa forma:

```
# —— AttentionLayer ——
context_vector, attention_scores = AttentionLayer(name="attention")(
    x,
    mask=masked._keras_mask
)
# Semplifico: riduco subito a [batch, D]
squeezed      = Lambda(lambda      z:      tf.squeeze(z,      axis=1),
name="context_squeeze")(context_vector)
squeezed = GaussianNoise(0.05)(squeezed)
```

Per regolarizzare il tutto, ho inserito un leggero rumore gaussiano a valle dell’AttentionLayer, in modo da stabilizzare ulteriormente l’addestramento. Per continuare, ho indirizzato la rete nei due branch principali, dopo aver correttamente traslato il vettore di contesto in una lista dimensionale:

- i. **Regessione.**
- ii. **Classificazione.**

Per quanto riguarda la regressione, ho voluto provare ad inserire dei layer convoluzionali prima di ognuna delle due uscite regressive, per analizzare la sequenza per intera e cercare di migliorare ciascun layer senza andare ad introdurre ulteriore complessità e bias nel modello. Ho optato per un approccio di Conv1D con layer 3x3 e una tecnica di residual connections per evitare il problema del vanishing gradient (molto ricorrente nelle LSTM). Naturalmente, come affermato precedentemente, la maschera generata per coprire gli zeri inerenti al padding, è stata eliminata in prossimità dell’inizio convoluzione (poiché non è supportata) e ricostruita successivamente alla fine di essa.

```
# —— blocco RETT ——
conv_r = Conv1D(64, 3, padding='same', activation=None)(x_no_mask)
conv_r = Activation('relu')(conv_r)
conv_r = LayerNormalization()(conv_r)
```

```

conv_r = Dropout(0.1)(conv_r)
proj_r = Conv1D(64, 1, padding='same', activation=None)(x_no_mask)
proj_r = LayerNormalization()(proj_r)
# residual + relu
r1_unmasked = Add()([conv_r, proj_r])
r1Activated = Activation('relu')(r1_unmasked)
# qui riapplico la mask: tutti i padding (0) rimangono 0
r1 = Multiply(name="r1_masked")([r1Activated, mask_float])

# —— Ramo regressivo (Many-to-Many) ——
# ramo remaining_rett
rett = MyTimeDistributed(Dense(64, activation='relu',
kernel_regularizer=l2(0.0001)))(r1)
rett = LayerNormalization()(rett)
rett = Dropout(0.15)(rett)
out_rett = MyTimeDistributed(Dense(1, activation='linear'),
name="remaining_rett")(rett)
output_rett = Lambda(lambda x: tf.clip_by_value(x, 0.0, 1.0),
name="remaining_rett")(out_rett)

# → uso blocco Conv1D 3x3 + residual connections per boostare regressione
SOST senza aumentare complessità della rete
# —— blocco SOST ——
conv_s = Conv1D(64, 3, padding='same', activation=None)(x_no_mask)
conv_s = Activation('relu')(conv_s)
conv_s = LayerNormalization()(conv_s)
conv_s = Dropout(0.1)(conv_s)
proj_s = Conv1D(64, 1, padding='same', activation=None)(x_no_mask)
proj_s = LayerNormalization()(proj_s)
# residual + relu
s1_unmasked = Add()([conv_s, proj_s])
s1Activated = Activation('relu')(s1_unmasked)
# qui riapplico la mask: tutti i padding (0) rimangono 0
s1 = Multiply(name="s1_masked")([s1Activated, mask_float])

# ramo remaining_sost
sost = MyTimeDistributed(Dense(64, activation='relu',
kernel_regularizer=l2(0.0001)))(s1)
sost = LayerNormalization()(sost)
sost = Dropout(0.15)(sost)
output_sost = MyTimeDistributed(Dense(1, activation='sigmoid'),
name="remaining_sost")(sost)

```

I layer regressivi utilizzati sono una variante – adattata al mio specifico problema - del custom layer di Keras: **Time Distributed**. Esso è un wrapper layer di Keras che permette di applicare il medesimo layer (o blocco di layer) a ciascun timestep della sequenza temporale di ingresso (proprio come iterare un ciclo for su step temporali o su una lista). Quindi, è un metodo molto conveniente se si utilizzano sequenze temporali (come nel mio caso) dove ogni timestep è significativo – many-to-many - a livello di output predetto. Tuttavia, tra i due branch regressivi c’è una differenza sostanziale: la funzione di attivazione differente nella testina Dense finale. Ho effettuato vari tentativi di addestramento, alcuni migliori e altri peggiori in termini prestazionali e, in fin dei conti, ho osservato che: il modello rispondeva molto bene sulla regressione di Remaining_rett con una funzione di attivazione lineare – la quale consente al modello di elaborare i valori nella scala numerica originaria – probabilmente dovuto al fatto che: l’intervallo intrinseco del massimo e minimo dei valori di Remaining_rett erano nettamente inferiori a quelli di Remaining_sost e, di conseguenza, anche il gap numerico di differenza, tra i due estremi, era chiaramente minore. Proprio per questo, ho nutrito una certa confidenza nell’addestrare il modello con questa configurazione sul ramo delle Rettifiche. Diverso scenario, invece, ho dovuto elaborare per le Sostituzioni: il range tra il massimo e il minimo valore per un ciclo di vita intero è, normalmente, molto più elevato (nell’ordine di decine o centinaia di migliaia vs centinaia o migliaia per i sottocicli). Questo implica una varianza dei valori più elevata e una conseguente possibilità di outlier lungo i timestep temporali. Dove nel ramo di Rettifica, a differenza da quello di Sostituzione, sono riuscito a gestire gli outlier con un semplice clipping sui valori predetti dopo il Time Distributed, lo stesso non ho potuto confermare per la Sostituzione. Per questo, dopo molteplici tentativi, ho deciso di cambiare la funzione di attivazione, sul layer Dense della testina di Sostituzione, da “linear” a “sigmoid”. La funzione di attivazione sigmoidale è riuscita, infatti, a gestire questa varianza elevata e il problema degli outlier in maniera molto più robusta, rispetto a quella lineare, in un contesto come quello attuale poiché, anche pur ottenendo, probabilmente, valori agli estremi meno precisi (sigmoid tende ad appiattire i valori agli estremi se essi sono troppo fuori scala rispetto all’andamento corrente), grazie alla standardizzazione e normalizzazione delle features in ingresso, è riuscita a conferire una costante progressività alla monotonia decrescente dei valori di Remaining_sost, garantendo così robustezza e maggiore precisione di una funzione lineare. Ovviamente, questa non è l’unica soluzione possibile: sicuramente, assemblando la rete in modo differente e normalizzando i dati in modi alternativi, si sarebbe forse potuto ottenere un altro andamento meno rigido a livello regressivo per i valori di Sostituzione ma, nel mio caso, ho preferito seguire questo approccio asimmetrico.

Proseguendo, ho implementato il branch classificativo many-to-one, sfruttando il meccanismo di Attention che ho implementato precedentemente dopo i layer BiLSTM e, passando il vettore di contesto appiattito in una lista, per fare coincidere le dimensionalità e sfruttarne comunque l’efficacia. Ho gradualmente inserito tre layer Dense (Fully-Connected) con funzione di attivazione “relu” - perfetta per evitare il problema del vanishing gradient nelle reti ricorrenti e per garantire valori ≥ 0 (non possono esserci valori di lavorazioni rimanenti negativi) - regolarizzandoli con un regolarizzatore L2 molto basso (0.0001) a causa di un underfitting costante sul branch classificativo e utilizzando sottomultipli di 128 unità neuronali iniziali per popolare i livelli più profondi. Naturalmente, come da buona prassi di Deep Learning, ho anche inserito ulteriori livelli di LayerNormalization e Dropout (con valore 0.1) dopo ogni layer Dense per prevenire l’overfitting e stabilizzare il gradiente durante l’addestramento.

```

# —— Ramo classificativo (Many-to-One) usa il context_vector per
MultiHeadAttention —
    # Layer Fully Connected
    class_dense = Dense(128, activation='relu',
kernel_regularizer=l2(0.0001))(squeezed)
    class_dense = LayerNormalization()(class_dense)
    class_dense = Dropout(0.1)(class_dense)

    class_dense = Dense(64, activation='relu',
kernel_regularizer=l2(0.0001))(class_dense)
    class_dense = LayerNormalization()(class_dense)
    class_dense = Dropout(0.1)(class_dense)

    class_dense = Dense(32, activation='relu',
kernel_regularizer=l2(0.0001))(class_dense)
    class_dense = LayerNormalization()(class_dense)
    class_dense = Dropout(0.1)(class_dense)

```

Ho effettuato svariati tentativi con: più neuroni su ciascun livello, meno neuroni su ciascun livello e regolarizzando di più o di meno tramite L2, Dropout e LayerNormalization... questa è stata finora la combinazione migliore che potessi testare. Oltre al fatto che, per gestire alla meglio il fine-tuning successivo ed evitare il fenomeno di “**catastrophic forgetting**”, ho aggiunto un meccanismo di Adapter Layer dopo i layer Dense e prima della testa di output sigmoidale di classificazione effettiva. L’Adapter è un piccolo modulo “down-up projection” inserito nel backbone di un modello pre-allenato ed è molto importante in un contesto futuro di fine-tuning perché:

- Riduce enormemente i parametri da aggiornare (solo quelli degli adapter).
- Preserva il comportamento del modello originario, minimizzando il “catastrophic forgetting”.
- Permette modularità multi-tasking: basta adibire un adapter per ogni nuovo compito.

Esso si integra anche molto semplicemente. Infatti, solitamente, si inserisce dopo blocchi nevralgici del modello – come i layer Dense di cui ho parlato – e, in fase di elaborazione dei dati, li proietta verso uno spazio di dimensione ridotta detta “bottleneck” e applica loro una trasformazione, per poi farli ripristinare alla dimensione originale... eventualmente anche utilizzando una Residual Connection in modo tale che, se l’Adapter non serve in quel determinato contesto, il modello continua comunque a comportarsi esattamente com’era prima, ignorandolo. Questo aspetto assicura alcuni vantaggi cruciali:

- **Salvataggio leggero**: per ogni nuovo task basta salvare pochi Mb di pesi adapter invece di tutto il modello.
- **Multi-task**: si possono collegare adapter diversi ad uno stesso modello backbone e alterarli a runtime secondo le proprie esigenze.
- **Catastrophic forgetting**: essendo residuali, preservano il comportamento originale della rete e riducono il rischio di dimenticare ciò che conosceva prima.

- **Specializzazione:** se si necessita di uno sblocco di pochi layer finali durante il fine-tuning, esso permette di adattare meglio la parte di output in maniera modulare e flessibile senza stravolgere l'intero network.

```
# — Adapter leggero con gating per evitare "catastrophic forgetting" nel
fine-tuning —
# 1) recupero dimensione corrente di class_dense
orig_dim = K.int_shape(class_dense)[-1] # es. 32
# 2) "cuore" dell'adapter: scendo a un quarto (o anche un ottavo) della
dimensione
hidden_dim = max(4, orig_dim // 8)
adapter_hidden = Dense(hidden_dim,
                       activation='relu',
                       name='adapter_hidden')(class_dense)
# 3) risalgo subito a orig_dim (senza ulteriori normalizzazioni intermedie)
adapter_out = Dense(orig_dim,
                     activation=None, name='adapter_out')(adapter_hidden)
adapter_drop = Dropout(0.05, name='adapter_dropout')(adapter_out)
# 4) somma residua (più leggera di un gating scalare) per evitare l'uso
dell'Adapter se non serve
residual = Add(name='adapter_residual')([class_dense, adapter_drop])
# 5) una sola LayerNormalization finale per stabilizzare
normed = LayerNormalization(name='adapter_ln')(residual)
# 6) testa di output per la classificazione
output_class = Dense(1,
                      activation='sigmoid',
                      name='next_maint_type')(normed)
```

Ho registrato inizialmente la dimensione originaria degli strati Dense che serve per passarla allo strato nascosto dell'adapter. Dopodiché, ho creato un “bottleneck” con attivazione “relu” nello strato nascosto, per scendere ad un ottavo della dimensione originaria e quindi adattare l'output ad una dimensione differente da quella iniziale. Infine, ho ripristinato la dimensione originaria tramite il layer di Adapter output, sovrapponendolo ad un leggero dropout per stabilizzare l'addestramento, ad un livello di LayerNormalization e ad un Residual Connection per evitare l'uso dell'adapter stesso in caso di non necessità. Alla fine di tutte queste concatenazioni, ho situato la testina Dense di classificazione, con attivazione sigmoidale, vera e propria. Ho preferito l'uso di “sigmoid” invece di “softmax” perché si è trattato di un problema di classificazione binaria (solo due classi in gioco) e, pertanto, sigmoid ha funzionato meglio. Per concludere la spiegazione della funzione di creazione del modello neurale, ho incluso la generazione effettiva del modello stesso tramite la direttiva “Model”, una maschera ulteriore per gestire una funzione di perdita personalizzata di reset a fine e inizio sottociclo/ciclo e un'altra loss personalizzata di Consistency. Le ultime due citate sono servite per conferire al modello una spinta ulteriore nell'apprendere il pattern della regressione in modalità monotona decrescente ed escludere, da tale calcolo, gli eventi effettivi di manutenzione dal dataset (poiché il calcolo della regressione per Remaining_rett e Remaining_sost avviene sugli eventi di produzione). Grazie alla loss di Consistency applicata nei confronti di output_sost, si è insegnato al modello che le sue predizioni sarebbero

state penalizzate se, per qualsiasi motivo durante l'andamento regressivo, i valori di Remaining_rett avessero superato significativamente i valori di Remaining_sost – cosa realmente impossibile poiché al limite potrebbero essere stati quasi equivalenti, ma una sostituzione (essendo a fine ciclo di vita) ha sempre più eventi mancanti rispetto ad una rettifica, salvo alcune eccezioni – mentre, con la loss di Reset, il modello sarebbe stato penalizzato se, a fine ciclo/sottociclo, non fosse ripartito, con la predizione, dal valore effettivo storico iniziale di lavorazioni rimanenti totali per ogni tipo distinto.

```
# — Maschera eventi di manutenzione per le custom losses —
idx_event = features.index("EventType")
ev_type    = Lambda(lambda z: z[:, :, idx_event])(inputs) # [batch, T]
mask_maint = Lambda(lambda z: K.equal(z, 0))(ev_type) # True dove
EventType==0

model = Model(
    inputs=inputs,
    outputs={
        "remaining_rett": output_rett,
        "remaining_sost": output_sost,
        "next_maint_type": output_class
    }
)

# — Loss globali di consistency / reset —
_ = ConsistencyLossLayer(0.1)([output_rett, output_sost])
_ = ResetLossLayer(0.1)([output_rett, #pred
                        output_sost,
                        mask_maint])

return model
```

Il passo successivo alla creazione è stato, naturalmente, la compilazione del modello nella pipeline principale e il suo effettivo addestramento tramite `model.fit()`. Di seguito illustro il codice di entrambi i passaggi:

1. Compilazione

Rappresenta il passaggio in cui si definisce **come** la rete deve apprendere dai dati stessi. In pratica:

- **OTTIMIZZATORE**: Algoritmo che, ad ogni batch, aggiorna i pesi della rete per ridurre la loss function (SGD, Adam, RMSProp, Adagrad ecc).

- **LOSS FUNCTION:** Metrica scalare che misura l'errore tra predizioni e target. Il processo di backpropagation calcola i gradienti di questa loss e l'ottimizzatore scelto li usa per aggiornare i pesi del modello. In contesti standard si usa MSE e MAE per la regressione e Binary_Crossentropy per la classificazione.
- **METRICHE:** Sono calcoli accessori (accuracy, precision, recall, f1-score, specificity, AUC, ecc.) utili a monitorare l'andamento del training/validation, ma non utilizzate direttamente per l'aggiornamento dei pesi.

Durante la compile, viene inizializzato l'ottimizzatore con i propri stati interni (momentum, accumulatore ecc), vengono poi associate, la loss e le metriche, al grafo computazionale del modello stesso e vengono create funzioni (callable) che, per ogni batch ad ogni epoca, calcolano:

- FORWARD PASS: predizione sui dati.
- LOSS: funzione di perdita.
- BACKWARD PASS: gradiente di tutti i pesi.
- UPDATE: modifica dei pesi tramite ottimizzatore.
- CALCOLO METRICHE: analisi delle metriche sui vari batch.

Quindi, solamente dopo aver compilato il modello tramite `model.compile()`, si può parlare di addestramento vero e proprio. Di seguito, il codice inerente alla compilazione:

```
# compilazione del modello
optimizer = tf.keras.optimizers.Adam(learning_rate=initial_lr, clipnorm=1.0)
# uso ottimizzatore Adam con learning rate personalizzato e clipping
# 1) Definisco una variabile globale di soglia, inizialmente a 0.5
model_combined.compile(
    optimizer=optimizer,
    loss={
        "remaining_rett": DynamicHuber(name="huber_rett"),
        "remaining_sost": DynamicHuber(name="huber_sost"),
        "next_maint_type": dynamic_focal_loss(gamma=2.0, minor_weight=3.0)
    },
    loss_weights={
        "remaining_rett": 2.0,
        "remaining_sost": 2.0,
        "next_maint_type": 3.0
    },
    metrics={
        "remaining_rett": ["mse", "mae"],
        "remaining_sost": ["mse", "mae"],
        "next_maint_type": ["accuracy",
                           MaskedAUC(name="auc", from_logits=False), # ← metrica mascherata
                           precision_at_thr,
                           recall_at_thr,
                           f1_score],
        "next_maint_type": [tf.keras.metrics.Precision(name="precision"),
                            tf.keras.metrics.Recall(name="recall"),
                            tf.keras.metrics.F1Score(name="f1")]
    }
)
```

```

        f1_score_metric
    ]
}
)
model_combined.summary()

```

E, analogamente, quello inerente all'implementazione e ottimizzazione iniziale del learning rate:

```

# Uso combo di tecniche di warm-up + reduceonplateau del learning_rate da valore
iniziale a valore stabile finale
# 1) parametri warm-up
warmup_epochs = 5
initial_lr    = 1e-6 # punto di partenza (può essere anche 0)
target_lr     = 1e-5 # LR a fine warm-up

def warmup_scheduler(epoch, lr):
    if epoch < warmup_epochs:
        # scala linearmente lr da initial_lr a target_lr
        return initial_lr + (target_lr - initial_lr) * ((epoch + 1) /
warmup_epochs)
    else:
        # dopo il warm-up, non tocchiamo più lr qui
        return lr

# provo a "giocare" col learning rate warmup + decay conservativo
lr_warmup = LearningRateScheduler(warmup_scheduler, verbose=1)

# 2) ReduceLROnPlateau conservativo per stabilizzare il decay
lr_plateau = ReduceLROnPlateau(
    monitor='val_remaining_rett_mae',
    mode='min',
    factor=0.5,
    patience=2,
    min_lr=1e-6,
    verbose=1
)

```

Parlerò più avanti riguardo alle loss utilizzate, alle metriche e ai loss weights. Per ora, basti sapere che ho utilizzato un ottimizzatore Adam, con clipping del gradiente integrato, per evitare problematiche di exploding gradient durante l'addestramento e un modulo di WarmUpScheduler + ReduceLROnPlateau per gestire il corretto approccio e aggiornamento del learning rate. Il primo consente di contenere i valori del gradiente entro un range stabile e consone all'addestramento del modello, il secondo, invece, è diviso in due parti e momenti distinti:

1. *WarmUpScheduler*: Fa in modo che il modello inizi l'addestramento, in alcune prime epoche, con un learning rate iniziale più basso rispetto alla necessità reale; questo serve principalmente per evitare, in primis, esplosioni improvvise garantendo un “warm-up” iniziale del gradiente e, successivamente adattare il tutto in progressione, sbloccando incrementalmente il modello e consentendo lui di crescere solo una volta trovata la direzione corretta.
2. *ReduceLROnPlateau*: Meccanismo, a mio parere, sottovalutato ma fondamentale soprattutto in un contesto regressivo. Serve a ridurre gradualmente il learning rate attuale, ad un valore minimo settato inizialmente, dopo un tot di epoche in cui la metrica monitorata non migliora in modo significativo e, quindi, dove il modello stesso si trova in una fase di stallo o plateau. Questo è un ulteriore aiuto, sommato al Momentum già integrato in Adam, che consente al gradiente di valicare zone più ripide del normale e di non adattarsi mai a punti critici di stallo. A livello regressivo è stato veramente utile nel mio contesto attuale, poiché ha permesso al modello di migliorare le performance di validazione durante l'addestramento, monitorando direttamente il MAE di validazione di Remaining_rett.

2. Addestramento

Il momento effettivo in cui il modello inizia a processare il proprio algoritmo di apprendimento sui dati di training. Ho associato correttamente il training set utilizzato e il validation set da tenere in considerazione, generati tf.data pipeline per generare il set desiderato “on-the-fly”. Siccome i dati da tenere in considerazione erano un numero considerevole – poiché, oltre a tutti gli eventi manutentivi, erano presenti anche i vari blocchi di eventi di produzione – ho consolidato che era impossibile mantenere tutto in memoria centrale senza andare ad intaccare le prestazioni pesantemente; proprio per questo, grazie a questo generatore, i batch del dataset sono stati caricati uno alla volta in memoria, con notevoli vantaggi:

- **Flessibilità**: dati letti da liste, file o database; applicazione poco costosa di augmentazioni o altre operazioni di pre-processing (oversampling, class weight ecc) effettuate direttamente real-time.
- **Dinamico**: condizioni, filtri, maschere o sampling complessi si codificano direttamente nel generatore, lasciando la RAM molto meno sovraccaricata.

Ci sono, naturalmente, anche alcuni svantaggi:

- Bisogna inserire gli **steps_per_epoch** da compiere perché, essendo tutto caricato on-the-fly, il compilatore non conosce effettivamente quanti batch-step effettuare a priori. Si fanno quindi delle prove con valori iniziali, per poi variarli in fase di post-addestramento, ripetendo l'addestramento stesso.
- **Debug più complesso** a runtime, perché gli errori nel generator appaiono solo durante il fit, e quindi va gestito un possibile fallback manuale.

Di conseguenza, se non si ha a disposizione un hardware potente, conviene utilizzare tf.data pipeline, ma gestendo il tutto con cura per evitare overhead Python. Di seguito, il codice della creazione dei vari set di dati:

```

real_train_ds = (
    tf.data.Dataset.from_tensor_slices((x_tr, y_tr))
    .map(custom_map_fn, num_parallel_calls=tf.data.AUTOTUNE)
    .shuffle(1000)
    .batch(32)
    .map(mask_padding, num_parallel_calls=tf.data.AUTOTUNE)
    .prefetch(tf.data.AUTOTUNE)
)
train_ds = (
    tf.data.Dataset.from_tensor_slices((x_tr_os, y_tr_os))
    .map(tf_augment, num_parallel_calls=tf.data.AUTOTUNE)
    .map(custom_map_fn, num_parallel_calls=tf.data.AUTOTUNE)
    .shuffle(1000)
    .batch(32)
    .map(mask_padding, num_parallel_calls=tf.data.AUTOTUNE)
    .prefetch(tf.data.AUTOTUNE)
)
val_ds = (
    tf.data.Dataset.from_tensor_slices((x_va, y_va))
    .map(to_dict, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(32)
    .map(mask_padding, num_parallel_calls=tf.data.AUTOTUNE)
    .prefetch(tf.data.AUTOTUNE)
)
test_ds = (
    tf.data.Dataset.from_tensor_slices((x_te, y_te))
    .map(to_dict, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(32)
    .map(mask_padding, num_parallel_calls=tf.data.AUTOTUNE)
    .prefetch(tf.data.AUTOTUNE)
)

```

Come si evince dal codice, si possono notare esattamente quattro dataset. Train_ds, val_ds e test_ds sono quelli utilizzati nell'addestramento e sono derivanti dallo splittaggio di train/val/test dopo aver effettuato già una stratificazione totale. Sono stati tutti generati con tf.data pipeline in streaming. Ho deciso di attuare data augmentation – con la logica vista precedentemente – direttamente on-the-fly sul batch corrente in questione, facendo anche shuffle(1000) di tutti i dati. Quest'ultima è servita, principalmente, a fare in modo che i dati all'interno del dataset fossero mescolati ad ogni iterazione, per evitare il sovrardattamento. Con questa modalità, ogni batch è stato caricato in maniera

progressiva e singola, aumentato localmente ed elaborato runtime. Il processo di normalizzazione e mappatura del dataset (determinazione delle shape dei tensori ecc.) sono stati sempre effettuati runtime. Invece, `real_train_ds`, è il dataset di train che è servito solamente per calcolare correttamente l'aggiornamento del callback `DynamicAlphaCallback` della `DynamicFocalLoss`; perché, per riflettere correttamente la rappresentazione delle classi con la `focal_loss`, serve un riferimento reale del dataset senza che abbia subito operazioni di oversampling o augmentazione. In poche parole, è buona prassi mantenere, sempre, una copia effettiva del dataset originale senza che abbia subito, ancora, alcuna operazione di bilanciamento.

Dopodichè, si è stabilito un numero di epochhe iniziale - scalato da 100 a 50 per testare vari risultati - consono ad indirizzare correttamente l'andamento dell'addestramento e, allo stesso tempo, ho creato vari callback personalizzati per gestire varie funzionalità durante l'addestramento stesso. I callback, in Keras, sono metodi di hook in forma di oggetti osservatori innestabili nel processo di training, per eseguire certe azioni automatiche in momenti prestabiliti, come ad esempio: all'inizio o fine di un'epoca, di un batch o alla fine del training stesso. Nella pratica, servono, prettamente, per automatizzare e personalizzare il processo di training, gestendo salvataggi, adattamenti dinamici dei parametri e monitoraggio avanzato, senza dover riscrivere manualmente tutto il loop di training. I callback che ho utilizzato io sono stati i seguenti:

- **ModelCheckpoint:** Monitorando una metrica desiderata con modalità ‘min’ o ‘max’, è possibile salvare in automatico, su disco, i pesi migliori del modello ogni volta che la metrica in questione migliora. Questo è molto comodo, poiché evita il salvataggio manuale finale del modello dopo l'addestramento e consente di avere in memoria sempre i pesi migliori tra tutti.
- **EarlyStopping:** Meccanismo essenziale in tutti i modelli neurali. Interrompe l'addestramento, ripristinando anche i pesi migliori del modello, se una metrica monitorata non migliora significativamente per N epochhe consecutive. Solitamente, si sceglie di minimizzare la loss di validazione e, quando essa smette di migliorare, tale callback agisce, interrompendo l'addestramento e restituendo in output i pesi migliori elaborati fino a quel momento. Oltre tutto, è utilizzato anche per prevenire **overfitting** in quanto, monitorando una metrica di validazione, se essa non migliora **ma** migliora solo la corrispettiva nel training, l'addestramento viene comunque interrotto.
- **ReduceLROnPlateau:** Già citato precedentemente, perché utilizzato durante la compilazione in combinazione con **WarmUpScheduler** (utilizzato sempre come callback). Serve ad abbassare il learning rate sui punti di stallo o plateau, per consentire al modello di superarli.
- **RocAucCallback:** Callback custom che, durante il training, calcola automaticamente la curva ROC (Receiver Operating Characteristics) e AUC (Area Under the Curve) su un set di validazione alla fine di ogni epoca. È molto utile, soprattutto in un contesto di grande sbilanciamento come quello attuale, dove un'alta accuracy può essere ingannevole e nascondere prestazioni scarse sulle classi meno numerose. Infatti, ROC-AUC misura la capacità del modello di separare positivi e negativi in tutti i possibili threshold di decisione e di avere un report più significativo per problemi di classificazione binaria sbilanciata.

- **DynamicAlphaCallback:** Altro callback personalizzato che regola dinamicamente, durante l’addestramento, il valore di un iper-parametro “alpha” - tipicamente un peso di bilanciamento tra più termini di loss – in funzione dell’andamento delle metriche di training e di validation. Funziona in simbiosi con la loss personalizzata DynamicFocalLoss e il risultato è un training in cui il modello si concentra, in modo adattivo, sul termine di loss che ha più bisogno in quel momento.
- **DeltaUpdateCallback:** Callback personalizzato che mi è servito per gestire l’aggiornamento dinamico di una funzione di loss personalizzata implementata per la regressione: **DynamicHuber**. Esso **modifica dinamicamente**, durante il training, il valore di un parametro “delta” usato in tale loss, oppure in qualunque altro componente del modello che richieda un iper-parametro da adattare man mano che l’errore varia, migliorando stabilità e performance del training.
- **UpdateThresholdCallback:** Ultimo callback personalizzato che, alla fine di ogni epoca, calcola, sul validation set, la soglia ottimale di decisione (best_threshold) per la classificazione binaria e la aggiorna in modo che il modello possa usarla, fin da subito dalla epoca successiva, nei suoi report e – se opportuno – in fase di inferenza interna. Nei problemi sbilanciati è particolarmente utile, perché la soglia binaria di default 0.5 può non essere ottimale e, pertanto, questo aiuta a trovare la soglia migliore al momento migliore.

Di seguito, la definizione in codice dei vari callbacks personalizzati:

```
class DeltaUpdateCallback(tf.keras.callbacks.Callback):
    def __init__(self, val_ds, percentile=95, smoothing=0.2, min_delta=0.05):
        super().__init__()
        self.val_ds = val_ds
        self.percentile = percentile
        self.smoothing = smoothing
        self.min_delta = min_delta
    def on_epoch_end(self, epoch, logs=None):
        errs = []
        for batch in self.val_ds:
            # il val_ds restituisce (x, y), non (x, y, meta)
            if len(batch) == 3:
                x_batch, y_batch, _ = batch
            else:
                x_batch, y_batch = batch
            # fai predict
            preds = self.model.predict(x_batch, verbose=0)

            # estrai l'output "remaining_rett"
            if isinstance(preds, dict):
                out_rett = preds["remaining_rett"]
            else:
                out_rett = preds[0]
```

```

        pred_rett = out_rett.flatten()
        # y_batch è un dict con chiave "remaining_rett"
        true_rett = y_batch["remaining_rett"].numpy().flatten()
        errs.append(np.abs(true_rett - pred_rett))
        errs = np.concatenate(errs)
        raw_delta = np.percentile(errs, self.percentile)

        old_delta = delta_var.numpy()
        smoothed = self.smoothing * raw_delta + (1 - self.smoothing) * old_delta
        smoothed = max(smoothed, self.min_delta)
        delta_var.assign(smoothed)
        print(f"\n→ Aggiornato δ DynamicHuber a {smoothed:.4f}")
    
```

```

class DynamicAlphaCallback(tf.keras.callbacks.Callback):
    """
    Ogni epoch ricalcola alpha_0/alpha_1
    dando peso `minor_weight` (es. 2.0) alla classe minoritaria
    sul dataset *reale* (non oversamplato).
    """

    def __init__(self, train_ds, minor_weight=2.0):
        super().__init__()
        self.train_ds = train_ds
        self.minor_weight = minor_weight

    def on_epoch_begin(self, epoch, logs=None):
        count_0 = count_1 = 0
        # qui il dataset restituisce 3 elementi: (x_batch, y_batch, sample_weight)
        for x_batch, y_batch, _ in self.train_ds:
            # estraggo y_cls dal y_batch
            if isinstance(y_batch, (tuple, list)):
                y_cls = tf.squeeze(y_batch[2], -1).numpy()
            else:
                y_cls = tf.squeeze(y_batch["next_maint_type"], -1).numpy()
            valid = np.isin(y_cls, [0,1])
            count_0 += int((y_cls[valid] == 0).sum())
            count_1 += int((y_cls[valid] == 1).sum())
        # se manca una classe
        if count_0 == 0 or count_1 == 0:
            dynamic_alpha.assign([0.5, 0.5])
            print(f"\nEpoch {epoch+1} – solo una classe: α = [0.5, 0.5]")
            return
        # decido quale è minoritaria
        if count_0 < count_1:

```

```

        raw = np.array([self.minor_weight, 1.0], dtype=np.float32)
        note = "classe 0 minoritaria"
    else:
        raw = np.array([1.0, self.minor_weight], dtype=np.float32)
        note = "classe 1 minoritaria"
    new_alpha = raw / np.sum(raw)
    dynamic_alpha.assign(new_alpha)
    print(f"\nEpoch {epoch+1} - Dynamic alpha ({note}): {dynamic_alpha.numpy()}")

```

```

class RocAucCallback(tf.keras.callbacks.Callback):
    def __init__(self, val_data):
        super().__init__()
        self.val_data = val_data
    def on_epoch_end(self, epoch, logs=None):
        y_trues = []
        y_preds = []
        for batch in self.val_data:
            # unpack (x, y) o (x, y, meta)
            if len(batch) == 3:
                x_batch, y_batch, _ = batch
            else:
                x_batch, y_batch = batch
            preds = self.model.predict(x_batch, verbose=0)
            # Se è dict, prendi direttamente l'array
            if isinstance(preds, dict):
                out_class = preds["next_maint_type"]
            else:
                # altrimenti assume che sia [rett, sost, class]
                out_class = preds[2]

            preds_cls = out_class.flatten()
            true_cls = y_batch["next_maint_type"].numpy().flatten()

            y_preds.append(preds_cls)
            y_trues.append(true_cls)

        y_trues = np.concatenate(y_trues)
        y_preds = np.concatenate(y_preds)

        mask = (y_trues != -1)
        valid_true = y_trues[mask]
        valid_pred = y_preds[mask]

```

```

if np.unique(valid_true).size < 2:
    print(f"\nEpoch {epoch+1} ROC-AUC: N/D (una sola classe presente)")
else:
    auc = roc_auc_score(valid_true, valid_pred)
    print(f"\nEpoch {epoch+1} ROC-AUC: {auc:.4f}")
# 3) Callback per aggiornare threshold_var dopo la prima epoca
class UpdateThresholdCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        if epoch == 0:
            val_preds, val_targets = [], []
            for x_batch, y_dict, _ in val_ds:
                p = self.model.predict(x_batch, verbose=0)[ "next_maint_type" ].flatten()
                t = y_dict[ "next_maint_type" ].numpy().flatten()
                mask = (t != -1)
                val_preds.append(p[mask])
                val_targets.append(t[mask])
            val_preds = np.concatenate(val_preds)
            val_targets = np.concatenate(val_targets)
            prec, rec, th = precision_recall_curve(val_targets, val_preds)
            f1 = 2 * (prec * rec) / (prec + rec + 1e-8)
            best_thr = th[np.nanargmax(f1)]
            K.set_value(threshold_var, best_thr)
        print(f"\n→ Soglia aggiornata a {best_thr:.3f} per le epoche successive")

```

Mentre, per quanto riguarda l'addestramento finale vero e proprio, è stato affrontato così:

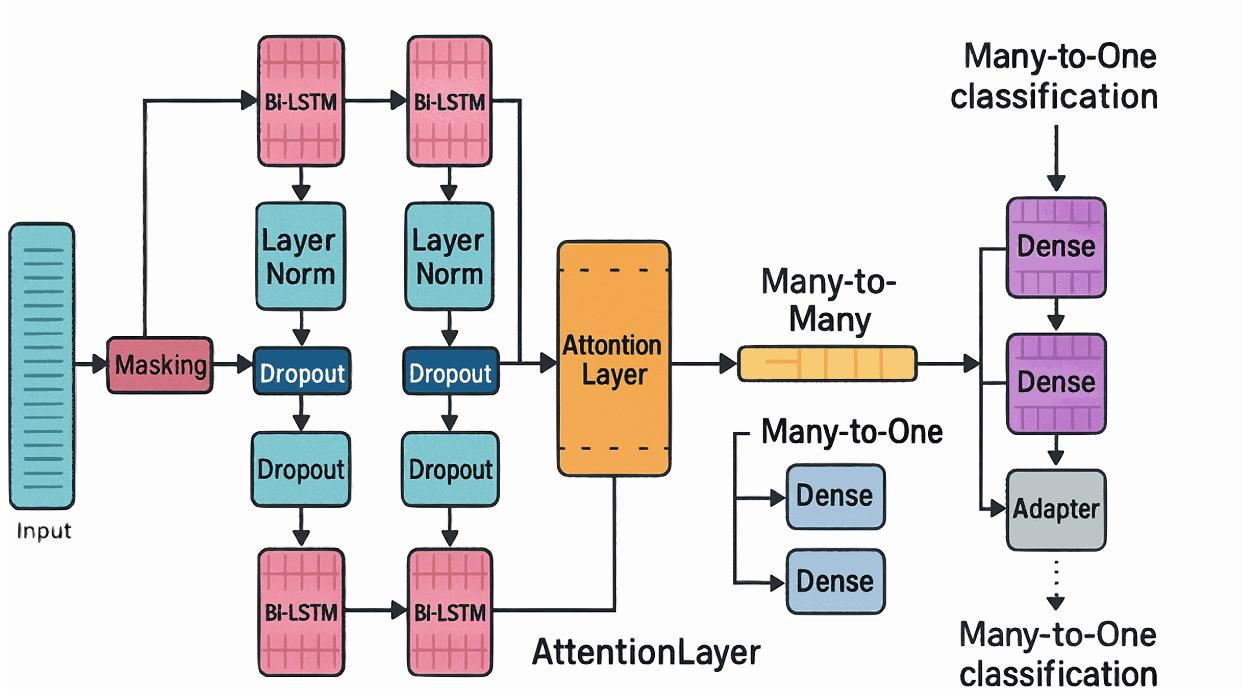
```

history = model_combined.fit(
    train_ds,
    validation_data=val_ds,
    epochs=50,
    callbacks=[
        early_stopping,
        delta_cb,
        lr_warmup, lr_plateau,
        checkpoint,
        dynamic_alpha_callback,
        roc_auc_callback,
        UpdateThresholdCallback()
    ]
)

```

Dove ho implementato tutti i callbacks di cui ho parlato, ho associato il train set e il val set in modo corretto e assegnato anche un numero di epoche pari a 50, per gestire correttamente l'andamento dell'addestramento.

Per concludere, l'immagine vettoriale seguente, generata tramite ChatGPT di OpenAI, rappresenta, a grandi linee, il modello neurale che ho realizzato ed utilizzato:



3.3.3 Funzioni di loss e metriche

LOSS FUNCTIONS

Un altro aspetto fondamentale dell'addestramento del modello è stato l'utilizzo di funzioni di loss personalizzate, che hanno consentito di adattare meglio l'addestramento in fase di training. Come è certamente noto, la funzione di loss generale deve essere minimizzata per garantire un corretto apprendimento da parte del modello. In un problema di manutenzione predittiva così complesso però, i valori nel dataset, spesso subiscono variazioni improvvise e, non sempre, l'adattamento da parte del modello è lineare e regolare. Proprio per questo motivo, ho deciso di introdurre due tipologie di loss function molto efficienti con i problemi sbilanciati:

- **Dynamic Huber:** Funzione di loss personalizzata dinamica basata sulla funzione di Huber statica, utilizzata per il corretto andamento dei problemi regressivi. Innanzitutto, la Huber loss statica è un

compromesso e un ibrido fra MSE - sensibile agli outlier - e MAE - più robusta ma meno levigante dei gradienti - ed è definita per ciascun errore $\epsilon = y_{\text{pred}} - y_{\text{true}}$ nel modo seguente:

$$L_\delta(\epsilon) = \begin{cases} \frac{1}{2} \epsilon^2, & |\epsilon| \leq \delta, \\ \delta (|\epsilon| - \frac{1}{2}\delta), & |\epsilon| > \delta. \end{cases}$$

Dove:

- Per $|\epsilon|$ piccoli, agisce come **MSE** (gradiente lineare in ϵ).
- Per $|\epsilon|$ grandi, si comporta come **MAE** (gradiente costante $\pm \delta$).

Questo, naturalmente, accade a livello statico, ma cosa succede se tale funzione statica diventa dinamica?

Nella pratica, a mano a mano che l'errore medio cala, la taglia δ ottimale può variare. Questo può portare a delineare uno scenario in cui:

- All'inizio il modello fa errori grandi, per cui serve un δ più alto per non essere troppo "punitivo".
- Man mano che il modello inizia a convergere, gli errori, se tutto è corretto, diminuiscono. Per questo motivo, conviene ridurre δ per far diventare "outlier" anche deviazioni minime o più piccole, rendendo la loss più sensibile e più granulare e aumentando, di fatti, l'efficienza dell'addestramento del modello.

La DynamicHuber è associata, inoltre, ad un callback custom (*DeltaUpdateCallback*) che, ad ogni epoca, ricalcola δ in base alla distribuzione attuale degli errori e aggiorna, quindi, la formula con il nuovo valore. In sostanza, l'ottimizzatore, per minimizzare tale loss correttamente, cerca di ridurre il valore medio di $L\delta(\epsilon)$, dove però δ cambia dinamicamente per adattarsi all'entità degli errori riscontrati.

```
loss={
    "remaining_rett": DynamicHuber(name="huber_rett"),
    "remaining_sost": DynamicHuber(name="huber_sost"),
```

Con classe personalizzata DynamicHuber:

```
delta_var=tf.Variable(1.0,trainable=False, dtype=tf.float32, name="delta_hubert")
class DynamicHuber(tf.keras.losses.Loss):
    def call(self, y_true, y_pred):
        err = y_true - y_pred
        abs_err = tf.abs(err)
        # condizione |e| <= delta
        is_small = abs_err <= delta_var
```

```

small_loss = 0.5 * tf.square(err)
large_loss = delta_var * (abs_err - 0.5 * delta_var)
return tf.where(is_small, small_loss, large_loss)

```

Tale loss personalizzata è stata molto importante nel corretto svolgimento dell’addestramento del ramo regressivo del mio modello. Dato che la funzione Huber è una eccellenza nell’ambito regressivo, ho deciso di sperimentare prima quella statica e poi quella dinamica, riscontrando un netto miglioramento delle prestazioni con l’adattamento dinamico.

- **Dynamic Focal Loss:** Altra funzione di loss personalizzata, che è nata principalmente per aiutare le problematiche di classificazione sbilanciata. Anche qui è necessario effettuare un preambolo inherente alla Focal Loss statica. La Focal Loss è nata, prettamente, per la **classificazione sbilanciata** ed è definita su ciascun esempio di classe positiva/negativa con probabilità predetta p_t come:

$$\text{FL}_\gamma(p_t) = -(1 - p_t)^\gamma \log(p_t)$$

Infatti, se le classi a disposizione del dataset sono molto sbilanciate, il termine $(1 - p_t)^\gamma$ enfatizza le istanze **già classificate correttamente** – quindi p_t alto – ponendo più peso sugli esempi difficili (quelli poco riconosciuti fino a tale momento) e, quindi, p_t basso.

Il parametro γ (tipicamente tra 1 e 3) regola **quanto e quanto intensamente** focalizzarsi sui casi difficili.

Spesso, però, la difficoltà relativa degli esempi nel dataset cambia, in corso di opera, durante il training. Infatti, se all’inizio molti esempi sono difficili, serve un basso γ per non soffocare il segnale di perdita complessivo. Mentre, a livello di addestramento prolungato, un γ più alto enfatizza, sicuramente meglio, questi pochi casi rari.

Tra l’altro, la Dynamic Focal Loss aggiorna γ in base a metriche monitorate – come l’accuracy - o in base alla curva di apprendimento che si sta determinando. Infatti, se la precision o la recall aumentassero troppo rapidamente, si **aumenterebbe** di conseguenza anche γ per focalizzare il modello sugli errori rimanenti. Mentre, se il training diventasse instabile o oscillatorio e la loss smettesse di decrescere, si **diminuirebbe** γ per smussare la loss e favorire, pertanto, la convergenza. Per cui, anche la Dynamic Focal Loss deve essere minimizzata e l’ottimizzatore deve cercare di ridurre il valore medio di:

$$(1 - p_t)^{\gamma(t)} [-\log(p_t)]$$

Dove $\gamma(t)$ è il parametro aggiornato dinamicamente all’epoca t . Grazie a tali adattamenti “dinamici”, il training complessivo risulta più stabile, meno sensibile a errori grossolani iniziali o a class imbalance variabile e, quindi, più rapido e preciso nella convergenza.

METRICHE

In un problema multi-output di classificazione binaria sbilanciata e regressione numerica duale, il monitoraggio della metrica di accuracy è pressoché fuorviante poiché, se ipoteticamente nel validation set si avesse il 95% di classi positive e solo il 5% di classi negative, allora l'accuracy sarebbe elevatissima perché il modello riconoscerebbe sempre la classe maggioritaria; tuttavia questo **NON** implicherebbe che il modello sia perfettamente in grado di distinguere correttamente le due classi in questione anzi, confermerebbe il reale sbilanciamento del problema e la sua difficoltà nel riconoscerlo. Proprio per questo motivo, se da una parte il problema della regressione è stato affrontato, metricamente, in maniera standard e semplice, lo stesso non si è potuto affermare per la classificazione, dove ho dovuto creare e monitorare quattro metriche personalizzate sulla base di quelle già esistenti. Procediamo con ordine:

1. **REGRESSIONE:** A livello regressivo il problema non è stato troppo diverso da un qualsiasi problema standard di regressione, fatta eccezione per la presenza, in parallelo, di due branch regressivi e, quindi, della loro convergenza in contemporanea e non del singolo caso. Ho comunque deciso di utilizzare le due classiche metriche regressive:

- **MAE** (Mean Absolute Error):

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

È la **media delle deviazioni assolute tra valore predetto e valore reale atteso**. È molto robusta agli outlier, infatti ogni errore influisce, su di essa, in modo lineare. Un outlier con errore doppio pesa esattamente il doppio sul MAE. Viene utilizzata quando è necessaria equità tra tutti gli errori, senza far arrancare troppo il modello per pochi outlier estremi e quando l'unità di misura dell'errore è immediatamente interpretabile con la medesima scala dei valori ottenuti. Inoltre, i gradienti che agiscono sono costanti. Questo offre un vantaggio considerevole poiché, in tal modo, la derivata rispetto all'errore vale +1 oppure -1 e, quindi, il training è meno rumoroso intorno ad errori grandi.

- **MSE** (Mean Squared Error):

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

Precisamente, è la **media dei quadrati degli errori dei valori predetti**. Si calcola il quadrato di tali errori e poi si utilizza RMSE (Root Mean Squared Error), ovvero la radice quadrata, per tornare all'unità di misura originale. È molto sensibile agli outlier, infatti errori grandi pesano in maniera quadratica; ad esempio, se un outlier ha errore doppio, contribuisce con un peso quattro volte superiore. Analogamente però, anche i gradienti sono proporzionali all'errore; quindi, allo stesso

tempo, il modello emette una spinta più elevata per correggere errori grandi. Metrica estremamente utile per penalizzare severamente grandi deviazioni dal valore originale, soprattutto nei modelli in cui si ha una convergenza veloce verso media-zero, poiché in essi il gradiente cresce con l'errore.

Nel caso del mio problema, ho deciso di utilizzarle entrambe. Più specificatamente, ho utilizzato MAE per i valori target di Remaining_rett, poiché sono valori più contenuti e il gradiente deve propagarsi con precisione; mentre, a livello di Remaining_sost, ho deciso di utilizzare MSE, poiché ho avuto bisogno di penalizzare fortemente grandi deviazioni dal valore totalitario iniziale. Infatti, essendo Remaining_sost un valore molto grande, ipoteticamente, una piccola variazione non avrebbe creato troppi problemi... per questo MSE è stata più indicata.

2. **CLASSIFICAZIONE:** A livello di metriche, questa è stata la parte più complessa da implementare, poiché il problema dello sbilanciamento è stata un'ardua sfida da portare a termine. Ad ogni modo, qualunque problema di classificazione sbilanciata che si rispetti, non fa affidamento sulla metrica di Accuracy perché, come detto precedentemente, può essere fuorviante e ingannevole. La metrica di riferimento è, per l'appunto:

ROC-AUC (Receiving Operating Characteristics – Area Under the Curve): In un problema di classificazione binaria, il modello restituisce una probabilità p di appartenere alla classe positiva. Per trasformare tale probabilità in una soglia di decisione (0 o 1) si usa una soglia (threshold) t che, di default, vale 0.5. Se la threshold, invece, viene adattata a qualsiasi altro valore tra 0 e 1, la soglia di distinzione delle due classi varia di conseguenza.

Variando t tra 0 e 1, per la ROC si ottengono diversi punti di compromesso tra:

- True Positive Rate (TPR):

$$\text{TPR}(t) = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- False Positive Rate (FPR):

$$\text{FPR}(t) = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Invece, AUC non è altro che l'integrale - quindi l'area - sotto la curva ROC appena originata, tenendo in considerazione, naturalmente, il tasso di falsi positivi e veri positivi menzionati qui sopra:

$$\text{AUC} = \int_0^1 \text{TPR}(f) d(\text{FPR})$$

Questa metrica è stata veramente molto utile in questo contesto per i seguenti vantaggi:

- **Threshold-independent**: Valuta il modello su tutti i possibili cut-off, per cui non dipende mai da una singola soglia.
- **Robusta al class imbalance**: considera tassi relativi, non assoluti.
- **Interpretabile**: AUC consente di fornire la probabilità di corretto ordinamento tra un positivo e un negativo.

Nel mio problema, ho dovuto effettuare un passo ulteriore, per rendere l'AUC di Keras adattabile al mio contesto ed evitare che prendesse in considerazione, nella misura della predizione, i valori “-1” di NextMaintenanceType inerenti ai record di produzione intermedi delle sequenze. Chiaramente, le uniche classi ammissibili erano 0 e 1; “-1” era solo un placeholder per gestire correttamente i record che non avevano correlazione e importanza nella predizione di NextMaintenanceType. Quindi, ho originato una funzione di MaskedAUC, che andasse a “mascherare” i placeholder non desiderati e gestisse direttamente i sample weights nel modello. Allo stesso tempo, anche gli zeri delle sequenze derivanti dallo zero-padding sono stati mascherati allo stesso modo.

```
class MaskedAUC(tf.keras.metrics.AUC):  
    def __init__(self, name="masked_auc", **kwargs):  
        super().__init__(name=name, **kwargs)  
    def update_state(self, y_true, y_pred, sample_weight=None):  
        # costruisco la mask di validità  
        mask = tf.not_equal(tf.reshape(y_true, [-1]), -1)  
        # appiattisco y_true, y_pred e sample_weight  
        y_true_flat = tf.reshape(y_true, [-1])  
        y_pred_flat = tf.reshape(y_pred, [-1])  
  
        # filtro y_true e y_pred  
        valid_true = tf.boolean_mask(y_true_flat, mask)  
        valid_pred = tf.boolean_mask(y_pred_flat, mask)  
  
        # filtro anche sample_weight se presente  
        if sample_weight is not None:  
            sw_flat = tf.reshape(sample_weight, [-1])  
            sample_weight = tf.boolean_mask(sw_flat, mask)  
  
        # chiama il super passando i pesi filtrati  
        return super().update_state(  
            valid_true,  
            valid_pred,  
            sample_weight=sample_weight  
        )
```

Le altre metriche prese in considerazione e personalizzate per funzionare correttamente col dataset fornito e col modello corrente erano identiche a quelle classiche, ma con la differenza che sono state adattate al callback di `UpdateThresholdCallback` e quindi al calcolo della best threshold di classificazione di ogni epoca per trovare il risultato - di selezione - dinamico migliore per ogni classe distinta:

- RECALL (“True Positive Rate” o “Sensitivity”): valore tra 0 e 1 che indica la capacità del modello di riconoscere veramente i veri positivi tra tutti i positivi presenti. Più è alta, più il modello riesce a distinguere correttamente tutti i veri positivi.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- PRECISION (“Positive Predictive Value”): valore tra 0 e 1 che indica la capacità del modello di riconoscere davvero il vero positivo ogni qualvolta che identifica uno dei valori come tale.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- F1-SCORE: valore tra 0 e 1 che indica la media armonica di Precision e Recall. Penalizza fortemente se uno dei due valori è basso, quindi spinge il modello a mantenere un buon equilibrio tra Precision e Recall. In uno scenario di sbilanciamento come quello attuale, è una metrica molto importante per la classificazione.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Quindi, l'obiettivo principale è ottenere valori di queste metriche molto elevati, sia in validazione che in training. In questo modo, si è certi che il modello stia imparando a dividere esattamente e perfettamente le due classi, anche se una è sbilanciata verso l'altra o viceversa.

3.3.4 Predizioni e funzione di inferenza dinamica

L'ultima parte del codice del main core del programma, riguarda la parte di inferenza dinamica delle predizioni effettuate e della valutazione del modello su dati di test, mai visti prima durante l'addestramento. Dopo aver generato il modello, creato le funzioni di perdita e portato a termine il training con successo, è stato il momento di verificarne la reale efficacia e la bontà delle predizioni restituite. La prima azione da svolgere, dopo aver gestito correttamente l'addestramento stesso, è stata utilizzare la direttiva **model.evaluate()** sul test set, per verificare immediatamente come il modello si è comportato su dati di test mai visti prima.

```
results = model_combined.evaluate(test_ds, return_dict=True)
print("Test-set performance:", results)
```

E, subito dopo, ho effettuato una calibrazione della best threshold sul validation set, per garantire correttamente che, la divisione delle classi e il riconoscimento di esse, riflettessero correttamente la distribuzione sbilanciata iniziale. In questo modo, si sarebbe trovata la soglia perfetta di separazione tra le due classi e si sarebbe ottenuta una predizione più veritiera sulla distribuzione finale. Anche in questo caso, ho utilizzato un masking dei NextMaintenanceType uguali a “-1” per non corrompere la qualità della distribuzione delle etichette.

```
# Calibrazione del threshold sul validation set
# 1) Raccolgo predizioni e target
val_preds, val_targets = [], []
for x_batch, y_dict, _ in val_ds:
    # invoco il modello in modalità inference, senza alterare lo stato
    preds = model_combined(x_batch, training=False)
    p_cls = preds["next_maint_type"].numpy().flatten()
    t_cls = y_dict["next_maint_type"].numpy().flatten()
    # filtro i passi con target == -1
    mask = (t_cls != -1)
    val_preds.append(p_cls[mask])
    val_targets.append(t_cls[mask])
val_preds = np.concatenate(val_preds)
val_targets = np.concatenate(val_targets)
```

Dopodiché, ho costruito le due curve fondamentali per capire esattamente **come** e **dove** scegliere la corretta rappresentazione delle due classi:

- **PR-AUC:** Ovvero la curva tracciata dalla combinazione media della metrica AUC insieme alla Precisione e alla Recall, che sono esattamente le metriche di interesse per i problemi sbilanciati.
- **ROC:** Come detto precedentemente, curva fondamentale per collocare - nell'esatto punto - la distribuzione delle classi. Lavora in simbiosi con AUC.

Per calcolare la soglia ottimale, ho utilizzato l'**indice di Youden**, il quale è un criterio per scegliere il cut-off ottimale sulla curva ROC nel problema di classificazione binaria, massimizzando il compromesso tra recall e specificity. Tale indice va a massimizzare $\text{TPR} + \text{TNR} - 1$, cioè la differenza tra **TPR** e **FPR**. Youden è molto utile perché va a bilanciare, pertanto, le metriche sopracitate e quindi si massimizza la somma di veri positivi e veri negativi normalizzati. Oltre tutto, è indipendente dalla prevalenza del class balance del dataset. È un criterio threshold-independent perché si basa sui valori di TPR e FPR calcolati su tutta la gamma di possibili t. La soglia Youden funziona nella modalità seguente:

Per ciascun possibile valore di soglia t, si calcolano Recall (True Positive Rate TPR) e Specificity (True Negative Rate, TNR) e, da essi, si deriva poi l'**indice di Youden**:

$$J(t) = \text{TPR}(t) + \text{TNR}(t) - 1 = \text{TPR}(t) - \text{FPR}(t)$$

Da cui: $\text{FPR}(t) = 1 - \text{TNR}(t)$.

I valori di J vanno da -1 (caso peggiore) a $+1$ (caso migliore). Si deduce, successivamente, che la **soglia decisionale Youden** migliore è:

$$t^* = \arg \max_t J(t).$$

Ovvero, il valore massimo interpretabile dell'indice stesso.

A livello di codice, è stato implementato così:

```
# 2) Costruisco la curva ROC
fpr, tpr, roc_thresholds = roc_curve(val_targets, val_preds, pos_label=1)
# 1.2 Curva Precision-Recall
prec, rec, thresholds = precision_recall_curve(val_targets, val_preds,
pos_label=1)
# 1.3 Trovo la soglia che massimizza F1
f1_scores = 2 * (prec * rec) / (prec + rec + 1e-8)
best_idx = np.argmax(f1_scores)
b_threshold = thresholds[best_idx] # soglia che massimizza F1
print(f"\n→ Soglia ottimale per F1 sulla val: {b_threshold:.3f}\n(F1={f1_scores[best_idx]:.3f})")

# 1.4 Calcolo PR-AUC (Average Precision)
pr_auc = average_precision_score(val_targets, val_preds)
print(f"PR-AUC (average precision) sulla val: {pr_auc:.3f}")
```

```

# 3) Calcolo Youden's J = TPR - FPR e trovo il massimo
youden_j = tpr - fpr
ix = np.argmax(youden_j)
best_threshold = roc_thresholds[ix]
print(f"\n→ Soglia ottimale (Youden J) = {best_threshold:.3f} "
      f"(TPR={tpr[ix]:.3f}, 1-FPR={1-fpr[ix]:.3f})")

```

Ho tracciato le due curve fondamentali e, subito dopo, ho calcolato i due valori delle soglie di interesse (Youden's J e b_threshold per massimizzare F1-score e quindi il rapporto tra Precision e Recall). Successivamente, con queste soglie definite, ho delineato tutte le predizioni su validation e test set eseguite dal modello, stampando anche la relativa **matrice di confusione** e il **classification report**:

```

# 4) Predizioni su validation set, binarizzazione e stampa report
val_pred_labels = (val_preds >= best_threshold).astype(int) # Youden's
accuracy = accuracy_score(val_targets, val_pred_labels)
precision = precision_score(val_targets, val_pred_labels, zero_division=0)
recall = recall_score(val_targets, val_pred_labels, zero_division=0)
f1 = f1_score(val_targets, val_pred_labels, zero_division=0)
auc_val = roc_auc_score(val_targets, val_preds)
print("Confusion Matrix:")
print(confusion_matrix(val_targets, val_pred_labels))
print("Classification Report:")
print(classification_report(val_targets, val_pred_labels))

# Stessa identica cosa su test set ma provando ad utilizzare PR-AUC
test_preds, test_targets = [], []
for x_batch, y_dict, sample_weight in test_ds:
    preds = model_combined.predict(x_batch, verbose=0)
    p = preds["next_maint_type"].flatten()
    test_preds.append(p.flatten())
    test_targets.append(y_dict["next_maint_type"].numpy().flatten())
test_preds = np.concatenate(test_preds)
test_targets = np.concatenate(test_targets)

# Binarizzo
test_pred_labels = (test_preds >= b_threshold).astype(int) # PR-AUC

# Matrice di confusione e report
cm = confusion_matrix(test_targets, test_pred_labels, labels=[0,1])
tn, fp, fn, tp = cm.ravel()
specificity = tn / (tn + fp) if (tn + fp)>0 else 0.0

```

```

print("\nTest - Confusion matrix (next_maint_type):")
print(f" TN={tn}  FP={fp}\n FN={fn}  TP={tp}")
print(f"\nTest - Specificità: {specificity:.3f}")
print("\nClassification report:")
print(classification_report(test_targets, test_pred_labels, digits=3))
mse_test = mean_squared_error(all_true_counts, all_pred_counts)
mae_test = mean_absolute_error(all_true_counts, all_pred_counts)
print(f"- Test MSE (countdown, ultimo ciclo): {mse_test:.2f}")
print(f"- Test MAE (countdown, ultimo ciclo): {mae_test:.2f}")

```

Pertanto, arrivati a questo punto, l'addestramento era stato svolto in maniera consona e le prestazioni del modello derivanti erano state calcolate. Mancava però un punto fondamentale, ovvero quello di implementare una funzione di inferenza dinamica, che andasse a mostrare, in tempo reale, i veri e propri risultati, della rete neurale, nelle predizioni effettuate sul test set mai visto prima. Finora, avevo ottenuto le percentuali del tasso di successo del modello, ma non avevo ancora validato la rete a livello di risultato e a livello di confronto tra **valore atteso vs valore predetto**. Per compiere tutto ciò, ho creato la seguente funzione:

```

def dynamic_predict_full_sequence(
    model, sequences, next_events_true,
    true_sub_total, true_cycle_total,
    avg_hours_rett, avg_hours_sost,
    expected_subcycle_value, expected_cycle_value,
    threshold=0.5, metas=None
):
    preds = [];
    t = 0
    counter_r = true_sub_total
    counter_s = true_cycle_total

    for idx, seq in enumerate(sequences):
        cycle, sub, L, seq_id, max_sub_h, max_cum_h = metas[idx]
        ne_true = next_events_true[idx]
        # ora divido i casi
        if ne_true == -1:
            # sottociclo incompleto → ciclo incompleto: uso il valore dallo storico
            real_C = int(expected_cycle_value)
            real_T = int(expected_subcycle_value)
        else:
            # sottociclo completo: posso usare il valore reale del ciclo
            real_C = int(true_cycle_total)
            real_T = L - 1

```

```

# contatori float iniziali
cnt_r_f = float(real_T)
cnt_s_f = float(real_C)

raw_preds = model.predict(seq[np.newaxis], verbose=0)
if isinstance(raw_preds, dict):
    pr = raw_preds["remaining_rett"]
    ps = raw_preds["remaining_sost"]
    pc = raw_preds["next_maint_type"]
else:
    pr, ps, pc = raw_preds

pr = pr.flatten()
ps = ps.flatten()
p_class = float(pc.flatten()[0])
#p_class = float(pc[0,0])

# scelgo next-event
if ne_true==0: nxt = "SOSTITUZIONE"
elif ne_true==1: nxt = "RETTIFICA"
else: nxt = "RETTIFICA" if p_class>=threshold else "SOSTITUZIONE"

# produzione
for fr, fs in zip(pr, ps):
    # delta frazione
    delta_r = max((cnt_r_f/real_T) - fr, 0.0) * real_T
    cnt_r_f = max(cnt_r_f - delta_r, 0.0)
    delta_s = max((cnt_s_f/real_C) - fs, 0.0) * real_C
    cnt_s_f = max(cnt_s_f - delta_s, 0.0)

    cnt_r = int(cnt_r_f)
    cnt_s = int(cnt_s_f)
    counter_s = max(counter_s - 1, 0)
    preds.append({
        'cycle': cycle, 'subcycle': sub, 'seq_id': seq_id,
        'time_step': t,
        'remaining_rett': cnt_r,
        'remaining_sost': counter_s,
        'predicted_days_rett': math.ceil(cnt_r*avg_hours_rett/24) if
cnt_r>0 else 0,
        'predicted_days_sost': math.ceil(counter_s*avg_hours_sost/24) if
cnt_s>0 else 0,
        'class_probability': p_class,
    })

```

```

        'event_type': "PRODUZIONE",
        'next_event': nxt
    })
    t += 1
# manutenzione (reset)
if ne_true in (0,1):
    if nxt=="RETTIFICA":
        new_r, new_s = real_T, counter_s
    else:
        new_r, new_s = real_T, real_C

preds.append({
    'cycle': cycle, 'subcycle': sub, 'seq_id': seq_id,
    'time_step': t,
    'remaining_rett': int(new_r),
    'remaining_sost': int(new_s),
    'predicted_days_rett': math.ceil(new_r*avg_hours_rett/24) if
new_r>0 else 0,
    'predicted_days_sost': math.ceil(new_s*avg_hours_sost/24) if
new_s>0 else 0,
    'class_probability': p_class,
    'event_type': "MANUTENZIONE",
    'next_event': nxt
})
t += 1
# resetto i contatori solo qui
cnt_r_f = new_r
cnt_s_f = new_s
counter_r = new_r
counter_s = new_s
# se ne_true == -1, non faccio nulla: rimango dentro al ciclo aperto
return preds

```

Questa funzione è stata essenziale per andare a mostrare i risultati veri e propri che il modello ha redatto, sulla base del dataset passato inizialmente come input. Infatti, tale metodo svolge, in un unico passaggio, sia la parte di inferenza sul modello sia la logica che trasforma le sue uscite in predizioni utili per il risultato finale, seguendo l'evoluzione di più sequenze di input e gestendo il reset dei contatori ad ogni evento di manutenzione imminente. I passaggi principali seguiti sono stati:

- Inizializzazione dei contatori:** All'inizio ho preparato due contatori generali (counter_r, counter_s) e due contatori flottanti per sottociclo (cnt_r_f) e ciclo (cnt_s_f), inizializzati rispettivamente ai valori **reali** di durata del sottociclo e del ciclo dell'ultimo evento noto. Questi valori rappresentano quanta vita residua - in timestep di sequenza - resta prima di dover eseguire ciascun evento di manutenzione.

2. **Scorrimento delle sequenze:** Il codice itera prettamente su ogni “seq” nella lista “sequences”. Quindi, ad ogni sequenza corrispondono:
 - La vera label ne_true del prossimo evento manutentivo, che è 0 oppure 1 se il ciclo o sottociclo è arrivato al termine, altrimenti vale -1.
 - I metadati metas[idx] da cui si ricava l’identificativo del numero del ciclo e sottociclo corrispondente, la lunghezza di essi e i valori di soglia storici medi.
3. **Determinazione della lunghezza “reale”:** Se ne_true = -1 significa che ancora non si è verificato un evento di manutenzione che ha determinato la fine del ciclo o sottociclo corrente; per cui, in questo caso, si utilizzano - come riferimenti - i valori storici medi ottenuti dai sottocicli e cicli completi.
4. **Calcolo delle predizioni:** Grazie alla chiamata a **model.predict** per ogni metadata, si ottengono due serie di residui (rettifica e sostituzione) e la probabilità del verificarsi del prossimo evento manutentivo successivo; dopodiché, ad ogni timestep, si aggiornano i due contatori flottanti sottraendo la frazione consumata in base alle predizioni, si convertono questi valori in giorni e si memorizza un record “PRODUZIONE” per indicare l’evento produttivo. Allo stesso modo, se il ciclo o sottociclo è completo, si aggiunge invece un record “MANUTENZIONE” con i contatori resettati e si riparte da quei valori massimi.

Il risultato finale è una lista ordinata di dizionari che traccia la mappatura dell’evoluzione completa di vita residua e manutenzioni correlate su tutte le sequenze, per ogni timestep, del piano martire attuale. Nella pipeline del main è stata chiamata in questo modo:

```
# _____
# 5) VALIDAZIONE “PER CICLO” SU UN TEST SET MAI VISTO
# _____

# 5.1 SELEZIONO DIRETTAMENTE L’ULTIMO CICLO DISPONIBILE (COMPLETO O INCOMPLETO)
last_cycle = df_prepared['cycle_id'].max()
df_last = df_prepared[df_prepared['cycle_id'] == last_cycle]

# 5.2 creo sequenze SOLO per l’ultimo ciclo
seq_x_last, seq_y_last, metas_last = create_representative_sequences(
    df_last, features, target, factor=1.0
)

# 5.3 inferenza “per ciclo”
# idxs_last = tutti gli indici di metas_last, già ordinati per subcycle
seqs_last      = seq_x_last
next_true_last = [int(y[-1,2]) for y in seq_y_last]

# 5.4 calcolo init_sub / init_cyc
if next_true_last[0] == -1:
    init_sub = expected_subcycle_total
```

```

else:
    init_sub = metas_last[0][4]

if 0 not in next_true_last:
    init_cyc = expected_cycle_total
else:
    init_cyc = max(m[5] for m in metas_last)

# 5.5 chiamo dynamic_predict_full_sequence
dynamic_predictions = dynamic_predict_full_sequence(
    model_combined,
    sequences=seqs_last,
    next_events_true=next_true_last,
    true_sub_total=init_sub,
    true_cycle_total=init_cyc,
    avg_hours_rett=avg_hours_rett,
    avg_hours_sost=avg_hours_sost,
    expected_subcycle_value=expected_subcycle_total,
    expected_cycle_value=expected_cycle_total,
    threshold=best_threshold,
    metas=metas_last
)

```

Così facendo, si è delineata una situazione in cui si è dichiarata inferenza dinamica **sempre** sull'ultimo ciclo di vita in possesso (che è, per l'appunto, il ciclo di vita del piano attualmente montato sulla macchina) e si sono redatte le predizioni direttamente su di esso, completo o incompleto che sia. Successivamente, ho voluto implementare anche una parte di “**sanity-check**”, utilizzando le **dynamic_predictions** ottenute, per verificare che i valori attesi combaciassero, quanto più possibile, con i valori predetti dal modello. Il metodo utilizzato è stato il seguente:

```

# 6) implemento logica di sanity check “per ciclo e sottociclo” per raccogliere
true/pred counts e next-event
all_true_counts, all_pred_counts = [], []
true_events, pred_events       = [], []
for sub_i, sub_meta in enumerate(metas_last):
    L = seqs_last[sub_i].shape[0]
    # trova il primo indice di produzione per questo sottociclo
    start = next(
        j for j, rec in enumerate(dynamic_predictions)
        if rec['cycle']==last_cycle
            and rec['subcycle']==sub_meta[1]
            and rec['event_type']=='PRODUZIONE'),

```

```

None
)
if start is None:
    continue
preds_sub = dynamic_predictions[start:start+L]

# conteggi regressivi
true_cnt = list(range(L, 0, -1))
pred_cnt = [p['remaining_rett'] for p in preds_sub]
all_true_counts.extend(true_cnt)
all_pred_counts.extend(pred_cnt)

# classification next-event: solo per l'ultimo sottociclo
if sub_i == len(metas_last)-1:
    true_events.append(
        next_true_last[sub_i] if next_true_last[sub_i] in (0,1) else -1
    )
    avg_prob = np.mean([p['class_probability'] for p in preds_sub])
    pred_events.append(0 if avg_prob < best_threshold else 1)

# Ricalibrazione eventi di manutenzione e inizializzazione del mapping storico
# mapping numerico → stringa per uniformare con rec['next_event']
label_map = {0: "SOSTITUZIONE", 1: "RETTIFICA"}

# 0) costruisco le transizioni “prev → next” a partire dallo storico dei cicli
transition_counts = defaultdict(Counter)
for cycle_id, grp in df_manutenzione.groupby('cycle_id'):
    # ordino per timestamp e prendo solo i tipi di manutenzione
    seq = grp.sort_values('Timestamp')['MaintenanceType'].tolist()
    # conto quante volte, dopo prev, viene nxt
    for prev, nxt in zip(seq, seq[1:]):
        transition_counts[prev][nxt] += 1

# 1) per ogni prev scelgo il next più frequente, converto in stringa
transition_map = {
    prev: label_map[ctr.most_common(1)[0][0]]
    for prev, ctr in transition_counts.items()
}

# 2) fallback: il tipo di manutenzione più frequente in assoluto
overall = Counter(df_manutenzione['MaintenanceType'])
fallback = label_map[ overall.most_common(1)[0][0] ]

```

```

last_known_per_cycle: Dict[int,str] = {}
for i, rec in enumerate(dynamic_predictions):
    cycle = rec['cycle']

    if rec['event_type'] == 'PRODUZIONE':
        last_known_per_cycle[cycle] = rec['next_event']

    else: # MANUTENZIONE
        # cerco il primo passo produttivo successivo
        next_prod = next(
            (nxt for nxt in dynamic_predictions[i+1:])
            if nxt['cycle']==cycle and nxt['event_type']=='PRODUZIONE'),
            None
        )
        if next_prod:
            rec['next_event'] = next_prod['next_event']
        else:
            # non c'è produzione successiva: uso la transizione più probabile
            prev = last_known_per_cycle.get(cycle)
            if prev in transition_map:
                rec['next_event'] = transition_map[prev]
            else:
                rec['next_event'] = fallback

```

Questo blocco di codice, situato nella pipeline del main core, esegue, sostanzialmente, un controllo di coerenza sugli esiti del ramo regressivo e classificativo **per ogni sottociclo** dell'ultimo ciclo: per ciascun sottociclo estrae, dalle predizioni dinamiche, la sezione corrispondente alla produzione e confronta la sequenza decrescente $L, L-1, \dots, 1$ con i valori stimati di **remaining_rett**, accumulando questi dati per tutti i sottocicli disponibili; contemporaneamente, per l'ultimo sottociclo, calcola l'evento di manutenzione successivo confrontando la probabilità media predetta con la soglia ottimale, in modo da ottenere sia la lista delle etichette reali sia quella di quelle previste. Quindi, costruisce una mappa di transizione basata sullo storico degli interventi manutentivi, contando quante volte ogni tipo di manutenzione è seguito da un altro e scegliendo per ciascuno il “next_event” più frequente, con un fallback pari all'intervento storico più comune in assoluto. Infine, scorrendo nuovamente tutte le predizioni, aggiorna gli eventi di manutenzione ereditando il **next_event** dal primo record di produzione successivo allo stesso ciclo – se presente – oppure, in mancanza di una produzione successiva, applica la mappa di transizione o il fallback globale, garantendo così che ogni manutenzione venga etichettata coerentemente, tanto con i dati effettivi quanto con le tendenze storiche.

4. Esperimenti e risultati

4.1 Risultati e prestazioni ottenute

Una volta conclusa tutta la parte relativa all'addestramento del modello di base, ho processato varie stampe per determinare, a terminale video, i risultati delle predizioni ottenute e le prestazioni finali della rete neurale del core. Ho deciso di addestrare il modello LSTM Bidirezionale su un dataset iniziale di circa 5000 record manutentivi totali, opportunamente suddivisi in cicli di N uguale a 20 - 35 operazioni di manutenzione totali – composte da N-1 Rettifiche e 1 Sostituzione (quindi, ad esempio, in un ciclo di 20 operazioni: 19 rettifiche e 1 sostituzione finale) - a loro volta, separate tra loro da blocchi di 450 – 550 operazioni di produzione l'uno. Questo approccio ha permesso una granularità massima per quanto riguarda la variabilità nei dati. Infatti, i cicli potevano essere di dimensione differente tra loro, con blocchi di produzione di lunghezza diversa l'uno dall'altro ecc. non rendendo **mai** costante la distribuzione delle sequenze temporali della mappatura del piano martire. Pertanto, il dataset di input risultante è stato composto da circa 5000 eventi manutentivi e 2500000 eventi produttivi (situazione improbabile nella realtà ma utile per mettere alla prova la robustezza del modello. In un contesto produttivo reale, in un anno di lavoro probabilmente non ci si avvicinerebbe ai 1000 eventi manutentivi totali). Siccome il dataset generato era enorme, ho aumentato la dimensione del mini-batch dei vari set da 32 (standard) a 64 e, in alcuni tentativi, anche a 128, ma le performance migliori le ho ottenute con la via di mezzo a 64. Nonostante la dimensione maggiore dei batch, sono serviti circa 247 batch totali da processare ad ogni epoca, con un tempo di esecuzione, tramite CPU e non GPU, di 150 secondi medi per ognuno di essi. L'esecuzione, da cui ho ottenuto performance migliori, è durata circa **otto** epoche totali con iterazioni molto lunghe causa la grande mole di dati. Ognuna di queste otto epoche, ironia della sorte, ha occupato circa otto ore di processing continuo; quindi, un addestramento intero del modello su un dataset del genere e con il solo uso di CPU (a causa della mancanza di GPU sul computer control panel delle macchine CNC) ci ha messo in media 48 ore di tempo. Tempistiche molto lunghe ma plausibili data la moltitudine di dati generati e, compromesso accettabile, dato che l'addestramento enorme iniziale sarebbe stato fatto solo la prima volta, per poi reindirizzare il modello pre-addestrato ad ogni cliente distinto tramite l'operazione di fine-tuning. Questo aspetto mi ha permesso anche di ridimensionare il numero di epoche plausibili per l'addestramento effettivo: ero partito da 100 epoche totali, poi mi sono accorto che, effettivamente, erano superflue, poiché la rete raggiungeva già un discreto risultato dopo una decina di esse. Per cui, ho deciso di diminuire il numero massimo di epochs a 50 e sperimentare vari valori col learning rate e con il batch size per sfruttarle al meglio.

A livello di metriche di validazione della rete - sul validation set dell'epoca n. 8 su cui è stato innescato EarlyStopping e, di conseguenza, ripristinati i pesi dell'epoca n. 6 poiché considerata la migliore – per mezzo dell'utilizzo delle librerie di Keras e Tensorflow, ho ottenuto i seguenti valori a terminale:

Epoch 8 ROC-AUC: 0.8124

```
247/247 ━━━━━━━━━━━━━━━━ 177s 5s/step - loss: 1.4285 -  
next_maint_type_accuracy: 0.9065 - next_maint_type_auc: 0.8691 - next_maint_type_f1_score_metric:  
0.9353 - next_maint_type_loss: 0.0718 - next_maint_type_precision: 0.9090 - next_maint_type_recall: 0.9633  
- remaining_rett_loss: 0.0192 - remaining_rett_mae: 0.1555 - remaining_rett_mse: 0.0389 -  
remaining_sost_loss: 0.0447 - remaining_sost_mae: 0.2508 - remaining_sost_mse: 0.1004 -
```

Validation set:

```
val_loss: 1.2939 - val_next_maint_type_accuracy: 0.9700 - val_next_maint_type_auc: 0.8051 -  
val_next_maint_type_f1_score_metric: 0.9880 - val_next_maint_type_loss: 0.0623 -  
val_next_maint_type_precision: 0.9700 - val_next_maint_type_recall: 1.0000 - val_remaining_rett_loss: 0.0101  
- val_remaining_rett_mae: 0.1081 - val_remaining_rett_mse: 0.0240 - val_remaining_sost_loss: 0.0455 -  
val_remaining_sost_mae: 0.2720 - val_remaining_sost_mse: 0.1251 - learning_rate: 0.0010
```

Epoch 8: early stopping - Restoring model weights from the end of the best epoch: 6.

```
1/4 ━━━━━━━━━━━━━━━━ 11s 4s/step - loss: 1.9204 - next_maint_type_accuracy:  
1.0000 - next_maint_type_auc: 0.8255208134651184 - next_maint_type_f1_score_metric: 1.0000 -  
next_maint_type_loss: 0.0337 - next_maint_type_precision: 1.0000 - next_maint_type_recall: 1.0000 -  
remaining_rett_loss: 0.0322 - remaining_rett_mae: 0.2274 - remaining_rett_mse: 0.0737
```

```
2/4 ━━━━━━━━━━━━━━━━ 2s 1s/step - loss: 1.9238 - next_maint_type_accuracy:  
1.0000 - next_maint_type_auc: 0.8323785628368 - next_maint_type_f1_score_metric: 1.0000 -  
next_maint_type_loss: 0.0336 - next_maint_type_precision: 1.0000 - next_maint_type_recall: 1.0000 -  
remaining_rett_loss: 0.0323 - remaining_rett_mae: 0.2278 - remaining_rett_mse: 0.0739 - remainin
```

```
3/4 ━━━━━━━━━━━━━━━━ 1s 1s/step - loss: 1.9268 - next_maint_type_accuracy:  
0.9896 - next_maint_type_auc: 0.8406 - next_maint_type_f1_score_metric: 0.9945 - next_maint_type_loss:  
0.0343 - next_maint_type_precision: 0.9896 - next_maint_type_recall: 1.0000 - remaining_rett_loss: 0.0323 -  
remaining_rett_mae: 0.2278 - remaining_rett_mse: 0.0739
```

```
4/4 ━━━━━━━━━━━━━━━━ 6s 833ms/step - loss: 1.9244 - next_maint_type_accuracy:  
0.9778 - next_maint_type_auc: 0.8668 - next_maint_type_f1_score_metric: 0.9775 - next_maint_type_loss:  
0.0364 - next_maint_type_precision: 0.9778 - next_maint_type_recall: 1.0000 - remaining_rett_loss: 0.0323 -  
remaining_rett_mae: 0.2279 - remaining_rett_mse: 0.0739 - remaining_sost_loss: 0.0553 -  
remaining_sost_mae: 0.3264 - remaining_sost_mse: 0.1670
```

Come già affermato precedentemente, l'utilizzo della metrica Accuracy è stato fuorviante in questo tipo di problema, ma ho voluto comunque includerla per avere un quadro completo e generale su tutte le metriche comunemente utilizzate e disponibili. Mentre, a tempo di test, ho ottenuto invece questi risultati:

Test-set performance: {

```
'loss': 1.9332501888275146, 'next_maint_type_accuracy': 0.9599999785423279,  
'next_maint_type_auc': 0.8255208134651184, 'next_maint_type_f1_score_metric': 0.9519906044006348,  
'next_maint_type_loss': 0.039402469992637634, 'next_maint_type_precision': 0.9599999785423279,  
'next_maint_type_recall': 1.0, 'remaining_rett_loss': 0.03239322453737259,  
'remaining_rett_mae': 0.22794415056705475, 'remaining_rett_mse': 0.07394938170909882,  
'remaining_sost_loss': 0.05321785435080528, 'remaining_sost_mae': 0.22945725321769714,  
'remaining_sost_mse': 0.16514477133750916 }
```

1/1 ————— 1s 1s/step
1/1 ————— 1s 997ms/step
1/1 ————— 1s 978ms/step
1/1 ————— 0s 247ms/step

→ Soglia ottimale (Youden J) = 0.674 (TPR=0.247, 1-FPR=1.000)

Best_threshold = 0.674

Sensitivity(Recall) = 1.000

Specificity = 1.000

Complessivamente, considerando che ho creato i dati da zero – in maniera fittizia - poiché non erano disponibili a priori e che il test set è originato da una porzione di dati mai vista prima dal modello durante l’addestramento, i risultati delle performance ottenuti sono soddisfacenti. Più nello specifico, a livello di test set si è ottenuto:

'next_maint_type_auc': 0.8255208134651184

'remaining_rett_mae': 0.22794415056705475

'remaining_sost_mse': 0.16514477133750916

Questo significa che, il modello, distingue esattamente le due classi tra loro con un'**accuratezza AUC dell'82%**. Per una visione d’insieme più dettagliata, di seguito sono presenti il classification report e la matrice di confusione generati:

```

Confusion Matrix:
[[ 3  0]
 [9 88]]

Classification Report:
              precision    recall   f1-score   support
0.0          1.00     1.00     1.00      3
1.0          0.94     1.00     0.97     97

accuracy      0.97     1.00     0.99     100
macro avg     0.97     1.00     0.99     100
weighted avg  0.94     1.00     0.97     100

```

I valori della matrice di confusione indicano che il modello ha riconosciuto la classe 0 per tre volte e tutte e tre quelle volte è stata classificata come **vero negativo**. Supportato dal fatto che la recall e la precision sono molto elevate, si può affermare che il pre-processing sullo sbilanciamento ha funzionato alla perfezione e il modello è stato in grado di riconoscere completamente la classe minoritaria. D'altro canto, però probabilmente, aver forzato il modello stesso a riconoscere meglio la classe minoritaria, ha fatto in modo che esso fosse leggermente più impreciso sulla classe maggioritaria. Infatti, su 97 esempi di classe maggioritaria 88 sono stati riconosciuti come **veri positivi** (risultato comunque notevole) e 9 esempi sono stati riconosciuti come **falsi positivi**. Questo può essere dovuto, molto probabilmente, al fatto che la precision sulla classe 1 è 0.94 e, di conseguenza, il margine di errore – anche se lieve – esiste.

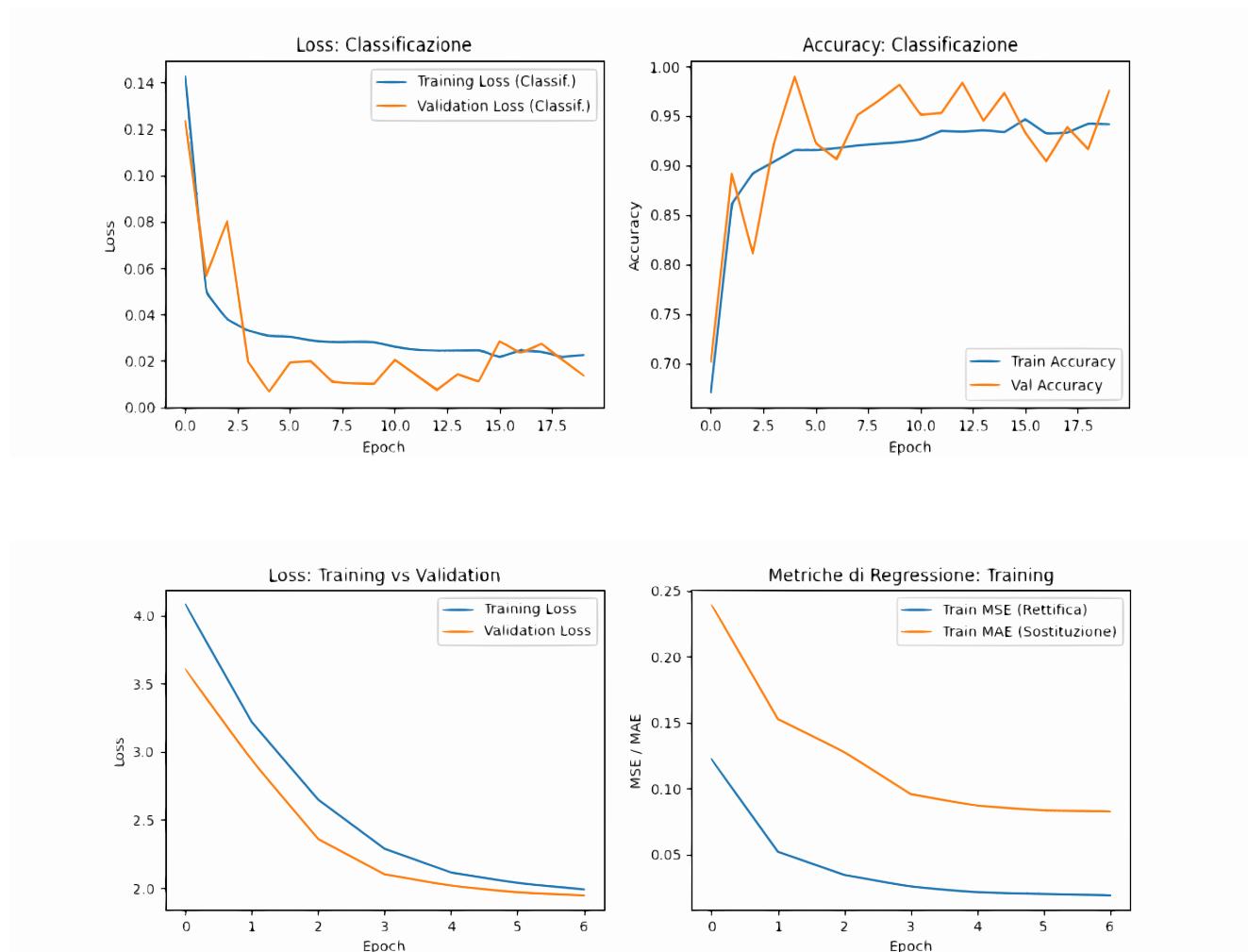
Come step successivo, regressivamente parlando, ho riscontrato un **MAE = 0.22** su Remaining_rett e un MSE di 0.16 su Remaining_sost, da cui **RMSE = 0.406**. Questo significa che, il modello, in un sottociclo di rettifica, emette predizioni, che deviano, in media, dal valore reale di circa 22 unità, quindi, nel mio caso, di circa 22 lavorazioni totali su sottocicli, ma considerando che essi dispongono di un numero di lavorazioni totali dell'ordine del migliaio... quindi, risultato molto buono. Mentre, in un ciclo di vita intero da sostituzione pregressa a sostituzione prossima, il modello ottiene un RMSE = 0.406 su Remaining_sost, il che significa che circa 40 lavorazioni, in media, lungo il corso del ciclo di vita, vengono predette in maniera incorretta... nonostante ciò, non è un numero scomodo.

Da tutto ciò si può concludere che, nella classificazione, il modello è in grado di distinguere esattamente le due classi, salvo per particolari situazioni più complesse (scarsità dei dati, cicli e sottocicli troppo variabili ecc.) mentre, nella regressione, il modello lavora straordinariamente bene su errori grandi, ma tende ad essere leggermente più impreciso su errori più piccoli; probabilmente, dovuto al fatto che i due branch (classificativo e regressivo) condividono l'intero backbone della rete e questo elargisce tanti vantaggi (features condivise, addestramento simultaneo ecc.), ma, contemporaneamente, può originare anche alcune problematiche legate all'impossibilità, da parte loro, di specializzarsi al massimo nel proprio compito distinto. Inoltre, allo stesso tempo, una rete LSTM Bidirezionale potrebbe essere meno performante di un Transformer per questo tipo di task... nonostante ciò, il ramo regressivo ha funzionato in maniera corretta e i risultati ottenuti sono stati comunque soddisfacenti.

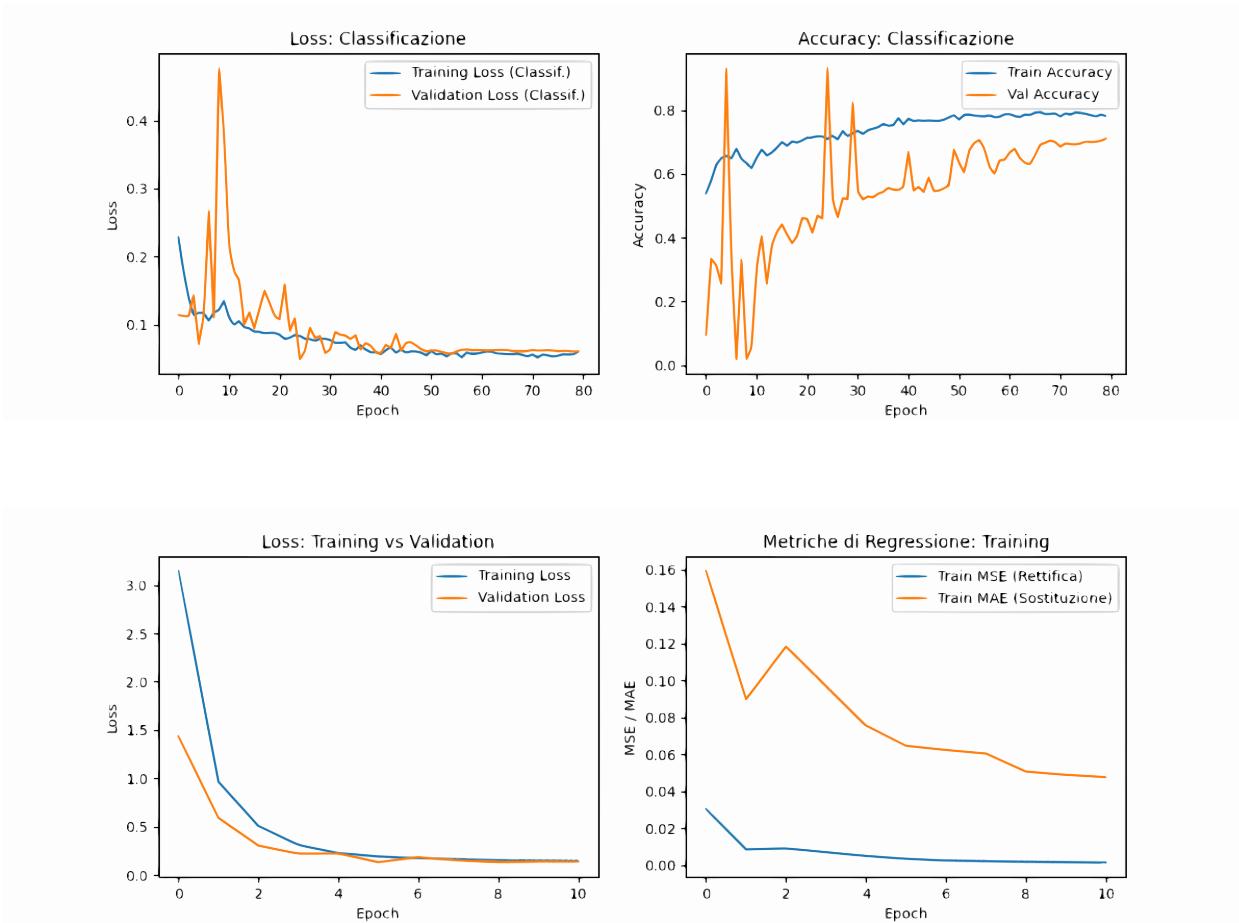
4.2 Grafici delle metriche e risultati di inferenza

Di seguito, si mostreranno i relativi grafici associati – generati appositamente tramite le funzionalità delle librerie di Tensorflow - delle varie metriche utilizzate e dei risultati delle performance ottenuti in fase di test. I grafici verranno inseriti in ordine cronologico di esecuzione, delineando tutti i vari tentativi di addestramento effettuati, fino ad arrivare alla fase finale che è stata, per l'appunto, quella migliore riscontrata. Tutte le immagini fornite di seguito sono in formato vettoriale .emf e sono state generate da me tramite la libreria Tensorflow di Python.

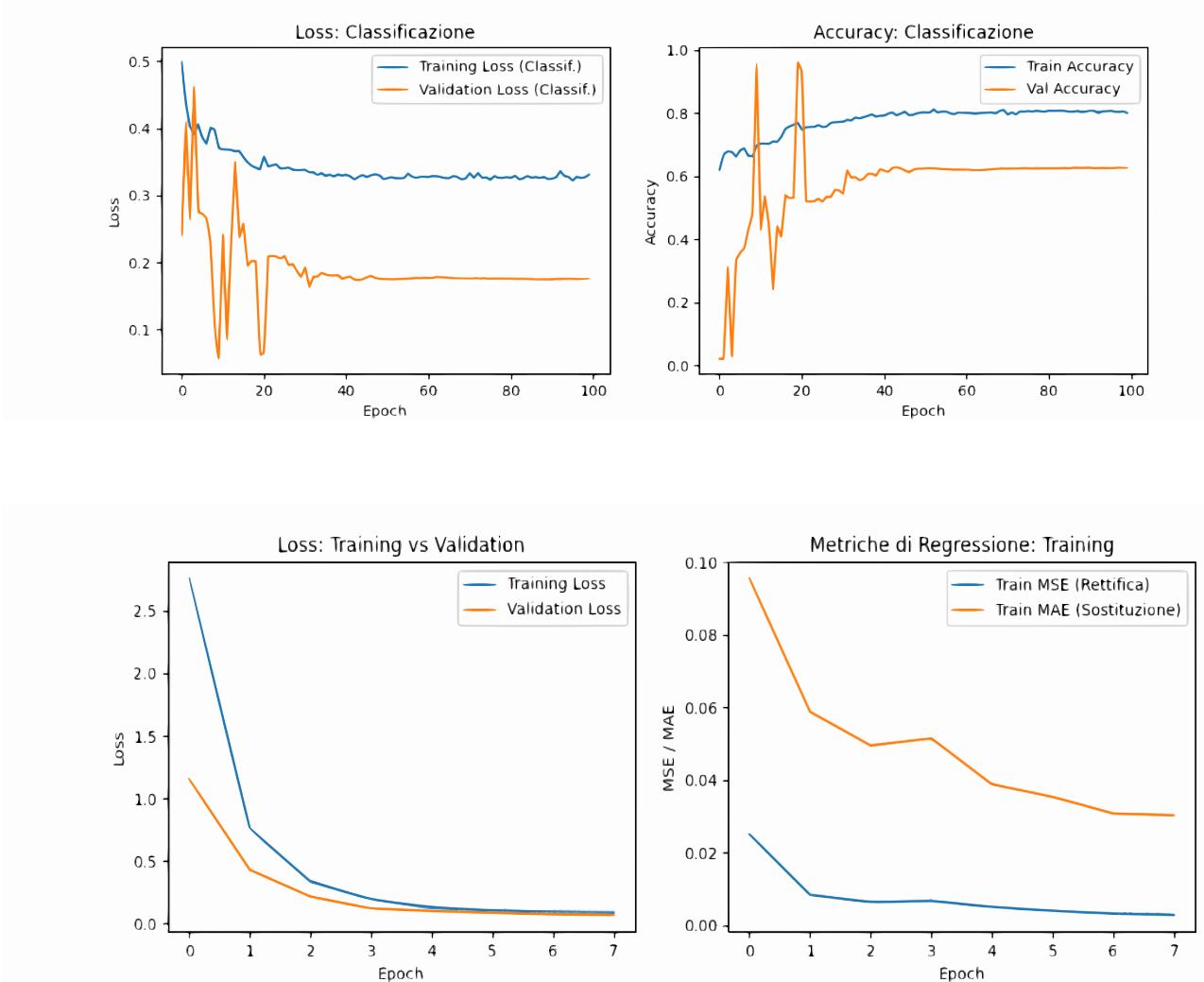
Innanzitutto, i grafici riguardanti la loss e le metriche di regressione (MAE e MSE) sono stati i seguenti inizialmente:



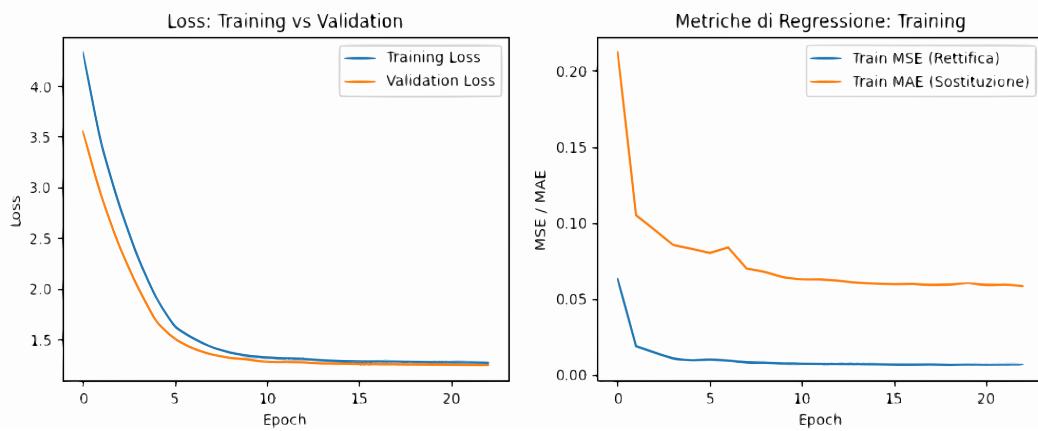
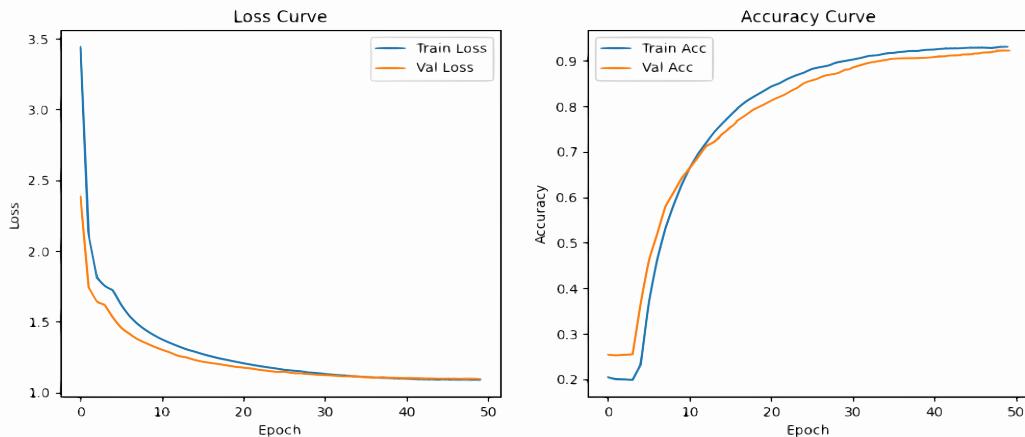
Come si può ben notare, inizialmente la loss del classificatore era molto rumorosa, con un continuo sali e scendi, specialmente in validazione. Mentre, per quanto riguarda la regressione, i valori erano decisamente più stabili nella loss generale e più oscillanti in procinto delle metriche di regressione classiche, specialmente sulla Sostituzione, dove stavo ancora testando MAE. Con ulteriori aggiustamenti e tentativi si è arrivati a questo risultato:



Monitorando i grafici finali dell'addestramento, alzando il numero di epoche e aggiustando un po' di iperparametri e di layers nel modello, le prestazioni erano leggermente migliorate. Tuttavia, il classificatore continuava a oscillare considerevolmente, anche se lo faceva in modo proporzionale e coerente. Si poteva tranquillamente fare di meglio. Ho provato, quindi, ad aumentare il numero di epoche a 100 totali, aggiungere un layer Conv1D per ogni branch regressivo – per un totale di due layer convoluzionali – e raddoppiare i neuroni su ciascun layer LSTM Bidirezionale, con la dovuta regolarizzazione, ottenendo questo:

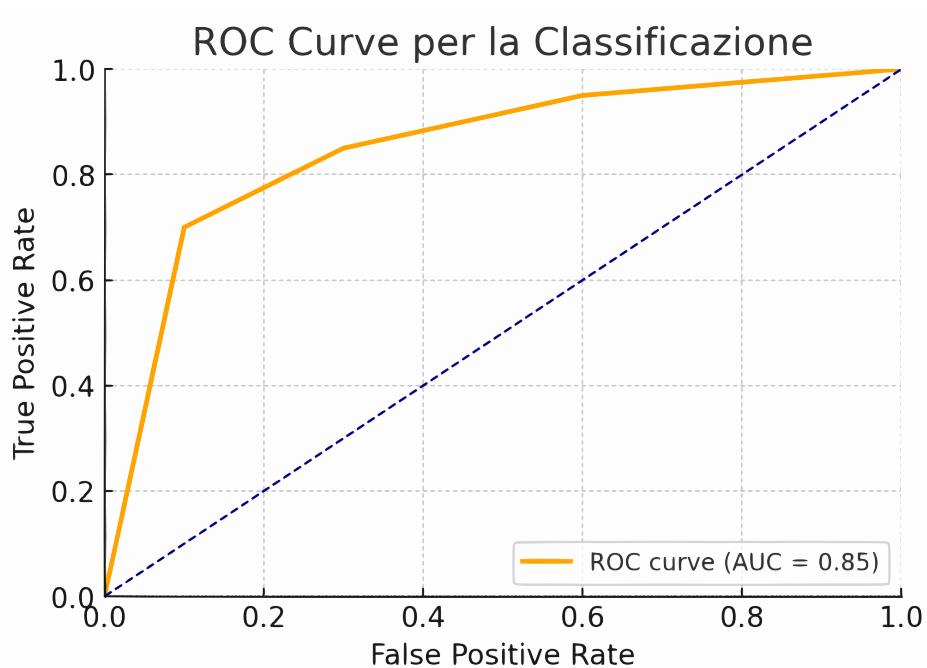


Il miglioramento era evidente, ma non abbastanza. Per cui, era necessario effettuare varie modifiche all'architettura del modello, ai dataset on-the-fly ed altre strutture del pre-processing, per ottenere risultati ancora più soddisfacenti. Dopo svariati tentativi, cambiamenti, fallback e frustrazioni, ho trovato la configurazione ottimale che ho mostrato nei capitoli precedenti, con un numero di epoche di iterazione pari a 50:



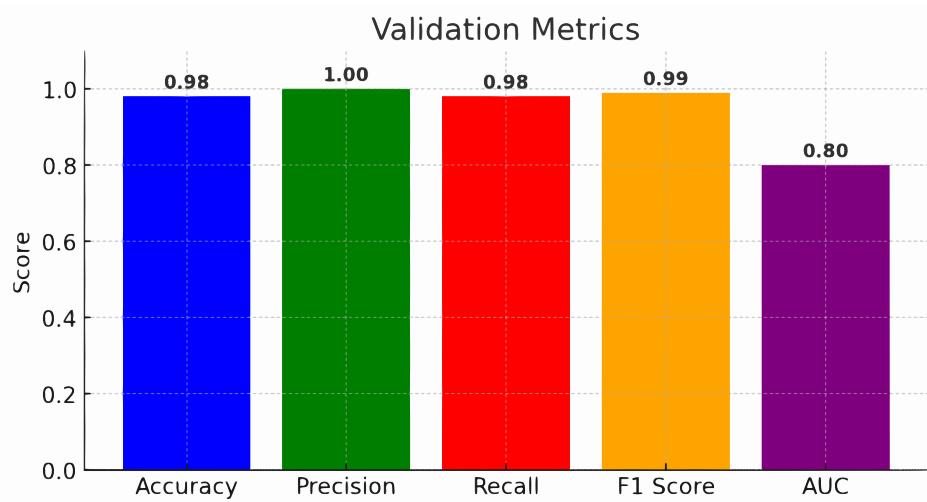
Queste sono le loss e le metriche regressive migliori che ho ottenuto a livello grafico. Qui ho scelto di plottare MSE per la Rettifica e MAE per la Sostituzione per verificare meglio le inaccuratezze (poiché MAE per la Rettifica e MSE per la Sostituzione funzionavano, numericamente parlando, già in modo ottimale), ma era chiaramente possibile tracciare anche i grafici delle metriche dominanti; tuttavia, per brevità, sono stati omessi.

Proseguendo, ho plottato la curva ROC-AUC (più rilevante dell'Accuracy in un problema sbilanciato) definitiva che il modello ha delineato:

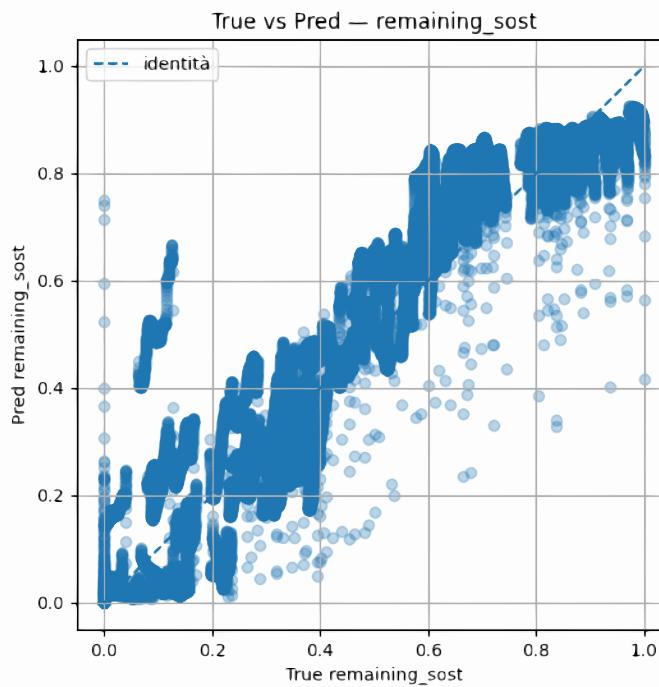
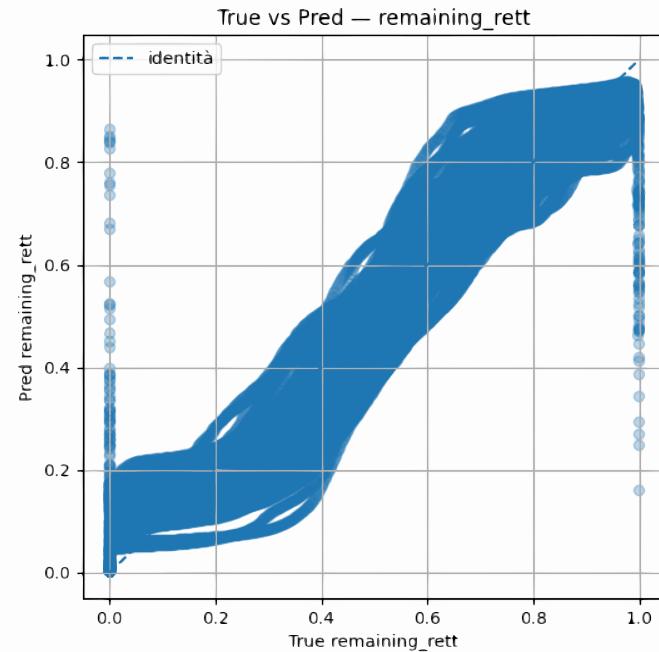


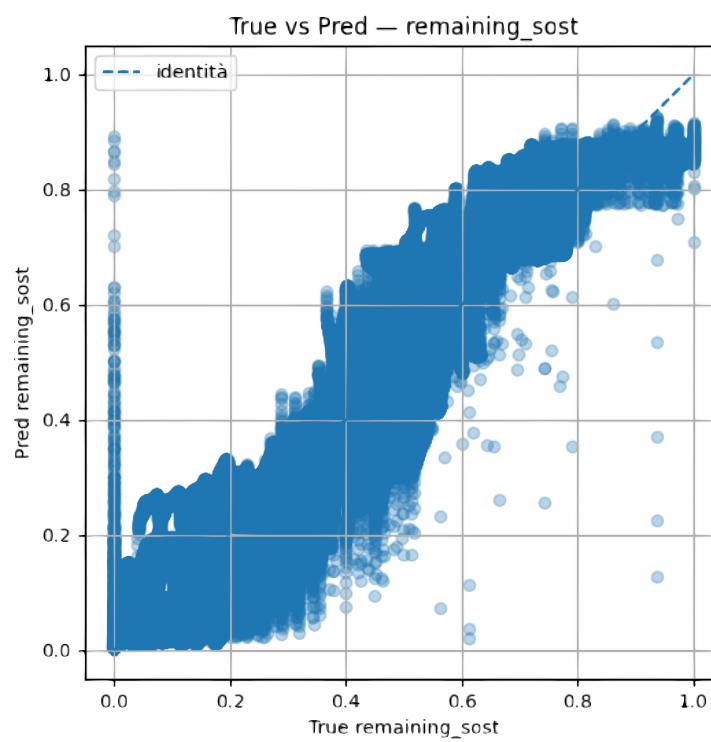
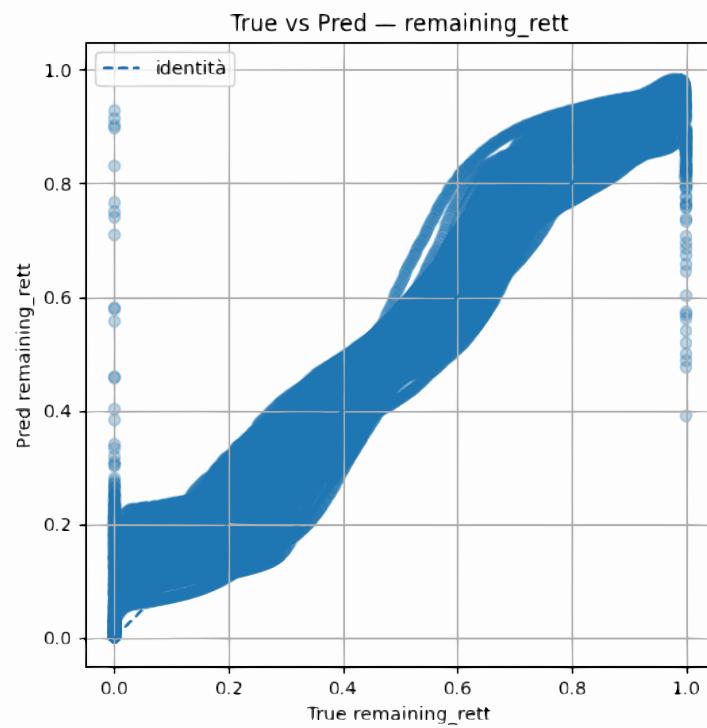
Mi sono ritenuto soddisfatto dell'AUC raggiunta con questo tipo di architettura e con tutti i tentativi effettuati. Probabilmente, per spingere ulteriormente i margini di miglioramento, sarebbe stata necessaria l'implementazione di un'architettura, basata sul MultiHeadAttention, come un Transformer o l'introduzione di nuove features rilevanti (anche se, attualmente per questo tipo di problema, esse non sono formulabili con i dati attuali) o l'utilizzo di altre tecniche di pre-processing più avanzate, che avrebbero consentito di far recepire un segnale più forte e pulito alla rete neurale.

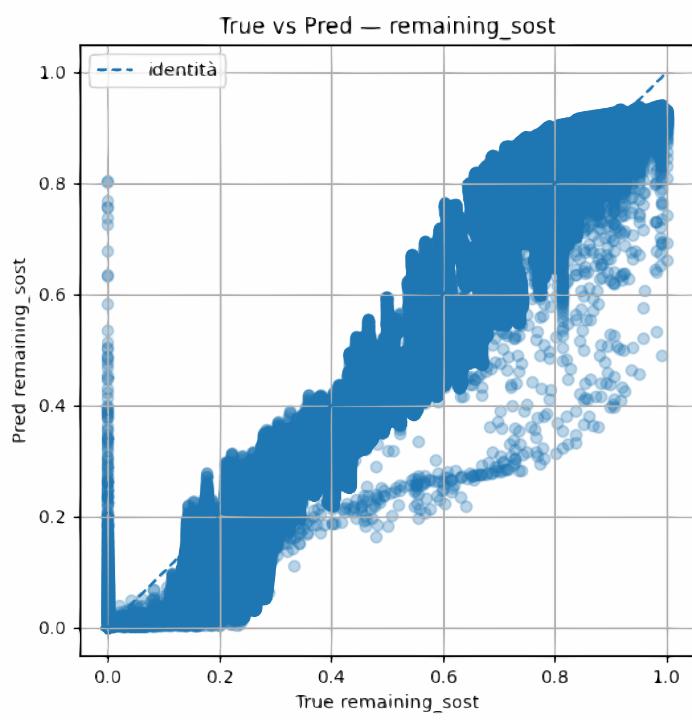
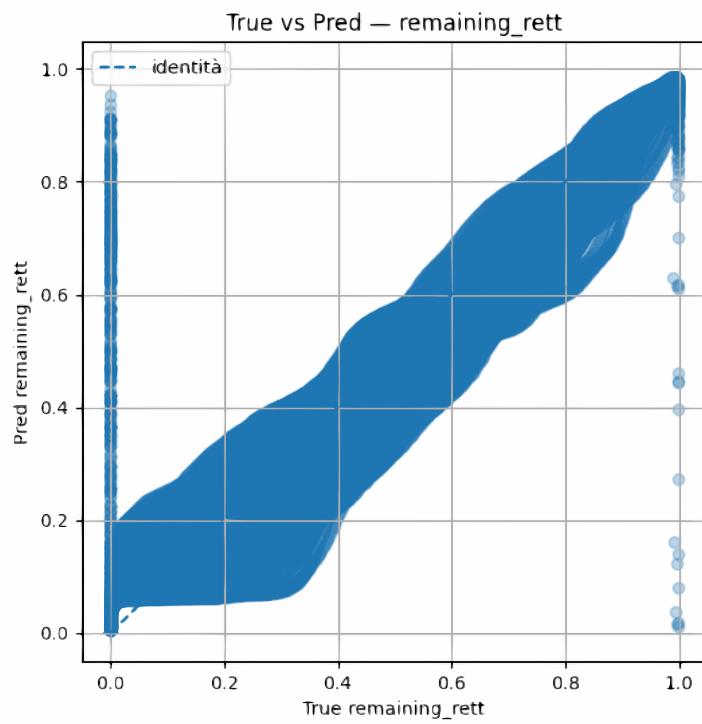
Per riassumere sommariamente, le metriche di validazione ottenute e spiegate precedentemente, le ho riportate in questo schema grafico generato da Keras a fine addestramento:

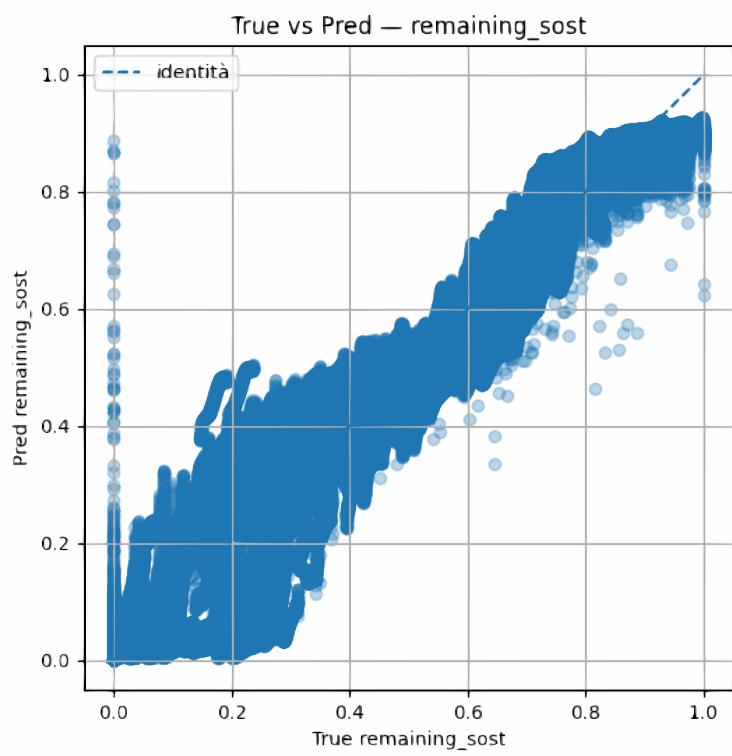
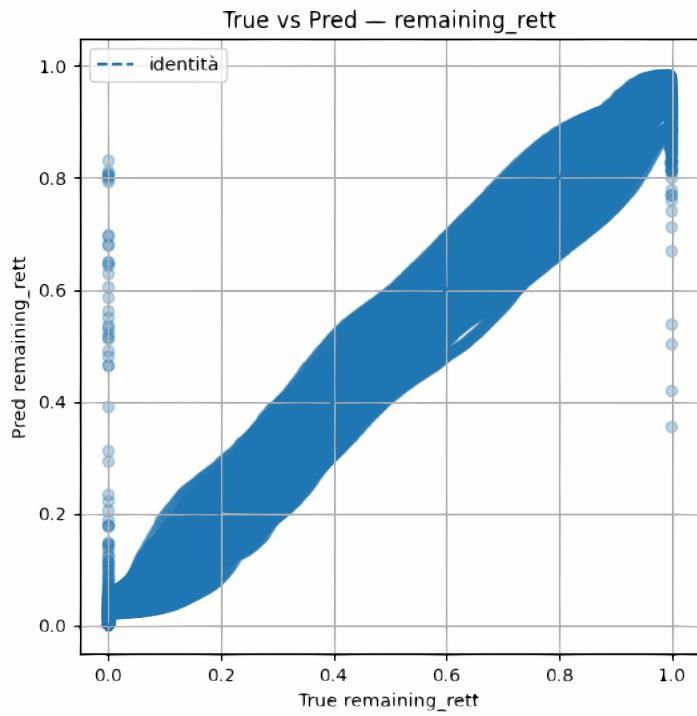


Per concludere la parte grafica e visiva, voglio, inoltre, mostrare dei plot che tracciano la distribuzione cronologica tra valore atteso e valore predetto, relativo alle uscite regressive: Remaining_rett e Remaining_sost.









Per dare un'idea di come funziona il modello nella fase di inferenza dinamica finale, espongo di seguito un piccolo snippet di output, estratto da terminale, che mostra i risultati delle predizioni sull'ultimo ciclo di vita, di un dataset iniziale, non completo (quindi senza sostituzione finale):

```
→ Ciclo 34, Sottociclo 25:
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12246, 'remaining_rett': 497, 'remaining_sost': 1401, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12247, 'remaining_rett': 496, 'remaining_sost': 1400, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12248, 'remaining_rett': 495, 'remaining_sost': 1399, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12249, 'remaining_rett': 494, 'remaining_sost': 1398, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12250, 'remaining_rett': 493, 'remaining_sost': 1397, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12251, 'remaining_rett': 492, 'remaining_sost': 1396, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12252, 'remaining_rett': 491, 'remaining_sost': 1395, 'predicted_days_rett': 3, 'predicted_days_sost': 8, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12253, 'remaining_rett': 490, 'remaining_sost': 1394, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12254, 'remaining_rett': 489, 'remaining_sost': 1393, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12255, 'remaining_rett': 488, 'remaining_sost': 1392, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12256, 'remaining_rett': 487, 'remaining_sost': 1391, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12257, 'remaining_rett': 486, 'remaining_sost': 1390, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12258, 'remaining_rett': 485, 'remaining_sost': 1389, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12259, 'remaining_rett': 484, 'remaining_sost': 1388, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12260, 'remaining_rett': 483, 'remaining_sost': 1387, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12261, 'remaining_rett': 482, 'remaining_sost': 1386, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12262, 'remaining_rett': 481, 'remaining_sost': 1385, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12263, 'remaining_rett': 480, 'remaining_sost': 1384, 'predicted_days_rett': 3, 'predicted_days_sost': 7, 'class_probability': 0.5628395676612854, 'event_type': 'PRODUZIONE', 'next_event': 'RETTIFICA'}
  {'cycle': 34, 'subcycle': 25, 'seq_id': 26, 'time_step': 12264, 'remaining_rett': 479, 'remaining_sost': 1383, 'predicted_days_rett': 3,
```

Da questo frammento di output, si può identificare che, gli eventi produttivi rimanenti rispettivi di Remaining_rett e Remaining_sost, diminuiscono regressivamente al processarsi dei vari timestep e, siccome il ciclo in questione non è ancora stato completato, il prossimo evento manutentivo predetto "next_event" è, di fatto, RETTIFICA. Questo si verifica proprio perché la rete ha imparato che, da questo particolare dataset, il cliente esegue l'operazione di SOSTITUZIONE solo quando il piano martire è completamente usurato (quindi quando Remaining_rett e Remaining_sost sono, pressoché identiche); per cui, tale piano viene sfruttato fino all'ultimo con operazioni di RETTIFICA e, se il valore di Remaining_sost > Remaining_rett come si evince da qui, allora è molto probabile che, il prossimo evento manutentivo da predire, sarà ancora un'ennesima RETTIFICA. I giorni predetti, invece, sono stati determinati utilizzando i valori regressivi predetti e i valori medi orari per evento di produzione che ho spiegato inizialmente.

5. Fine-tuning e Transfer Learning

5.1 Fine-tuning “regression-only” vs “full”

Siamo giunti ora al secondo macro-argomento di questa tesi di laurea magistrale. Quello in cui, si salvano i pesi del modello migliore trovato e si importano nel processo di raffinamento della conoscenza: il **fine-tuning**. Essa è una procedura mediante la quale, un modello di intelligenza artificiale già pre-allenato su un ampio set di dati generalisti, viene ri-adattato ad un compito specifico, sfruttando sia le conoscenze già acquisite durante l’addestramento profondo iniziale, sia una porzione di dati più mirata e di dimensioni ridotte. In pratica, a partire da una rete neurale con pesi già modellati su un dominio ampio di partenza, si introduce un nuovo dataset – spesso di minor dimensione rispetto a quello originario - etichettato per il problema concreto emergente (ad esempio classificazione o regressione real-time come nel mio caso) e si riprende l’addestramento da dove si era rimasti. Il fine-tuning si divide principalmente in tre fasi distinte:

- **Caricamento del modello pre-addestrato**, comprensivo della sua architettura e dei pesi già ottimizzati su dati generici iniziali.
- **Definizione e configurazione del nuovo task**, definendo, solitamente, il nuovo layer di output e le funzioni di perdita/metriche appropriate per il nuovo compito da trattare. Nel mio caso specifico, il nuovo compito è stato un adattamento – in tempo reale – del compito precedente, ma su dati che arrivavano in maniera incrementale, presenti in numero ridotto e diversi per ogni cliente distinto.
- **Addestramento mirato**, in cui si congela - tramite Transfer Learning – solo una parte dei layer originali - spesso i primi – e si riaddestrano tutti i livelli lasciati sbloccati, oppure si inseriscono moduli di adapter layer (come ho fatto io nel modello del core) che apprendono esclusivamente piccole correzioni, mantenendo il backbone solido iniziale. Durante questa fase si utilizza un learning rate ridotto, rispetto all’originale, per evitare di degradare le conoscenze pre-apprese e scivolare nel “catastrophing forgetting” sui nuovi dati.

Nel mio caso specifico, ho dovuto adottare un approccio di fine-tuning **ibrido, modulare e duale**, di modo che si adattasse, in maniera sensata, al modello multi-output che avevo utilizzato nel core principale del codice. Il motivo principale era il fatto che, il modello stesso, aveva compiti logici distinti (classificazione e regressione) ma venivano tutti processati allo stesso momento da un’architettura ibrida. Questo aspetto ha aumentato notevolmente la complessità della procedura poiché, le tempistiche e le esigenze di un fine-tuning della parte regressiva erano differenti da quelle che, invece, riguardavano la parte classificativa. Infatti, ad esempio, era possibile ri-addestrare il classificatore solo quando entrambe le classi fossero state disponibili nel dataset (poichè, per addestrare un modello classificatore, tutte le classi inerenti devono essere presenti) e, pertanto, bisognava per forza aver affrontato almeno un evento di Sostituzione, quindi aver terminato con successo un ciclo completo. Mentre, per ri-addestrare i due branch regressivi e mantenere un certo grado di conoscenza già acquisita, era sufficiente aver a disposizione almeno un sottociclo interno e, quindi, un’operazione di Rettifica avvenuta con successo.

Queste discrepanze, tra obiettivi funzionali diversi, hanno fatto in modo che, da un semplice criterio di adattamento ai nuovi dati, si sia invece originato, un vero e proprio “core secondario” a sé stante. Proprio per tale motivo, è stato ampiamente necessario - a livello di codice - effettuare la distinzione tra i due approcci separati, per gestire correttamente il fine-tuning in fase di deployment.

1. *Fine-tuning: Regression-only*

È la sezione di fine-tuning real-time che agisce solamente sui rami regressivi e ri-addestra, costantemente, le testine dei layer Dense regressivi di output. Ho programmato il codice in modo che, ogni volta che un nuovo evento di manutenzione fosse stato registrato nel nuovo dataset del cliente specifico (questa parte verrà poi approfondita meglio nel terzo e ultimo macro-argomento della tesi), sarebbe stato innescato il trigger di avvio, del fine-tuning regression-only, su tutto il dataset accumulato fino a quel preciso evento di manutenzione appena raggiunto.

La caratteristica principale della funzione di fine-tuning regression only è il fatto di agire direttamente su ogni singolo batch, con la tecnica chiamata “train-on-batch streaming”. Questo consente di applicare il raffinamento, sulle uscite regressive, in maniera incrementale, a mano a mano che i dati nuovi sono disponibili. Di seguito, il codice dell’intera funzione generata:

```
def fine_tune_regression_only(
    json_path: str,
    batch_size: int    = BATCH_SIZE,
    base_weights: str = BASE_WEIGHTS_PATH,
    lr: float         = LR,
    strategy: str    = STRATEGY
):
    global base_lr
    global _last_compile_mode
    _last_compile_mode = None
    # -- 0) Model + weights
    model = core.modelling_combined(num_features=len(core.features),
                                      units=128)
    w = FINE_TUNED_WEIGHTS_OUT if Path(FINE_TUNED_WEIGHTS_OUT).exists() \
        else BASE_WEIGHTS_PATH
    model.load_weights(w)
    base_lr = lr * (0.1 if w == FINE_TUNED_WEIGHTS_OUT else 1.0)
    ensure_compile(model, base_lr, mode="reg")
    model.optimizer.learning_rate.assign(base_lr)

    #threshold = load_threshold()
    prev      = load_expected()
    init_cycle = prev["sum_cycle_totals"]/prev["count_cycle_totals"] \
        if prev["count_cycle_totals"]>0 else 0.0
```

```

init_sub    = prev["sum_subcycle_totals"]/prev["count_subcycle_totals"] \ 
              if prev["count_subcycle_totals"]>0 else 0.0

      # 1) Caricamento e “enrichment” dei JSON
      raw_events = json.load(open(json_path))  # lista di dict, come nel tuo
esempio

      # 1.a) mettiamo in ordine cronologico REGISTRO per REGISTRO
      raw_events = sorted(raw_events, key=lambda e: e["Timestamp"])

      # 1.b) costruiamo due liste di supporto:
      #       - prod_list = [(timestamp_produzione, durata_ore), ...]
      #       - manut_list = [timestamp_manutenzione, ...]
      prod_list = []
      manut_list = []

      # => prima, estraiamo da raw_events tutte le produzioni già presenti
      for rec in raw_events:
          if rec.get("EventType") == 1:
              # ricavo la durata in ore: qui End e Start sono ISO8601
              t0 = datetime.fromisoformat(rec["Start"])
              t1 = datetime.fromisoformat(rec["End"])
              durata_h = (t1 - t0).total_seconds() / 3600.0
              prod_list.append((datetime.fromisoformat(rec["Timestamp"]),
durata_h))
          else:
              # record di manutenzione già esistente (magari ha MaintenanceType e WearValue)
              manut_list.append(datetime.fromisoformat(rec["Timestamp"]))

      # 1.c) adesso navighiamo di nuovo i raw_events e, per ciascuna manutenzione,
      # assegnamemo “TotalEvents” e “WorkingHours” basandoci su prod_list e
manut_list
      enriched = []
      for rec in raw_events:
          if rec.get("EventType") == 0:
              ts_manut = datetime.fromisoformat(rec["Timestamp"])
              # trovo l’ultima manutenzione precedente (se esiste)
              prev_manut = None
              prevs = [m for m in manut_list if m < ts_manut]
              if prevs:
                  prev_manut = max(prevs)

              # filtro le produzioni in (prev_manut, ts_manut]

```

```

        if prev_manut is not None:
            prods_int = [(t,d) for (t,d) in prod_list if prev_manut < t <=
ts_manut]
        else:
            prods_int = [(t,d) for (t,d) in prod_list if t <= ts_manut]

        rec = rec.copy()
        rec["TotalEvents"] = len(prods_int)
        rec["WorkingHours"] = round(sum(d for (_,d) in prods_int), 2)

        # aggiungo questo ts_manut alla lista delle manutenzioni già viste,
        # così il prossimo ciclo lo utilizzerà come "precedente"
        manut_list.append(ts_manut)
    # se è produzione, non tocco nulla (lascia TotalEvents magari assente o
1)
    enriched.append(rec)
df = merge_data_list(enriched)
df = core.missing_values(df)
df = core.compute_counters(df)
df = core.encoding(df)
df = core.segment_cycles(df)
df = core.segment_subcycles(df)
df_pre = core.prepare_targets_combined(df)
df_pre["Timestamp"] = df_pre["Timestamp"].astype(np.int64)//10**9

# --- 2) Update expected values
exp_cycle, exp_sub = core.compute_expected_values(df, dedupe=False)

# --- 3) Compute avg_hours
m = df_pre[df_pre['EventType']==0]
default_rett = core.calcola_default_avg_hours(m, 1, "T_Rettifica",
"TotalEvents")
default_sost = core.calcola_default_avg_hours(m, 0, "T_Sostituzione",
"CumulativeEvents")
vr = m[(m['MaintenanceType']==1)&(m['T_Rettifica']>0)]
vs = m[(m['MaintenanceType']==0)&(m['T_Sostituzione']>0)]
avg_rett = (vr['T_Rettifica']/vr['TotalEvents']).mean() \
            if len(vr)>=2 else (default_rett or 0.0)
avg_sost = (vs['T_Sostituzione']/vs['CumulativeEvents']).mean() \
            if len(vs)>=2 else (default_sost or 0.0)
print(f"avg_hours_rett={avg_rett:.3f}, avg_hours_sost={avg_sost:.3f}")

core.scaler_numeric = load_scaler()

```

```

numeric_cols = [
    "CumulativeEvents", "SubCycle_Events", "TotalEvents",
    "WorkingHours", "WearValue",
    "Tot_Events_Rett", "Tot_Events_Sost",
    "T_Rettifica", "T_Sostituzione"
]
# normalizzazione applica lo scaler_numeric già caricato in core
core.scaler_numeric.feature_names_in_ = np.array(numeric_cols, dtype=object)
df_pre[numeric_cols] = core.scaler_numeric.transform(df_pre[numeric_cols])
# -- 3) Crea le sequenze
x_all, y_all, metas = core.create_representative_sequences(
    df_pre, core.features, core.target, factor=1.0)
x_all, y_all, metas = core.filter_valid(x_all, y_all, metas)

# Validation set on the fly
n_seq = len(x_all)

if n_seq < 2:
    # se ho un solo ciclo, lo metto direttamente in val (serve per evitare
    sequenze vuote)
    val_buffer = [(x_all[0], y_all[0])]
    train_buffer = []
else:
    # altrimenti prendo il 20% (o almeno 1) per val, il resto per train
    n_val = max(1, n_seq // 5) # 20% ma almeno 1
    buffer_all = list(zip(x_all, y_all))
    random.shuffle(buffer_all)

    val_buffer = buffer_all[:n_val]
    train_buffer = buffer_all[n_val:]

def slice_last_cls(x, y_dict, sw):
    y_cls = y_dict["next_maint_type"]
    # y_cls: (batch, seq_len, 1) oppure (batch,1)
    batch = tf.shape(y_cls)[0]
    # appiattisco le ultime due dimensioni:
    flat = tf.reshape(y_cls, (batch, -1)) # → (batch, seq_len) o (batch,1)
    last = flat[:, -1] # → (batch,)
    last = tf.expand_dims(last, axis=-1) # → (batch,1)
    return x, {
        "remaining_rett": y_dict["remaining_rett"],
        "remaining_sost": y_dict["remaining_sost"],
        "next_maint_type": last

```

```

}, sw

def build_val_ds(buffer, batch_size):
    Xv, Yv = core.pad_dynamic_sequences(
        [x for x,y in buffer],
        [y for x,y in buffer]
    )
    ds = tf.data.Dataset.from_tensor_slices((Xv, Yv))
    ds = ds.map(lambda x, y: core.custom_map_fn(x, y),
                num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(core.mask_padding,
                num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(slice_last_cls,
                num_parallel_calls=tf.data.AUTOTUNE)
    return ds.batch(batch_size).prefetch(tf.data.AUTOTUNE)

val_ds = build_val_ds(val_buffer, batch_size)

# 4) Data augmentation: duplichiamo il training set
x_aug, y_aug = [], []
idx_wh = core.features.index("WorkingHours")
idx_wv = core.features.index("WearValue")
idx_te = core.features.index("TotalEvents")
for x_seq, y_seq in zip(x_all, y_all):
    x_j = x_seq.copy()
    x_j[:, idx_wh] *= np.random.uniform(0.95,1.05,size=x_j.shape[0])
    x_j[:, idx_wv] *= np.random.uniform(0.9,1.1, size=x_j.shape[0])
    noise = np.random.randint(-2,3, size=x_j.shape[0])
    x_j[:, idx_te] = np.clip(x_j[:, idx_te] + noise, a_min=0, a_max=None)
    x_aug.append(x_j); y_aug.append(y_seq)
x_train = x_all + x_aug
y_train = y_all + y_aug

# per il DynamicAlphaCallback
real_train_ds = (
    tf.data.Dataset
        .from_tensor_slices(core.pad_dynamic_sequences(x_train, y_train))
        .map(lambda x,y: core.custom_map_fn(x,y), tf.data.AUTOTUNE)
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
)
# -- 4) Loop di training "regression only"
history = {

```

```

        "train_remaining_rett_mae": [], "train_remaining_rett_mse": [],
        "train_remaining_sost_mae": [], "train_remaining_sost_mse": []
    }
    # 8) Callbacks standard + ClassificationFlushCallback
    delta_cb      = core.DeltaUpdateCallback(val_ds, percentile=98, smoothing=0.2,
min_delta=0.15)
    dyn_alpha_cb = core.DynamicAlphaCallback(real_train_ds)
    reduce_lr_cb = tf.keras.callbacks.ReduceLROnPlateau(
                    monitor="val_remaining_rett_mae", mode="min",
                    factor=0.5, patience=2, min_lr=1e-6, verbose=1)
    early_stop_cb = tf.keras.callbacks.EarlyStopping(
                    monitor="val_remaining_rett_mae", mode="min",
                    patience=5, restore_best_weights=True)
    checkpoint_cb = ModelCheckpoint(
                    filepath=FINE_TUNED_WEIGHTS_OUT,
                    monitor="val_remaining_rett_mae",
                    mode="min", save_best_only=True, save_weights_only=True,
verbose=1)

    # assegno il model a TUTTI i callback
    callbacks = [delta_cb, reduce_lr_cb,
                 #es1_cb,
                 early_stop_cb, checkpoint_cb,
                 #dyn_alpha_cb
                 ]
    for cb in callbacks:
        cb.set_model(model)
        # se il callback custom ha on_train_begin, invocatelo qui:
        if hasattr(cb, "on_train_begin"):
            cb.on_train_begin()

model.summary()
print(f"Init Expected Cycle={init_cycle:.1f}, Subcycle={init_sub:.1f}")
print(f"Updated Expected Cycle: {exp_cycle}, Subcycle: {exp_sub}")
print("">>>> FINE-TUNING → 'REG-ONLY' in corso...")
print("">>>> TRANSFER LEARNING => STRATEGY=hybrid → 50% freezing:")
flush_count = 0
for x_seq, y_seq in zip(x_train, y_train):
    if int(y_seq[-1,2]) in (0,1):
        head, hybrid = step_regression_only(
            model, x_seq, y_seq, base_lr, strategy
        )
        # head è un array [loss, rett_mse, rett_mae, sost_mse, sost_mae]
        # aggiorna history con head e - se vuoi - hybrid

```

```

        history["train_remaining_rett_mae"].append(head[2])
        history["train_remaining_rett_mse"].append(head[1])
        history["train_remaining_sost_mae"].append(head[4])
        history["train_remaining_sost_mse"].append(head[3])
    val_logs = model.evaluate(val_ds, verbose=0, return_dict=True)
    flush_count += 1
    for cb in callbacks:
        cb.on_epoch_end(flush_count, logs={"val_remaining_rett_mae": val_logs["remaining_rett_mae"]})

    # -- 5) Salvo pesi
    model.save_weights(FINE_TUNED_WEIGHTS_OUT)
    print(f"→ Pesi migliori modello fine-tuned salvati in {FINE_TUNED_WEIGHTS_OUT}")

    return model, history

```

Tale funzione, come accennato prima, implementa interamente la pipeline di **fine-tuning on-line per i soli rami regressivi** del modello multi-output, partendo da un file JSON di eventi (che andrà poi ad approfondire successivamente nella terza parte), ripetendo tutta la pipeline di pre-processing del main core e arrivando, infine, al salvataggio dei pesi del nuovo modello “fine-tuned” ottimizzato. Innanzitutto, dopo aver eseguito il pre-processing opportuno sui dati incrementali del file JSON, ho ricostruito l’architettura del modello con il metodo “modelling_combined” utilizzato nel core. Dopodiché, ho caricato i pesi di base (o, ricorsivamente, quelli del modello “fine-tuned”, se già presenti) e riallineato il learning rate e la modalità di compilazione per addestrare soltanto le uscite di regressione. Per quanto riguarda, invece, il pre-processing ripetuto iniziale, ho importato e ordinato cronologicamente i record di produzione e manutenzione dal JSON ricevuto, ne ho calcolato, per ognuno, il numero di eventi produttivi intercorsi e le ore lavorate, arricchendo ciascun dizionario con i campi TotalEvents e WorkingHours, proprio come avveniva nel core principale. Questi dati sono stati poi amalgamati - ordinatamente per “Timestamp” - in un nuovo DataFrame, passati attraverso tutte le fasi di pulizia, calcolo dei contatori, normalizzazione delle features numeriche con lo scaler_numeric generato dal core, encoding, segmentazione in cicli e sottocicli, preparazione dei target e, infine, trasformati in sequenze temporali variabili da cui è stato estratto automaticamente un validation set (circa il 20% del totale) e un buffer di training. Per aumentare la variabilità del training ho, anche qui, applicato semplici **data augmentation** sulle feature numeriche e, successivamente, ho ri-costruito due dataset TensorFlow con tf.pipeline: uno per la validazione “on the fly” e uno per il training train-on-batch. Inoltre, prima di entrare nel loop di streaming, ho inizializzato gli stessi callback di monitoraggio (DeltaUpdate, ReduceLROnPlateau, EarlyStopping, ModelCheckpoint ecc.) utilizzati nel core principale e ho effettuato nuovamente zero padding per uniformare le sequenze generate. Nel ciclo principale, per ciascuna sequenza, ho chiamato **step_regression_only** con **train-on-batch**, ho raccolto le metriche regressive MSE/MAE, valutato periodicamente sul validation set e notificato i callback, i quali hanno costantemente adattato, dinamicamente, parametri come il delta della loss o il learning rate per salvare poi i pesi migliori. Al termine della procedura, ho salvato definitivamente il modello con i pesi migliori e ho restituito sia l’istanza della rete aggiornata sia l’history delle metriche raccolte e calcolate sull’andamento attuale.

In questo modo, prima di effettuare un fine-tuning completo su almeno un ciclo di vita totale, si effettua un upgrade passo passo a ogni evento manutentivo, in modo tale da:

- ridurre esponenzialmente gli outlier incombenti una volta che viene effettuato il fine-tuning full
- adattare dolcemente il modello ai nuovi dati in maniera incrementale.

Questo è il principio fondamentale del train-on-batch incrementale. Chiaramente, il processo è molto più veloce e meno oneroso rispetto a quello totale e permette di adattare la regressione costantemente. In questo scenario, tutta la parte inerente al classificatore viene mascherata e non viene considerata apertamente, poiché non è necessaria ai fini tecnici del fine-tuning regression only. Tra l'altro, per gestire l'andamento delle epoche in maniera incrementale, ho dovuto implementare, anche qui, manualmente gli **steps_per_epoch** da seguire per il processamento dei batch nell'addestramento. In questo modo, l'ottimizzatore ha avuto il pieno controllo di quando fermarsi nell'iterazione e di quando proseguire. Infine, il termine “flush” utilizzato in tale procedura, si riferisce al momento in cui si impacchettano i risultati del validation set nei callback e, pertanto, a ogni momento in cui il modello viene valutato.

2. **Fine-tuning: Full**

Il nucleo del fine-tuning vero e proprio però, entra in gioco grazie alla funzione di **fine-tuning full**, dove si va ad adattare il modello completo - comprendente dei due branch regressivi e di quello classificativo – ad ogni nuovo ciclo di vita del piano martire e, pertanto, ad ogni evento manutentivo di Sostituzione incontrato. Infatti, tale funzione realizza un flusso completo di Transfer Learning ibrido su entrambi i branch del modello caricato – che anche qui può essere un modello ricorsivamente fine-tunato, oppure il modello di default generato nel core e importato, per la prima volta, nella procedura di fine-tuning (quindi ancora da “fine-tunare”) - per applicare il metodo più opportuno alla problematica attuale. Nella pratica, prima di tutto, la funzione menzionata ricostruisce l'architettura del modello corrente, caricando i pesi più recenti e scelti automaticamente tra quelli del modello fine-tuned se esso esiste, utilizzando il modello di default altrimenti. Una volta effettuato con successo il caricamento, viene gestito il learning rate, sempre con la tecnica del WarmUpScheduler e ReduceLROnPlateau, con una partenza più lieve e ridotta per evitare picchi instabili da parte del gradiente.

A questo punto, il modello viene ricompilato in modalità “full”, cioè per minimizzare contemporaneamente le perdite di regressione sui conteggi residui e la loss di classificazione del prossimo evento manutentivo e, perciò, tenendo in considerazione tutti i branch del modello stesso, per il ri-addestramento. Di seguito, si espone la funzione, a livello di codice, per intero e se ne spiegano gli effetti:

```

def fine_tune_full(
    json_path: str,
    batch_size: int      = BATCH_SIZE,
    base_weights: str    = BASE_WEIGHTS_PATH,
    lr: float            = LR,
    epochs: int          = EPOCHS_FULL,
    strategy: str        = STRATEGY,
    threshold_factor: float = 0.9,
):
    global _last_compile_mode
    global base_lr

    _last_compile_mode = None
    # --- 0) Build model + load weights/scaler/threshold/expected
    model = core.modelling_combined(num_features=len(core.features),
                                      units=128,
                                      )
    # carico ricorsivamente last_finetunedmodel.weights.h5 se esiste, altrimenti
    il default iniziale
    weights_to_load = FINE_TUNED_WEIGHTS_OUT if
    Path(FINE_TUNED_WEIGHTS_OUT).exists() \
        else BASE_WEIGHTS_PATH
    #load_weights_by_name_skip_mismatch(model, weights_to_load)
    model.load_weights(weights_to_load)
    base_lr = lr * (0.1 if weights_to_load == FINE_TUNED_WEIGHTS_OUT else 1.0)

    ensure_compile(model, base_lr, mode="full")
    model.optimizer.learning_rate.assign(base_lr)

    threshold = load_threshold()
    prev      = load_expected()
    init_cycle = prev["sum_cycle_totals"]/prev["count_cycle_totals"] \
        if prev["count_cycle_totals"]>0 else 0.0
    init_sub   = prev["sum_subcycle_totals"]/prev["count_subcycle_totals"] \
        if prev["count_subcycle_totals"]>0 else 0.0

    # --- 1) Caricamento e “enrichment” dei JSON
    raw_events = json.load(open(json_path))  # lista di dict
    # 1.a) mettiamo in ordine cronologico REGISTRO per REGISTRO
    raw_events = sorted(raw_events, key=lambda e: e["Timestamp"])

    # 1.b) costruiamo due liste di supporto:

```

```

# - prod_list = [(timestamp_produzione, durata_ore), ...]
# - manut_list = [timestamp_manutenzione, ...]
prod_list = []
manut_list = []

# => prima, estraiamo da raw_events tutte le produzioni già presenti
for rec in raw_events:
    if rec.get("EventType") == 1:
        # ricavo la durata in ore: qui End e Start sono ISO8601
        t0 = datetime.fromisoformat(rec["Start"])
        t1 = datetime.fromisoformat(rec["End"])
        durata_h = (t1 - t0).total_seconds() / 3600.0
        prod_list.append((datetime.fromisoformat(rec["Timestamp"]),
durata_h))
    else:
        # record di manutenzione già esistente (magari ha MaintenanceType e
WearValue)
        manut_list.append(datetime.fromisoformat(rec["Timestamp"]))
# 1.c) adesso navighiamo di nuovo i raw_events e, per ciascuna manutenzione,
enriched = []
for rec in raw_events:
    if rec.get("EventType") == 0:
        ts_manut = datetime.fromisoformat(rec["Timestamp"])
        # trovo l'ultima manutenzione precedente (se esiste)
        prev_manut = None
        prevs = [m for m in manut_list if m < ts_manut]
        if prevs:
            prev_manut = max(prevs)
        # filtro le produzioni in (prev_manut, ts_manut]
        if prev_manut is not None:
            prods_int = [(t,d) for (t,d) in prod_list if prev_manut < t <=
ts_manut]
        else:
            prods_int = [(t,d) for (t,d) in prod_list if t <= ts_manut]

        rec = rec.copy()
        rec["TotalEvents"] = len(prods_int)
        rec["WorkingHours"] = round(sum(d for (_,d) in prods_int), 2)

        # aggiungo questo ts_manut alla lista delle manutenzioni già viste,
        # così il prossimo ciclo lo utilizzerà come "precedente"
        manut_list.append(ts_manut)

```

```

# se è produzione, non tocco nulla (lascia TotalEvents magari assente o
1)
    enriched.append(rec)
df = merge_data_list(enriched)
df = core.missing_values(df)
df = core.compute_counters(df)
df = core.encoding(df)
df = core.segment_cycles(df)
df = core.segment_subcycles(df)
df_pre = core.prepare_targets_combined(df)
df_pre["Timestamp"] = df_pre["Timestamp"].astype(np.int64)//10**9
# --- 2) Update expected values
exp_cycle, exp_sub = core.compute_expected_values(df, dedupe=False)
# --- 3) Compute avg_hours
m = df_pre[df_pre['EventType']==0]
default_rett = core.calcola_default_avg_hours(m, 1, "T_Rettifica",
"TotalEvents")
default_sost = core.calcola_default_avg_hours(m, 0, "T_Sostituzione",
"CumulativeEvents")
vr = m[(m['MaintenanceType']==1)&(m['T_Rettifica']>0)]
vs = m[(m['MaintenanceType']==0)&(m['T_Sostituzione']>0)]
avg_rett = (vr['T_Rettifica']/vr['TotalEvents']).mean() \
            if len(vr)>=2 else (default_rett or 0.0)
avg_sost = (vs['T_Sostituzione']/vs['CumulativeEvents']).mean() \
            if len(vs)>=2 else (default_sost or 0.0)
print(f"avg_hours_rett={avg_rett:.3f}, avg_hours_sost={avg_sost:.3f}")

core.scaler_numeric = load_scaler()
numeric_cols = [
    "CumulativeEvents", "SubCycle_Events", "TotalEvents",
    "WorkingHours", "WearValue",
    "Tot_Events_Rett", "Tot_Events_Sost",
    "T_Rettifica", "T_Sostituzione"
]
# normalizzazione applica lo scaler_numeric già caricato in core
core.scaler_numeric.feature_names_in_ = np.array(numeric_cols, dtype=object)
df_pre[numeric_cols] = core.scaler_numeric.transform(df_pre[numeric_cols])
# Prepare all sequences
x_all, y_all, metas_all = core.createRepresentativeSequences(
    df_pre, core.features, core.target, factor=1.0
)
x_all, y_all, metas_all = core.filterValid(x_all, y_all, metas_all)

if len(x_all) == 0:

```

```

        raise ValueError(
            "Nessuna sequenza valida: probabilmente non ci sono sottocicli
completi (EventType=1) "
            "seguiti da una manutenzione (0 o 1). Controlla i dati nel tuo
archive."
        )

# _____
# 4) Stratified split dei set (forzando entrambe le classi in val)
# _____
# estraggo le label genuine
cls_labels = np.array([int(y_seq[-1,2]) for y_seq in y_all])
idxs      = np.arange(len(x_all))
counts    = Counter(cls_labels)

# 4.a) forzo in val un esempio di ciascuna classe, se esistono
forced_val = []
if counts[0] >= 1 and counts[1] >= 1:
    forced_val = [
        int(np.where(cls_labels == 0)[0][0]),
        int(np.where(cls_labels == 1)[0][0])
    ]
# 4.b) costruisco il pool rimanente (senza i due forzati)
pool_idxs = [i for i in idxs if i not in forced_val]

# 4.c) split per il test set (20%) dal pool, se possibile
if len(pool_idxs) >= 2:
    trainval_pool, test_idxs = train_test_split(
        pool_idxs,
        test_size=0.2,
        random_state=42
    )
else:
    # fallback: niente test, tutto rimane in trainval
    trainval_pool = pool_idxs.copy()
    test_idxs     = []

# 4.d) ora estraggo la val dal trainval (20% di trainval), se possibile
if len(trainval_pool) >= 2:
    val_pool, train_idxs = train_test_split(
        trainval_pool,
        test_size=0.2,
        random_state=42

```

```

        )
else:
    val_pool      = []
    train_idxs    = trainval_pool.copy()

# 4.e) ricompongo la val aggiungendo i due forzati
val_idxs = val_pool + forced_val

# 4.f) sicurezza: controllo di aver assegnato tutti gli indici
all_assigned = set(train_idxs) | set(val_idxs) | set(test_idxs)
assert all_assigned == set(idxs), \
    f"Errore nello split: persi {set(idxs) - all_assigned}"

# 4.g) ricreio i set finali
x_train = [x_all[i] for i in train_idxs]
y_train = [y_all[i] for i in train_idxs]
x_val   = [x_all[i] for i in val_idxs]
y_val   = [y_all[i] for i in val_idxs]
x_test  = [x_all[i] for i in test_idxs]
y_test  = [y_all[i] for i in test_idxs]

# _____
# 6) real_train_ds per DynamicAlphaCallback
# _____
X0_real, Y0_real = core.pad_dynamic_sequences(x_train, y_train)
real_train_ds = (
    tf.data.Dataset
        .from_tensor_slices((X0_real, Y0_real))
        .map(lambda x, y: core.custom_map_fn(x, y),
num_parallel_calls=tf.data.AUTOTUNE)
        .map(core.mask_padding, num_parallel_calls=tf.data.AUTOTUNE) # <<< qui!
        .batch(batch_size)
        .prefetch(tf.data.AUTOTUNE)
)
# _____
# 5) Data augmentation SOLO sul train
# _____
x_aug, y_aug = [], []
idx_wh = core.features.index("WorkingHours")
idx_wv = core.features.index("WearValue")
idx_te = core.features.index("TotalEvents")

```

```

for x_seq, y_seq in zip(x_train, y_train):
    x_j = x_seq.copy()
    # piccole variazioni gaussiane su WorkingHours e WearValue
    x_j[:, idx_wh] *= np.random.uniform(0.95, 1.05, size=x_j.shape[0])
    x_j[:, idx_wv] *= np.random.uniform(0.90, 1.10, size=x_j.shape[0])
    # rumore discreto sui TotalEvents
    noise = np.random.randint(-2, 3, size=x_j.shape[0])
    x_j[:, idx_te] = np.clip(x_j[:, idx_te] + noise, a_min=0, a_max=None)
    x_aug.append(x_j)
    y_aug.append(y_seq)

# unisco originali + aumentati
x_train = x_train + x_aug
y_train = y_train + y_aug

# _____
# 6) Costruzione dei dataset TensorFlow
# _____
# padding dinamico
X_tr, Y_tr = core.pad_dynamic_sequences(x_train, y_train)
if len(x_val) == 0:
    # prendi l'ultimo elemento di train
    x_val = [x_train[-1]]
    y_val = [y_train[-1]]
    # rimuovilo da train
    x_train = x_train[:-1]
    y_train = y_train[:-1]
    print("ATTENZIONE: val era vuoto - ho spostato 1 campione da train a
val.")
X_va, Y_va = core.pad_dynamic_sequences(x_val, y_val)
X_te, Y_te = core.pad_dynamic_sequences(x_test, y_test)

def make_dataset(X_list, Y_list, batch_size, shuffle=False):
    ds = tf.data.Dataset.from_tensor_slices((X_list, Y_list))
    ds = ds.map(lambda x, y: core.custom_map_fn(x, y),
               num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(core.mask_padding,
               num_parallel_calls=tf.data.AUTOTUNE)
    if shuffle:
        ds = ds.shuffle(len(X_list))
    return ds.batch(batch_size).prefetch(tf.data.AUTOTUNE)

```

```

train_ds = make_dataset(X_tr, Y_tr, batch_size=batch_size, shuffle=True)
val_ds   = make_dataset(X_va,   Y_va, batch_size=batch_size, shuffle=False)
test_ds  = make_dataset(X_te,   Y_te, batch_size=batch_size, shuffle=False)

# 7) Model + compile_full
ensure_compile(model, base_lr, mode="full")
model.optimizer.learning_rate.assign(base_lr)

# 8) Callbacks standard + ClassificationFlushCallback
delta_cb      = core.DeltaUpdateCallback(val_ds, percentile=98, smoothing=0.2,
min_delta=0.15)
roc_auc_cb    = core.RocAucCallback(val_ds)
dyn_alpha_cb = core.DynamicAlphaCallback(real_train_ds)
reduce_lr_cb = tf.keras.callbacks.ReduceLROnPlateau(
                monitor="val_next_maint_type_last_step_auc", mode="max",
                factor=0.5, patience=2, min_lr=1e-6, verbose=1)
early_stop_cb= tf.keras.callbacks.EarlyStopping(
                monitor="val_next_maint_type_last_step_auc", mode="max",
                patience=5, restore_best_weights=True)
checkpoint_cb= ModelCheckpoint(
                filepath=FINE_TUNED_WEIGHTS_OUT,
                monitor="val_next_maint_type_last_step_auc",
                mode="max", save_best_only=True, save_weights_only=True,
verbose=1)

# il nuovo callback per gestire i flush delle epoche
flush_cb = ClassificationFlushCallback(
    buffer=list(zip(x_train, y_train)),
    strategy=strategy,
    lr=base_lr,
    val_ds=val_ds
)

# assegno il model a TUTTI i callback
callbacks = [delta_cb, roc_auc_cb, reduce_lr_cb,
            early_stop_cb, checkpoint_cb, flush_cb,
            #dyn_alpha_cb
            ]
for cb in callbacks:
    cb.set_model(model)
    # se il callback custom ha on_train_begin, invocatelo qui:
    if hasattr(cb, "on_train_begin"):
        cb.on_train_begin()

```

```

# 9) Fit
model.summary()
print(f"Init Expected Cycle={init_cycle:.1f}, Subcycle={init_sub:.1f}")
print(f"Updated Expected Cycle: {exp_cycle}, Subcycle: {exp_sub}")
print("">>>> Stratified data split → train/val/test set")
print("">>>> FINE-TUNING → 'FULL' in corso...")
print("">>>> TRANSFER LEARNING => STRATEGY=hybrid → 50% freezing:")
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs,
    callbacks=callbacks
).history

# --- 10) Recompute threshold on val set
y_val_true, y_val_scores = [], []
output_names = model.output_names
# -- 10) Recompute threshold on val set --
for xb, y_dict, _ in val_ds:
    raw = model.predict(xb, verbose=0)
    # se non è un dict, lo trasformo in dict con le chiavi corrette
    if not isinstance(raw, dict):
        raw = { name: raw[i] for i, name in enumerate(output_names) }
    sc = raw["next_maint_type"].reshape(-1)
    y_val_scores.extend(sc.tolist())
    yt = y_dict["next_maint_type"][:, -1, 0].numpy()
    y_val_true.extend(yt.tolist())
    # estraggo "next_maint_type" dal dizionario dei label
p, r, t = precision_recall_curve(y_val_true, y_val_scores)
best_t = float(t[np.argmax(2*p*r/(p+r+1e-8))])
#threshold = best_t * threshold_factor
threshold = best_t
save_threshold(threshold)

# --- 11) Final test evaluation + metrics
y_true, y_scores = [], []
# -- 11) Final test evaluation --
for xb, y_dict, _ in test_ds:
    raw = model.predict(xb, verbose=0)
    if not isinstance(raw, dict):
        raw = { name: raw[i] for i, name in enumerate(output_names) }
    prob = raw["next_maint_type"].reshape(-1)

```

```

y_scores.extend(prob.tolist())
yt = y_dict["next_maint_type"].numpy().flatten()
y_true.extend(yt.tolist())
y_pred = [1 if p>=threshold else 0 for p in y_scores]
cm = confusion_matrix(y_true, y_pred)
cr = classification_report(y_true, y_pred, digits=3)

model.save_weights(FINE_TUNED_WEIGHTS_OUT)
print(f"→ Pesi migliori modello fine-tuned salvati in
{FINE_TUNED_WEIGHTS_OUT}")

return model, history, cm, cr

```

Come si evince dal lungo body della funzione, il primo grande blocco di codice estrae da **json_path** tutti gli eventi, li ordina cronologicamente e separa quelli di produzione da quelli di manutenzione con il solito flag di supporto utilizzato anche precedentemente. Poi, per ogni manutenzione calcola il numero di eventi produttivi e le ore lavorate intercorse dall'ultima manutenzione, arricchendo, anche qui, ogni record con i campi **TotalEvents** e **WorkingHours**. Quindi, tutti i dizionari vengono composti in un DataFrame e sottoposti alle consuete e solite operazioni di pulizia, imputazione, normalizzazione, encoding, segmentazione in cicli e sottocicli, preparazione dei target e infine trasformati in sequenze dinamiche di input **x_all** e di target **y_all** resistenti a padding e mascheramento, secondo la logica effettiva del problema. Come avviene nel core principale, anche qui lo split non viene effettuato in maniera casuale, bensì si esegue uno **split stratificato**, mantenendo, nel validation set, almeno un esempio di ciascuna classe di evento manutentivo, per avere un set di validazione sempre bilanciato e non vuoto. Allo stesso modo, tale split viene effettuato anche sul test set, al quale viene riservato un 20% del pool restante, proprio come accade con il validation set. Per evitare di sollevare un'eccezione di errore, anche in questo modulo ho applicato un fallback, di splitting casuale, in caso di numero di esempi troppo esegui a disposizione per poter effettuare una stratificazione coerente. Anche se il fine-tuning viene applicato su un numero di dati minore rispetto all'addestramento iniziale vero e proprio, è comunque utile mantenere un approccio di stratificazione in presenza di problemi sbilanciati anche nel deployment in produzione. Per garantire una pipeline riprodotta e completamente coerente con quella del main core, anche in questo caso, ho adottato un approccio di augmentazione – con piccole variazioni di rumore introdotte sulle features numeriche - ed oversampling sulla classe minoritaria per migliorare la robustezza e la generalizzazione.

Come da prassi, anche qui si costruiscono sempre i tre **tf.data.Dataset** di interesse con **tf.pipeline** di **Tensorflow**:

- **Training set:** La parte preponderante del dataset (circa 80%), opportunamente shuffled e batched come nel core.
- **Validation set:** Circa il 20% del training set, monitorato con fallback in caso di troppi pochi esempi
- **Test set:** Anche qui circa il 20% del train/val set e usato per il testing delle prestazioni del modello fine-tuned.

Tutti i vari set dispongono di mascheramento e mapping personalizzato per estrarre solo l'ultimo step (quello “portavoce”) per gestire il branch classificativo e lo zero padding ed evitare qualunque inquinamento prodotto

da essi. Il modello viene poi compilato nuovamente per il fine-tuning full e dotato di una suite di callback, basata sugli stessi identici callbacks presenti nel core con l'aggiunta, però, di uno ulteriore: ClassificationFlushCallback. Esso è un callback personalizzato, pensato per integrare il ramo di classificazione all'interno di un training a flusso continuo real-time, senza dover attendere un'epoca tradizionale come si farebbe nei modelli statici. Nella pratica, all'atto della creazione, riceve in input un buffer contenente tutte le coppie (x_{seq} , y_{seq}) possibili del training set in cui, ogni y_{seq} , include l'informazione relativa all'**ultimo timestep** di classificazione NextMaintenanceType. Dopodiché, ad ogni chiamata del metodo “**on_epoch_end**”, il callback itera su ciascuna sequenza del buffer in modo ciclico effettuando tra operazioni principali in successione:

1. Estraе l'ultimo passo di ogni y_{seq} e, se indica un evento di manutenzione (cioè una classe valida NextMaintenanceType 0 o 1), richiama la routine **step_classification_flush(model, x_seq, y_seq, lr, strategy)** che analizzerà successivamente.
2. Quella routine esegue un **train_on_batch** **solo** sul ramo di classificazione (la testa Dense di output **next_maint_type**), lasciando congelati i pesi per la regressione.
3. Registra la loss di classificazione e aggiorna dinamicamente parametri come learning rate o soglia di decisione tramite i callbacks introdotti.

Questa logica è molto versatile, perché permette di scaricare continuamente tutti gli esempi di classificazione già disponibili, anche in assenza di epoche ben delimitate, garantendo che il ramo **next_maint_type** venga ottimizzato in parallelo a quello di regressione e riceva aggiornamenti mirati non appena emergono nuovi esempi di manutenzione. In conclusione, il classification flush callback non è altro che un addestratore dinamico on-demand, per la testata di classificazione, in un contesto di training ibrido e live streaming, il quale percorre il buffer di sequenze, estrae i casi rilevanti e li passa al modello tramite train-on-batch, mantenendo aggiornato il ramo classificativo ad ogni nuovo ciclo di vita registrato.

Successivamente, il training full procede con la direttiva `model.fit()` sul set di training on-the-fly: **train_ds**, per un numero prefissato di epoche, monitorando via callback sia le loss regressive, sia le metriche di classificazione sul validation set alla fine di ogni iterazione. Al termine di tutte le epoche, la funzione ricampiona le probabilità predette sul validation set, per ricomputare correttamente la soglia ottimale di classificazione tramite la curva PR-AUC, ne salva il valore corrispondente e valuta il modello sul test set, generando matrice di confusione e classification report correlati come nel main core. Infine, i pesi migliori vengono salvati in un file .h5 e il modello aggiornato, derivato da essi, viene restituito in output tramite la direttiva “return”.

5.2 Strategie di transfer learning

Adesso, è necessario trattare la vera e propria radice del fine-tuning: il **Transfer Learning**. Esso è considerato una tecnica di addestramento che sfrutta un modello già pre-allenato su un ampio insieme di dati generici, per adattarlo a un nuovo compito più specifico, congelando la conoscenza pregressa e risparmiando tempo di calcolo e quantità di dati etichettati. In pratica, si parte da un modello di default che ha già appreso rappresentazioni utili (feature di basso livello, legami tra di esse ecc.) e si riutilizzano quei pesi iniziali, mantenendoli non aggiornabili durante il fine-tuning, mentre invece, si sbloccano e ri-allenano solo i livelli finali – di solito una o più teste di output – in modo da specializzarli ai nuovi target o alla nuova distribuzione di dati. Questo sfrutta la **generalità** delle prime fasi del modello - dove si catturano caratteristiche di base come edges, texture o pattern temporali - e concentra l'apprendimento raffinato sulle parti finali e specifiche, riducendo drasticamente il rischio di overfitting e la quantità di dati necessari. Tale meccanismo assume un'importanza notevole, poiché permette al modello di salvaguardare la propria conoscenza pregressa, mitigare il fenomeno di “catastrophing forgetting” e riaggiornarsi, in maniera meno invasiva, più veloce ed incrementale, su tutti i nuovi dati e scenari successivi. Avendo ideato un fine-tuning duplice per il ramo regressivo e classificativo, anche per quanto riguarda il Transfer Learning, ho dovuto agire allo stesso modo, rispettando i canoni di specializzazione dettati dalla logica duale del mio modello. Infatti, nella pipeline di fine-tuning, ho messo a punto ben due strategie differenti di Transfer Learning, pensate per intervenire in modo chirurgico sul modello senza andare a riscrivere da zero l'intera rete.

1. Step regression-only

Questa strategia di aggiornamento viene invocata ad ogni nuova sequenza di training, nel flusso di fine-tuning inherente solamente ai branch regressivi: “**regression-only**”. Di seguito, il codice della funzione utilizzata:

```
def step_regression_only(model, x_seq, y_seq, base_lr, strategy="hybrid"):
    # prepara batch
    Xb, Yb = core.pad_dynamic_sequences([x_seq], [y_seq])
    # -- 1) congelo tutti i layer
    for l in model.layers:
        l.trainable = False
    # 2) sblocco solo la testa regressiva
    for name in ("remaining_rett_lin", "remaining_sost_lin"):
        model.get_layer(name).trainable = True
    # se hybrid, sblocco anche tutti i LayerNorm
    if strategy == "hybrid":
        for l in model.layers:
            if isinstance(l, tf.keras.layers.LayerNormalization):
                l.trainable = True
    # -- 3) ricompilo solo se non ero già in "reg"
    reg_lr = base_lr * 0.1
```

```

ensure_compile(model, reg_lr, mode="reg")
model.optimizer.learning_rate.assign(reg_lr)

# -- 4) passo head-only
head_loss = model.train_on_batch(
    Xb,
    {
        "remaining_rett": Yb[:, :, 0:1],
        "remaining_sost": Yb[:, :, 1:2],
        # azzero l'output classification perché uso solo uscite reg
        "next_maint_type": np.zeros((Xb.shape[0], 1), dtype=np.float32)
    }
)

hybrid_loss = None
if strategy == "hybrid":
    # -- 5) fase hybrid end-to-end
    for l in model.layers:
        l.trainable = True
    model.optimizer.learning_rate.assign(base_lr * 0.01)
    hybrid_loss = model.train_on_batch(
        Xb,
        {
            "remaining_rett": Yb[:, :, 0:1],
            "remaining_sost": Yb[:, :, 1:2],
            "next_maint_type": np.zeros((Xb.shape[0], 1), dtype=np.float32)
        }
    )

return head_loss, hybrid_loss

```

Tale routine è fondamentale per vari motivi:

1. **Congelamento selettivo:** Tutti i layer del modello vengono, inizialmente, marcati come “**non trainable**”, preservando tutti i pesi pre-allenati che catturano conoscenze generali dal pre-training di default.
2. **Sblocco delle teste di regressione:** Come di prassi in molte strategie di Transfer Learning, anche in questa situazione, ho optato per ri-abilitare al training solo i due layer di output finali che producono le uscite di regressione: **Remaining_rett** e **Remaining_Sost**. Oltretutto, se il flag “strategy” in input alla funzione è settato su “hybrid” (che di default è il comportamento atteso dal mio modello), allora

vengono ri-addestrati anche tutti i layer di LayerNormalization assieme alle teste finali, in modo tale da permettere leggeri adattamenti delle normalizzazioni di batch intermedie e, pertanto, mantenere gran parte dei filtri originali intatti.

3. **Ricompilazione head-only:** Per inferire un corretto meccanismo di fine-tuning, il modello viene ricompilato con learning rate ridotto ($\text{base_lr} * 0.1$) rispetto all'originale. Anche questa è una tecnica comune nel meccanismo di raffinamento, poiché si cerca di non fare oscillare troppo il gradiente... soprattutto in un contesto con pochissimi dati incrementali. Allo stesso modo, per il ri-addestramento regressivo, viene sempre presa in considerazione la loss di regressione – Dynamic Huber – in modo tale da adattare il nuovo apprendimento a quello di default. La classificazione, invece, in questo caso specifico ha peso nullo, poiché è irrilevante in un addestramento prettamente regressivo.
4. **Train-on-batch:** Il cuore del meccanismo di fine-tuning del ramo regressivo che ho implementato. Viene eseguito, difatti, un solo train-on-batch sulle due teste di regressione, utilizzando, come target, i valori reali di Remaining_rett e Remaining_sost e azzerando, forzatamente, l'output di classificazione che è inutile in questo contesto. In questo modo, il modello affina da subito le proprie capacità di prevedere gli eventi produttivi rimanenti, senza andare a deviare risorse computazionali sul ramo classificativo.
5. **Hybrid end-to-end:** Dopo la strategia “hybrid” che riaddestra i LayerNormalization e che si innesca immediatamente dopo aver eseguito il fine-tuning head only, si va ad effettuare, gradualmente e in maniera molto leggera, un ulteriore train-on-batch con learning rate ancora più ridotto ($\text{base_lr} * 0.01$). Questa è una seconda passata che, inizialmente, potrebbe risultare ridondante. Tuttavia, consente invece di ricalibrare, in modo soft, anche le rappresentazioni di basso e medio livello, garantendo che le modifiche alla regressione si armonizzino con le features globali, senza andarle a stravolgerle. È un passaggio, chiaramente, non obbligatorio, ma fortemente consigliato per allineare, in modo coerente, tutti i layer del modello sui nuovi dati, senza esercitare un'influenza pesante e rumorosa.

2. Step *classification-flush*

Questa seconda strategia, invece, alimenta il ramo di classificazione in un flusso continuo (flush). Di seguito, il codice della funzione utilizzata:

```
def step_classification_flush(model, buffer, strategy, base_lr):
    # - preparazione batch bilanciato -
    Xb_f, Yb_f = core.pad_dynamic_sequences(
        [x for x, _ in buffer], [y for _, y in buffer]
    )
    y_last = Yb_f[:, -1, 2]
    idx0 = np.where(y_last == 0)[0]
    idx1 = np.where(y_last == 1)[0]
    if len(idx0) + len(idx1) == 0:
        return None, None

    # oversampling 50/50
    if len(idx0) < len(idx1):
        minor_idx, maj_idx = idx0, idx1
    else:
        minor_idx, maj_idx = idx1, idx0
    n_min, n_maj = len(minor_idx), len(maj_idx)
    if 0 < n_min < n_maj:
        extra = np.random.choice(minor_idx, size=(n_maj - n_min), replace=True)
        minor_idx = np.concatenate([minor_idx, extra])
    sel = np.concatenate([maj_idx, minor_idx])
    np.random.shuffle(sel)
    Xb_bal, Yb_bal = Xb_f[sel], Yb_f[sel]

    # congela / sblocca
    # Freezo tutto
    for l in model.layers:
        l.trainable = False
    # e sblocco adapter + testa di classificazione
    for name in ("adapter_hidden", "adapter_out", "adapter_residual",
"adapter_dropout", "adapter_ln", "next_maint_type"):
        model.get_layer(name).trainable = True
    if strategy == "hybrid":
        for l in model.layers:
            if isinstance(l, tf.keras.layers.LayerNormalization):
                l.trainable = True

    # congela / sblocca e ricompila
```

```

ensure_compile(model, base_lr, mode="full")
model.optimizer.learning_rate.assign(base_lr)

y_dict = {
    "remaining_rett": Yb_bal[:, :, 0:1],
    "remaining_sost": Yb_bal[:, :, 1:2],
    "next_maint_type": Yb_bal[:, -1:, 2:3]
}

# - head-only train -
_ = model.train_on_batch(Xb_bal, y_dict)
train_res = model.test_on_batch(Xb_bal, y_dict, return_dict=True)

# - hybrid end-to-end -
hybrid_res = None
if strategy == "hybrid":
    for l in model.layers: l.trainable = True
    model.optimizer.learning_rate.assign(base_lr * 0.005)
    _ = model.train_on_batch(Xb_bal, y_dict)
    hybrid_res = model.test_on_batch(Xb_bal, y_dict, return_dict=True)
return train_res, hybrid_res

```

Questa funzione, a differenza di quella di step regression-only, integra una logica più dettagliata e complessa, poiché tiene conto di entrambi i branch del modello e si adatta di conseguenza:

- Bilanciamento dei batch:** Il problema della classificazione sbilanciata si ripercuote sempre per cui, anche in un contesto di fine-tuning, bilanciare correttamente le etichette all'interno dei batch è fondamentale. Si estraggono le etichette dell'ultimo step `next_maint_type`, si separano gli indici delle classi minoritarie e maggioritarie come nel main core e si fa oversampling della classe minoritaria in modo tale da distribuire, in maniera bilanciata con distribuzione 50/50, le classi nei vari batch.
- Congelamento selettivo:** Si disabilitano, anche qui, tutti i layer tranne quelli coinvolti nel meccanismo di Adapter leggero utilizzato nel backbone del core. Infatti, i layer: `adapter_hidden`, `adapter_out`, `adapter_residual`, `adapter_dropout` e `adapter_ln` assieme alla testa Dense finale di classificazione `next_maint_type` vengono mantenuti addestrabili. Analogamente alla modalità step regression-only, anche in questo contesto, se utilizzata la modalità “hybrid”, si lasciano ri-allenabili anche i layer di normalizzazione LayerNormalization, per adattare la normalizzazione ai nuovi dati.
- Ricompilazione full-mode:** Il modello viene ricompilato in modalità “full”, utilizzando tutte le loss opportune e le metriche di regressione e classificazione, ma con learning rate tarato in maniera più leggera e, come nella procedura step regression-only, ridotto ulteriormente per una seconda passata in modalità “hybrid”.
- Head-only classification:** Viene eseguito `train-on-batch` sul batch bilanciato, allenando quindi soltanto l'adapter e la testa di classificazione, mentre il resto della rete rimane congelato. Questo consente di adattare rapidamente il classificatore alle nuove etichette senza rischiare di corrompere le feature generali.

5. **Test on batch:** Immediatamente dopo, si invoca una procedura di test-on-batch, per ottenere un dizionario di metriche utilizzate (loss, accuracy, precision, recall, AUC, F1-score) che verranno messe a disposizione nei callback in uso.
6. **Hybrid end-to-end:** Anche in questo caso, se configurato correttamente “hybrid”, si sbloccano tutti i layer e si effettua un secondo e ulteriore **train-on-batch** e **test-on-batch** a learning rate ancora più ridotto ($\text{base_lr} * 0.005$), migliorando delicatamente anche le convenzioni di feature extraction.

Con la combinazione parallela di questi due approcci, si segue il principio cardine del **freezing e unfreezing graduale** tipico del Transfer Learning ibrido: ovvero, mantenere intatti i livelli più bassi pre-addestrati e aggiornare progressivamente gli strati più alti, limitando costantemente il rischio di overfitting e riducendo drasticamente i tempi di addestramento su dati nuovi. Tutto questo permette di integrare, incrementalmente, nuovi esempi in streaming in modo modulare e flessibile, aggiornando ciascun branch del modello esattamente quando serve e con la granularità massima richiesta.

5.3 Valutazione delle prestazioni e risultati ottenuti

Dopo aver eseguito il fine-tuning con successo su dati simulati, a puro scopo di testing delle prestazioni, ho ottenuto risultati promettenti, considerando che ho utilizzato pochissimi dati completi per riaddestrare il modello e non li ho forniti in maniera incrementale come ci si dovrebbe aspettare, bensì in blocchi statici iniziali generati dalla solita funzione di simulazione del dataset. Innanzitutto, voglio mostrare uno screenshot del terminale in fase di preparazione al processing del metodo di fine-tuning propagato su cinque epoch consecutive:

```
C:\WINDOWS\system32\cmd. + 
dropout_4 (Dropout)      (None, None, 64)          0   layer_normalization_5[0][...
layer_normalization_8 (LayerNormalization) (None, None, 64)        128  my_time_distributed_1[0][...
adapter_ln (LayerNormalization) (None, 32)           64   adapter_residual[0][0]
remaining_rett_lin (MyTimeDistributed) (None, None, 1)        65   dropout_4[0][0]
dropout_6 (Dropout)      (None, None, 64)          0   layer_normalization_8[0][...
next_maint_type (Dense) (None, 1)                 33   adapter_ln[0][0]
remaining_rett (Lambda)  (None, None, 1)           0   remaining_rett_lin[0][0]
remaining_sost (MyTimeDistributed) (None, None, 1)        65   dropout_6[0][0]

Total params: 1,119,272 (4.27 MB)
Trainable params: 1,119,272 (4.27 MB)
Non-trainable params: 0 (0.00 B)
Init Expected Cycle=13615.2, Subcycle=498.2
Updated Expected Cycle: 13607, Subcycle: 498
>>> Stratified data split → train/val/test set
>>> FINE-TUNING → 'FULL' in corso...
>>> TRANSFER LEARNING => STRATEGY=hybrid → 50% freezing:
Epoch 1/5
```

Come si può notare, il metodo di Transfer Learning utilizzato nella simulazione è quello “full” poiché, nella funzione di simulazione del dataset attuale ho generato almeno un ciclo di vita completo, con molte meno manutenzioni e produzioni rispetto ai cicli del dataset di default iniziale. I risultati ottenuti, a livello di performance nella ultima epoca e nella validazione, sono stati questi:

```
Epoch 5 ROC-AUC: 1.0000

Epoch 5: val_next_maint_type_last_step_auc did not improve from 1.00000

1/1 1s 1s/step - loss: 0.9202 -
next_maint_type_last_step_accuracy: 0.5000 - next_maint_type_last_step_auc: 0.9397 -
next_maint_type_last_step_f1: 0.6667 - next_maint_type_last_step_precision: 1.0000 -
next_maint_type_last_step_recall: 0.5000 - next_maint_type_loss: 0.1774 - remaining_rett_loss: 3.1312e-04 -
remaining_rett_mae: 0.5125 - remaining_rett_mse: 0.5006 - remaining_sost_loss: 0.1270 -
remaining_sost_mae: 0.4457 - remaining_sost_mse: 0.2540 - val_loss: 0.7387 -
val_next_maint_type_last_step_accuracy: 1.0000 - val_next_maint_type_last_step_auc: 0.9417 -
val_next_maint_type_last_step_f1: 1.0000 - val_next_maint_type_last_step_precision: 1.0000 -
val_next_maint_type_last_step_recall: 1.0000 - val_next_maint_type_loss: 0.0458 - val_remaining_rett_loss: 0.0751 -
val_remaining_rett_mae: 0.9053 - val_remaining_rett_mse: 1.0321 - val_remaining_sost_loss: 0.0794 -
val_remaining_sost_mae: 0.8690 - val_remaining_sost_mse: 0.8855 - learning_rate: 1.0000e-08 -
flush_train_loss: 1.4215 - flush_train_next_maint_type_last_step_accuracy: 0.8333 -
flush_train_next_maint_type_last_step_auc: 0.9435 - flush_train_next_maint_type_last_step_f1: 0.8889 -
flush_train_next_maint_type_last_step_precision: 1.0000 - flush_train_next_maint_type_last_step_recall: 0.8333 -
flush_train_next_maint_type_loss: 0.0856 - flush_train_remaining_rett_loss: 0.4037 -
flush_train_remaining_rett_mae: 0.8401 - flush_train_remaining_rett_mse: 1.0176 -
flush_train_remaining_sost_loss: 0.0919 - flush_train_remaining_sost_mae: 0.5977 -
flush_train_remaining_sost_mse: 0.4881 - flush_hybrid_loss: 1.3140 -
flush_hybrid_next_maint_type_last_step_accuracy: 0.9000 - flush_hybrid_next_maint_type_last_step_auc: 0.9452 -
flush_hybrid_next_maint_type_last_step_f1: 0.9333 -
flush_hybrid_next_maint_type_last_step_precision: 1.0000 - flush_hybrid_next_maint_type_last_step_recall: 0.9000 -
flush_hybrid_next_maint_type_loss: 0.0658 - flush_hybrid_remaining_rett_loss: 0.3646 -
flush_hybrid_remaining_rett_mae: 0.7768 - flush_hybrid_remaining_rett_mse: 0.8773 -
flush_hybrid_remaining_sost_loss: 0.0653 - flush_hybrid_remaining_sost_mae: 0.4605 -
flush_hybrid_remaining_sost_mse: 0.3339 - flush_val_loss: 0.7387 -
flush_val_next_maint_type_last_step_accuracy: 1.0000 - flush_val_next_maint_type_last_step_auc: 0.9500 -
flush_val_next_maint_type_last_step_f1: 1.0000 - flush_val_next_maint_type_last_step_precision: 1.0000 -
flush_val_next_maint_type_last_step_recall: 1.0000 - flush_val_next_maint_type_loss: 0.0458 -
flush_val_remaining_rett_loss: 0.0751 - flush_val_remaining_rett_mae: 0.9053 -
flush_val_remaining_rett_mse: 1.0320 - flush_val_remaining_sost_loss: 0.0794 -
flush_val_remaining_sost_mae: 0.8690 - flush_val_remaining_sost_mse: 0.8855
```

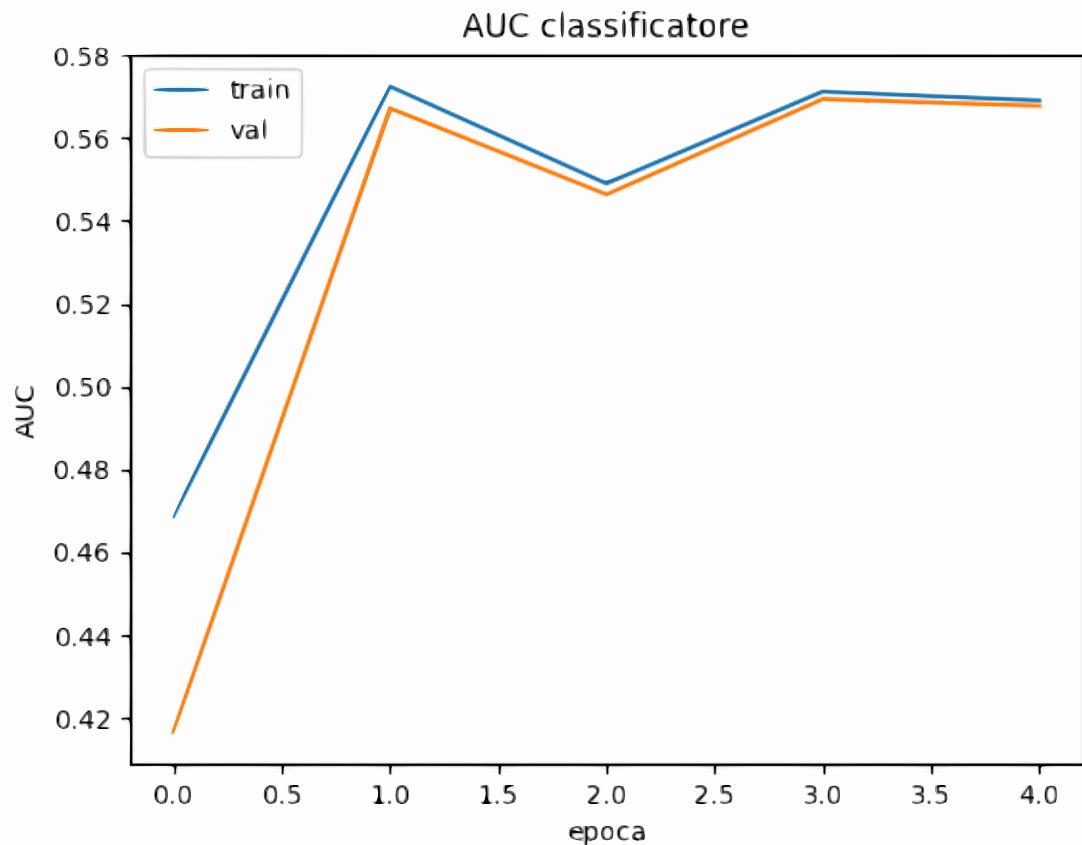
Da queste prestazioni si evince, fondamentalmente, che il processo di Transfer Learning e Fine-tuning sta lavorando davvero bene. L’idea di tenere congelato tutto il backbone principale è stata vincente, in quanto il modello, nel processo di raffinamento, anche con pochi dati è riuscito a mantenere un’AUC elevatissima in

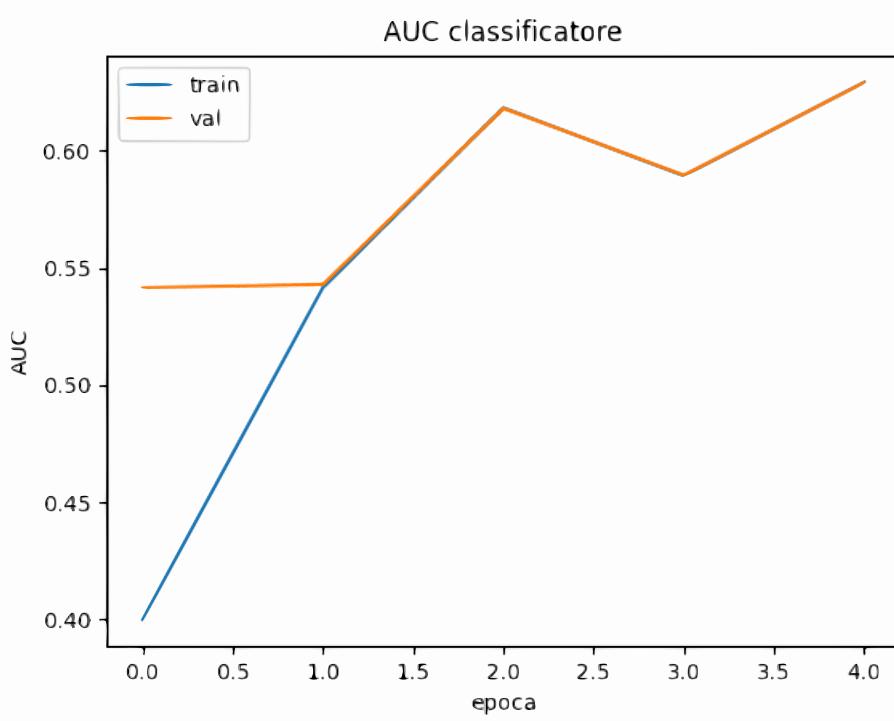
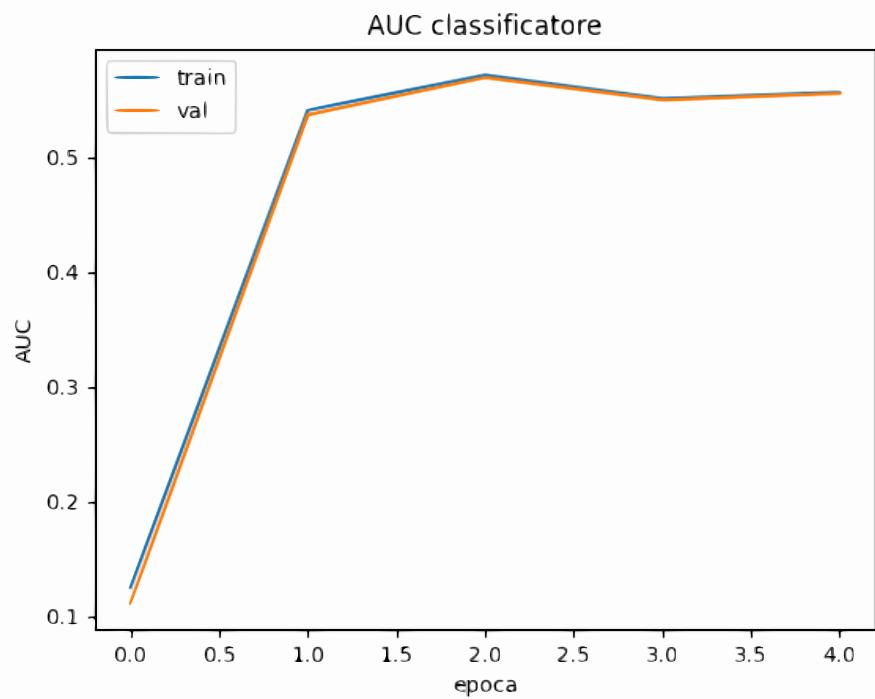
validazione, anche più elevata dell'addestramento di base. L'overfitting non si è verificato, in quanto la loss continua costantemente a decrescere sia in training che in validazione e, anche l'AUC e le altre metriche inerenti, rimangono alte in entrambi i contesti, ad eccezione dell'Accuracy che è però fuorviante in questo ambito. La **recall**, a momenti alterni sul training set, subisce alcune oscillazioni e, di conseguenza, anche la F1-Score, però – come ho potuto riscontrare anche sul web - è sicuramente dipendente dal fatto di avere pochissimi nuovi dati a disposizione per sostenere un processo di raffinamento approfondito. Tuttavia, nel validation set e nei flush delle epoche, i valori delle metriche rimangono solidi, robusti ed elevati. Sarebbe interessante, ripetere un fine-tuning con molti più dati accumulati e questo sarà proprio il principio del deployment in produzione di tale applicativo software. I risultati sul test set, con questa particolare configurazione, non sono alquanto rilevanti, perché avevo a disposizione troppi pochi dati per popolarlo correttamente e le performance ottenute su di esso non sono rilevanti e veritieri. È, pertanto, consigliabile effettuare fine-tuning quando il numero di dati disponibili sono un numero abbastanza considerevole, da rendere ragionevole lo stratified split di tutti e tre i set. Proprio per questa ragione, ho scelto di attuare, a livello logico, la suddivisione in cicli di vita anche per il riaddestramento e, in particolare, - come vedremo poi nella terza parte della tesi – rendere il fine-tuning “full” attuabile solo quando il numero di cicli di vita a disposizione nel buffer è di un numero N ragionevole a non creare scompensi e a performare un raffinamento corretto.

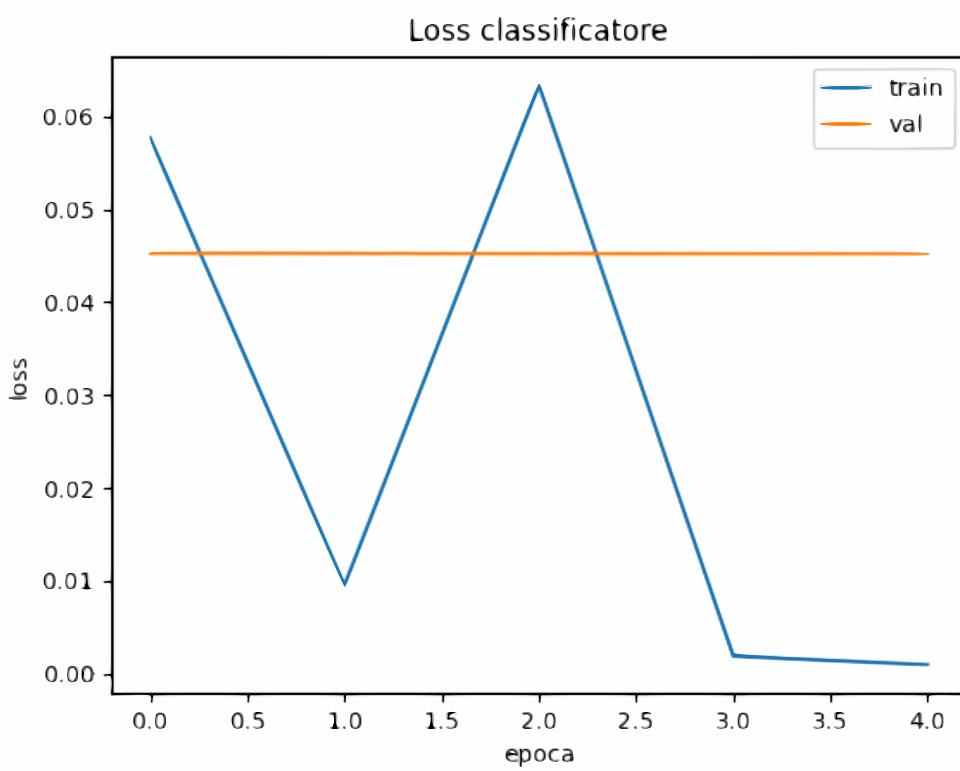
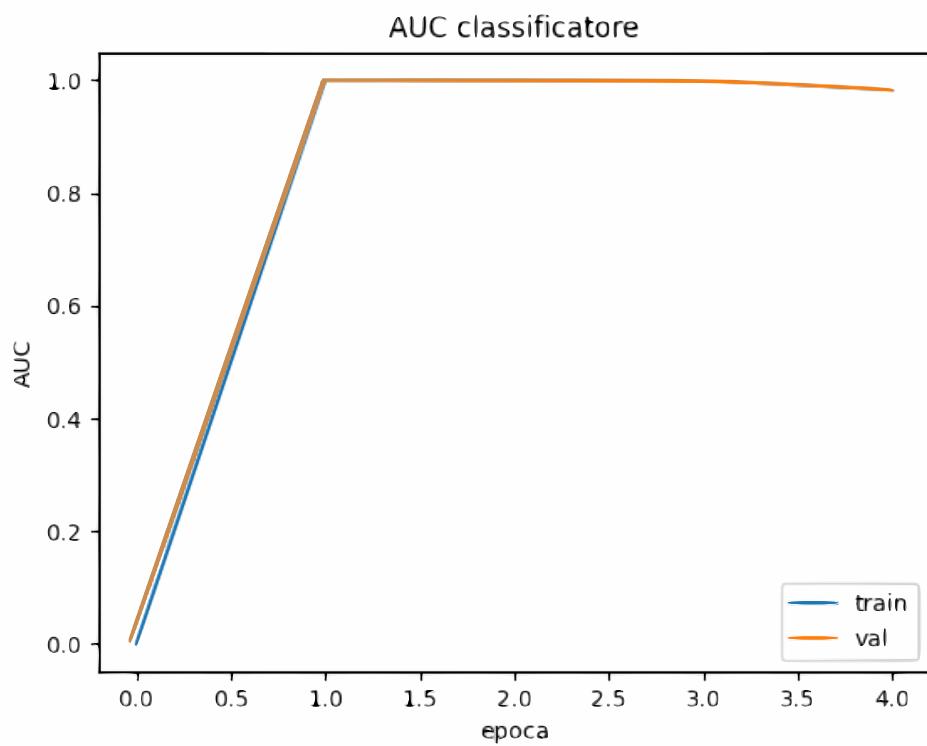
I grafici inerenti alle ultime 2-3 pagine del prossimo capitolo, sono gli indicatori delle prestazioni attuali del modello e di come è stato caricato nella fase di deployment in produzione. È, chiaramente, comprensibile che attualmente, le metriche di regressione, pur mostrando un andamento tendente al positivo, sono leggermente inquinate dall'esiguità intrinseca dei dati a disposizione. Per ottenere prestazioni ancora più soddisfacenti, è imperativo aggiungere più dati totali. Tutto sommato però, con i pochi record attuali e con il processo di Transfer Learning creato, mi ritengo abbastanza soddisfatto, in termini di conoscenza preservata per quanto riguarda il modello neurale stesso.

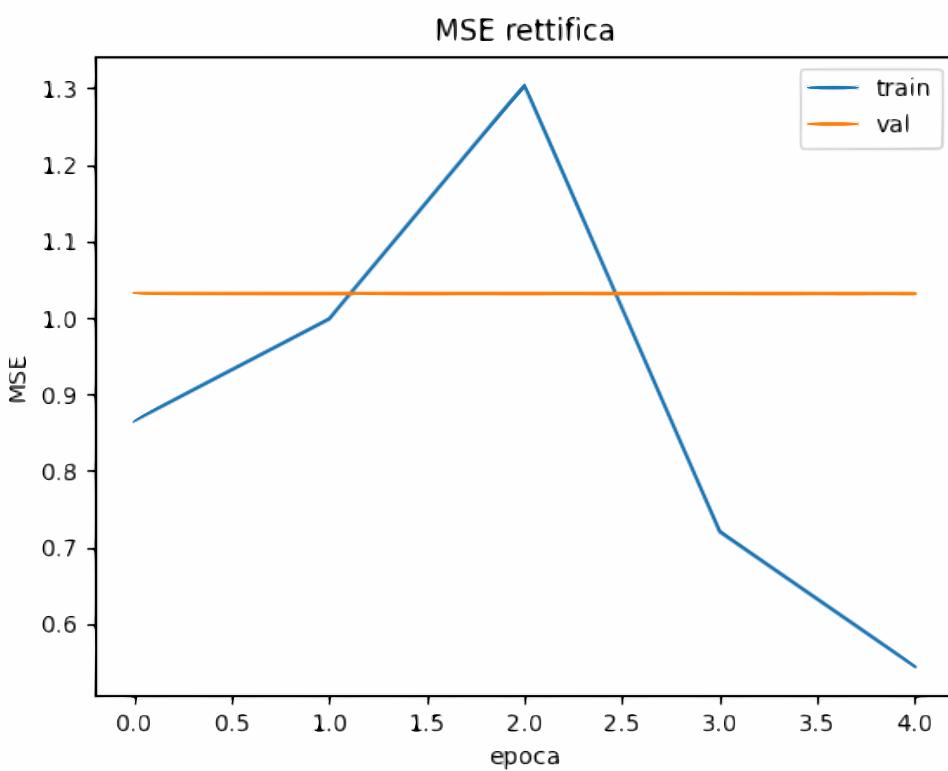
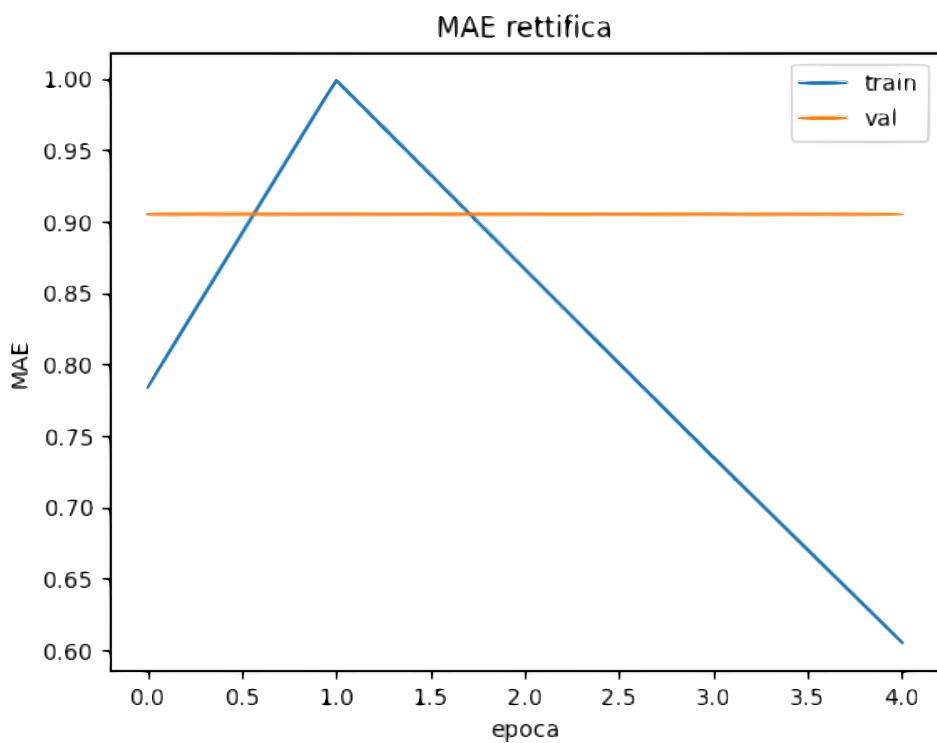
5.4 Grafici delle metriche

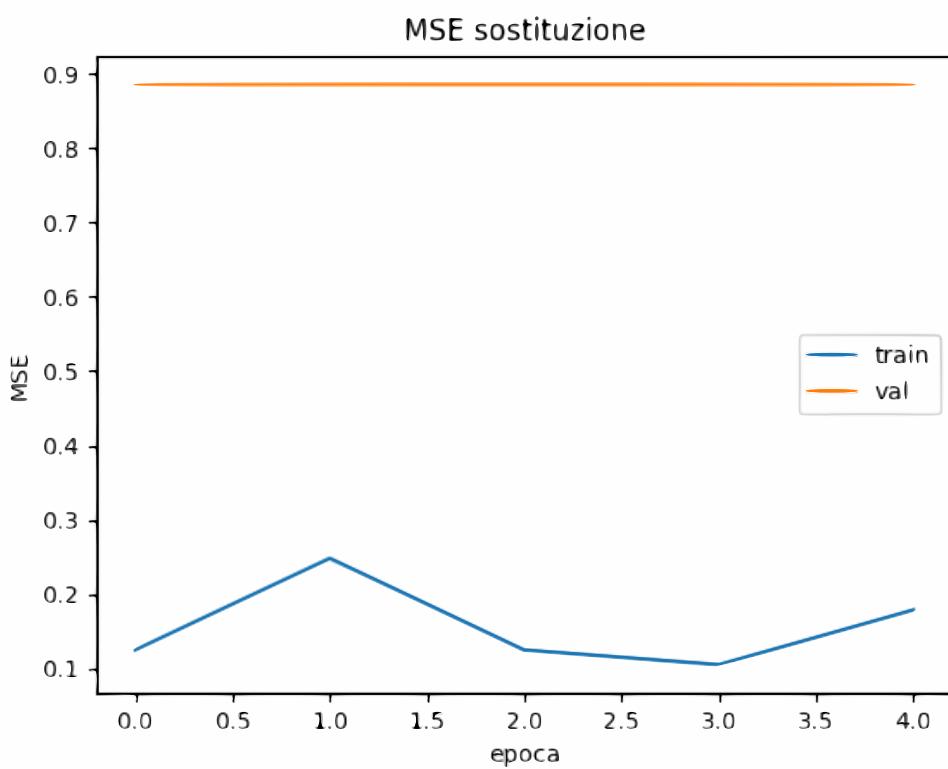
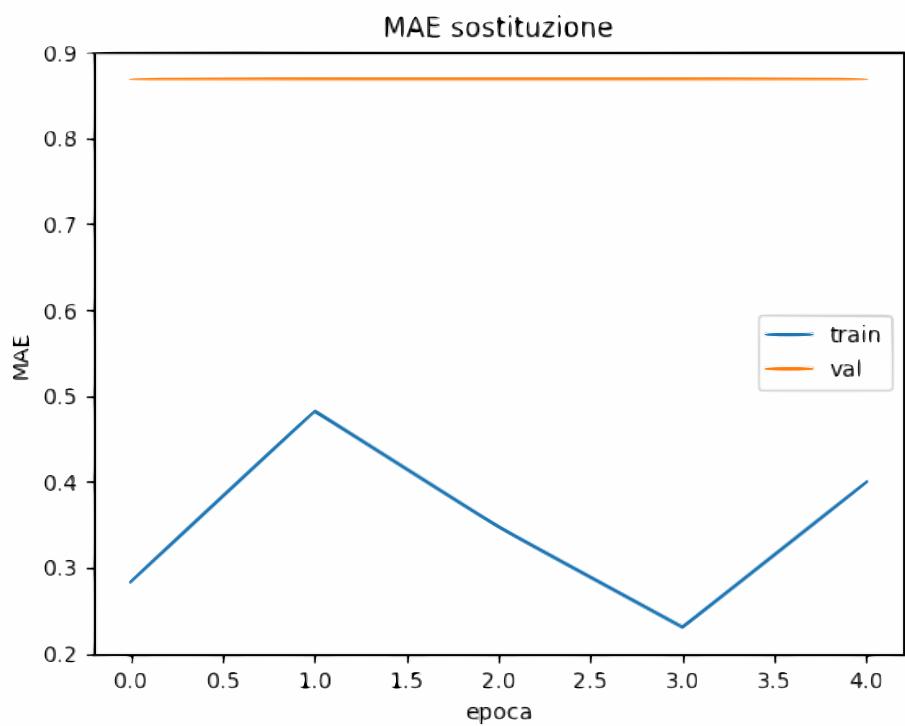
Di seguito, come già fatto per la parte del main core ma in numero ridotto, esporrò una serie di grafici, generati tramite le API di Tensorflow, inerenti ad alcune delle metriche in questione, in ordine cronologico, dopo vari tentativi e aggiustamento degli iperparametri del modello e quant'altro.











6. Deployment e integrazione in produzione

6.1 Architettura del servizio ReST Flask API ed end-points

Eccoci arrivati al terzo e ultimo macro-argomento di questa tesi di laurea... quello inerente all'integrazione finale, del software realizzato, in un server ReST autonomo e funzionante, interconnesso tramite microservizi, a tutto il resto dell'ecosistema digitale dell'infrastruttura Human Machine Interface (HMI) dell'azienda. Quindi, quest'ultima parte non riguarda un vero e proprio ulteriore sviluppo software basato su IA, bensì tratta la realizzazione di un'infrastruttura web che consente, all'algoritmo di IA creato, di interagire con il resto dei servizi aziendali tramite Application Program Interface (API).

Arrivati a questo punto, il software AI-based è stato completamente finalizzato e pronto per essere adattato - in prima linea - alla fase di lavoro su macchina vera e propria. Manca però ancora un componente importante, ovvero l'intersezione necessaria a fare in modo che, tale software, possa comunicare correttamente con la macchina su cui risiede e con i database aziendali inerenti ai vari clienti specifici. Pertanto, a lavoro concluso, ogni cliente distinto avrà la propria macchina funzionante col proprio software AI-based e la propria interfaccia ReST per consentirne il deployment effettivo.

Il servizio ReST implementato è stato organizzato come una piccola applicazione Flask che incapsula l'intero workflow di aggiornamento e inferenza real-time del modello predittivo. All'avvio vengono create - se non già esistenti - le directory per l'upload dei modelli .h5 importati e l'archivio JSON degli eventi di produzione e manutenzione. Inoltre, si definiscono due meccanismi di locking con mutex - uno in memoria centrale per i thread dentro lo stesso processo e uno su file per sincronizzare e gestire eventuali processi concorrenti - garantendo, con mutua esclusione, che non si sovrappongano insert dello stesso tipo o operazioni di fine-tuning contemporanee (solo un'operazione di fine-tuning alla volta può essere chiamata).

In testa al file sono state importate le funzioni di pre-processing, training e inferenza real-time dal modulo `fine_tuning.py`, si sono caricati i percorsi ai pesi (modello di default o fine-tuned) e si è impostato il logging a video per informazioni riguardo ad eventuali eccezioni, errori e processi terminati correttamente con esito positivo. Ogni end-point con protocollo HTTP ("GET /predictions", vari "POST" per import, export, append di eventi e avvio del fine-tuning) lavora sempre sotto lock, legge o aggiorna l'archivio `spoilboard_history.json` (dove sono contenuti i nuovi dati del cliente – ordinati per "Timestamp" - inerenti alle proprie abitudini operative), invoca le funzioni di merge data e di pre-processing o quelle di inferenza dinamica e infine restituisce in JSON il risultato della predizione attuale corrente o un codice di errore appropriato se essa non è disponibile o se si verifica un qualche tipo di errore. In questo modo, l'API offre sia servizi di CRUD sul database di eventi, sia vari punti di ingresso per scaricare o caricare i pesi del modello, sia la possibilità—attraverso "POST /fine_tune" di far partire, in sicurezza, il flusso di fine-tuning (full o regression-only) a seconda del numero di cicli completi già registrati secondo una costante threshold N. Nel seguito, verranno descritti, uno per uno, ciascun end-point... illustrandone scopi, parametri di ingresso e uscita, controlli di validità e concorrenza e formato dell'output e delle risposte finali.

POST /insert_prod

L'endpoint **/insert_prod** permette di inserire uno o più nuovi eventi produttivi, in un file di formato .json, all'interno dell'archivio cronologico senza interrompere il servizio né bloccare le richieste di inferenza. Il client invia tale **lista JSON** di record, ciascuno contenente almeno i campi:

- **Start**: timestamp ISO8601 di inizio produzione.
- **End**: timestamp ISO8601 di fine produzione.

Al momento della ricezione di tali record, si verificano alcune azioni:

1. **Parsing e validazione**: Il server richiama `request.get_json(force=True)` per ottenere la lista corrente; se il payload non è un array di oggetti o non contiene esattamente i campi obbligatori richiesti, il gateway risponde con **Error 400 Bad Request** e un log correlato di errore.
2. **Arricchimento**: Per ogni record valido, si imposta `EventType = 1` per marcare l'evento come "produzione". Dopodiché, come nel core di default, si copia il valore `End` in `Timestamp`, così da usare la fine del blocco come chiave di ordinamento.
3. **Locking e scrittura**: Tramite l'utilizzo di un thread-lock si garantisce l'atomicità e la mutua esclusione dell'operazione all'interno dello stesso processo. Per consentire tutto ciò, si utilizzano semafori mutex e tecniche di programmazione concorrente inerenti ai processi multithread.
4. **Risposta e output**: Una volta scritti con successo i dati, viene restituito un **200 OK** dal server come risposta e, in output, un JSON del tipo:

```
{  
    "message": "produzioni appended",  
    "size": 3  
}
```

Dove "size" è il numero totale di eventi registrati. Infine, per garantire un controllo intrinseco sulla possibile crescita esponenziale della dimensione del dataset, ho installato una logica di eliminazione dei record più vecchi finché, il dataset, non rientra nelle dimensioni massime consentite di 100 MB massimi. Di seguito, il codice esplicativo di tale end-point:

```
@app.route("/insert_prod", methods=["POST"])
def insert_prod():
    try:
        data = request.get_json(force=True)
    except:
        abort(400, "JSON non valido")
    if not isinstance(data, list):
        abort(400, "Serve un array di eventi JSON")
    # Ogni record deve avere almeno "Start" ed "End"
    to_append = []
    for rec in data:
```

```

        if "Start" not in rec or "End" not in rec:
            abort(400, "Ogni record produttivo deve avere 'Start' e 'End'")
        rec = rec.copy()
        rec["EventType"] = 1
        # usiamo End come timestamp per l'ordinamento
        rec["Timestamp"] = rec["End"]
        to_append.append(rec)

    with ARCHIVE_LOCK:
        arch = json.load(open(ARCHIVE_FILE))
        arch.extend(to_append)
        arch.sort(key=lambda e: e["Timestamp"])
        raw = json.dumps(arch).encode()
        while len(raw) > MAX_ARCHIVE_MB:
            arch.pop(0)
            raw = json.dumps(arch).encode()
        with open(ARCHIVE_FILE, "w") as f:
            json.dump(arch, f, indent=2)

    return jsonify({"message": "programmi appended", "size": len(arch)}), 200

```

POST /insert_manut

L'endpoint **/insert_manut** consente di aggiungere uno o più eventi di manutenzione, calcolandone automaticamente il numero di intercorse attività produttive e le ore di lavoro accumulate dall'ultimo intervento. Anche qui il client invia una **lista JSON** di oggetti, ognuno con:

- **Timestamp**: momento ISO8601 dell'intervento.
- **MaintenanceType**: intero (0 = sostituzione, 1 = rettifica).
- **WearValue**: misura dello stato di usura attuale al tempo dell'evento manutentivo correlato.

Anche qui, analogamente a /insert_prod, durante il ricevimento si svolge:

1. **Parsing e validazione**: Il payload viene convertito in lista; se non è un array o mancano campi obbligatori, il servizio risponde con **400 Bad Request**.
2. **Lettura dell'archivio**: In lettura sotto **ARCHIVE_LOCK**, il server carica l'intero elenco di eventi già presenti fino a tale momento.
3. **Costruzione delle liste di supporto**: Cicla su tutto l'archivio per separare **prod_list**, che è la lista di tutte le coppie per tutti gli eventi con EventType = 1 e **manut_list**, che è la lista dei Timestamp degli eventi manutentivi già registrati.

4. **Calcolo dei valori aggiuntivi:** Come nel main core, per ciascun nuovo record si:
 - a. Converte “Timestamp” in formato datetime.
 - b. Si trova l’ultima manutenzione precedente in manut_list e si filtra prod_list per includere solo le produzioni avvenute tra l’intervento precedente e quello successivo.
 - c. Si imposta TotalEvents al numero di produzioni in quell’intervallo e WorkingHours alla somma delle loro durate.
 - d. Si aggiunge il timestamp corrente a manut_list per calcolare correttamente, con la stessa logica, i record successivi.
5. **Locking e aggiornamento:** Con **ARCHIVE_LOCK** si carica di nuovo tutto l’archivio (eventuale aggiornamento concorrente), vi si appendono i nuovi record espansi, si ordinano per Timestamp, e si tronca l’inizio dell’archivio stesso se supera la dimensione massima consentita. Infine, si salva il JSON.
6. **Risposta e output:** Analogamente al POST /insert_prod, anche qui una volta scritti con successo i dati, viene restituito un **200 OK** dal server come risposta e, in output, un JSON del tipo:

```
{
    "message": "manutenzioni appended",
    "size": 5
}
```

Questi due end-point, pur trattandosi di operazioni di semplice **append** su file JSON, assicurano coerenza e consistenza anche in presenza di richieste concorrenti, e svolgono un ruolo cruciale nel mantenere aggiornato lo storico da cui dipendono inferenza e fine--tuning online “real-time”. Di seguito, il codice aggiornato del **POST /insert_manut**.

```
# -----
# 4b) POST /insert_manut - aggiungi eventi manutentivi
# -----
@app.route("/insert_manut", methods=["POST"])
def insert_manut():
    try:
        data = request.get_json(force=True)
    except:
        abort(400, "JSON non valido")
    if not isinstance(data, list):
        abort(400, "Serve un array di eventi JSON")

    # -----
    # 1) Leggo l'archivio esistente
    # -----
    with ARCHIVE_LOCK:
        arch = json.load(open(ARCHIVE_FILE))
```

```

# 2) Preparo due liste da arch:
#     - prod_list: elenco (timestamp_datetime, durata_in_ore) per ogni
produzione già registrata
#     - manut_list: elenco di tutti i datetime di manutenzioni già registrate
prod_list = []
manut_list = []
for e in arch:
    ts = datetime.fromisoformat(e["Timestamp"])
    if e.get("EventType") == 1:
        # produciamo la durata in ore = (End - Start)
        start_dt = datetime.fromisoformat(e["Start"])
        end_dt   = datetime.fromisoformat(e["End"])
        dur_h    = (end_dt - start_dt).total_seconds() / 3600.0
        prod_list.append((ts, dur_h))
    else:
        manut_list.append(ts)

to_append = []

# _____
# 3) Per ciascun record manutentivo in ingresso:
# _____
for rec in data:
    # a) controlli base
    if "Timestamp" not in rec or "MaintenanceType" not in rec or "WearValue" not in rec:
        abort(400, "Ogni record manutentivo deve avere 'Timestamp', 'MaintenanceType', 'WearValue'")

    nuovo = rec.copy()
    nuovo[ "EventType" ] = 0

    # b) converto in datetime
    try:
        ts_manut = datetime.fromisoformat(nuovo[ "Timestamp" ])
    except:
        abort(400, f"Timestamp non valido: {nuovo['Timestamp']}")

    # c) trovo l'ultima manutenzione precedente (se esiste)
    prev_manut = None
    if manut_list:
        # prendo la manutenzione massima < ts_manut
        precedenti = [t for t in manut_list if t < ts_manut]

```

```

        if precedenti:
            prev_manut = max(precedenti)

        # d) filtro le produzioni cadute in (prev_manut, ts_manut]
        if prev_manut is not None:
            prods_intervallo = [ (t, d) for (t, d) in prod_list if prev_manut < t
<= ts_manut ]
        else:
            # se non c'è manutenzione precedente, prendo tutte le produzioni fino a
            ts_manut
            prods_intervallo = [ (t, d) for (t, d) in prod_list if t <= ts_manut ]

        # e) TotalEvents = numero di produzioni in quell'intervallo
        nuovo[ "TotalEvents" ] = len(prods_intervallo)

        # f) WorkingHours = somma delle durate in ore di quelle produzioni
        nuovo[ "WorkingHours" ] = round(sum(d for (t, d) in prods_intervallo), 2)

        # g) aggiungo ts_manut a manut_list così che la prossima manutenzione lo
        veda come "precedente"
        manut_list.append(ts_manut)
        to_append.append(nuovo)

# _____
# 4) Riscrivo l'archivio aggiornato (ordinandolo per Timestamp)
# _____
with ARCHIVE_LOCK:
    arch = json.load(open(ARCHIVE_FILE))
    arch.extend(to_append)
    arch.sort(key=lambda e: e[ "Timestamp" ])
    raw = json.dumps(arch).encode()
    while len(raw) > MAX_ARCHIVE_MB:
        arch.pop(0)
        raw = json.dumps(arch).encode()
    with open(ARCHIVE_FILE, "w") as f:
        json.dump(arch, f, indent=2)

return jsonify({ "message": "manutenzioni appended", "size": len(arch)}), 200

```

POST /import

Questo endpoint accetta un file di pesi .h5 caricato dal client e lo sostituisce ai pesi attualmente installati sul software per l'inferenza e il fine-tuning. Vengo effettuati vari passaggi:

1. **Ricezione del file multipart:** Si aspetta in ingresso un campo model di tipo file, all'interno di una form-data structure. Se il campo non c'è o il filename è vuota, si restituisce l'errore **400 Bad Request** con messaggio di log esplicativo, altrimenti si restituisce **200 OK**.
2. **Sanitizzazione del nome del file:** Il nome originale del file viene passato alla procedura **secure_filename** per rimuovere caratteri potenzialmente problematici o path traversal.
3. **Salvataggio e copia atomica:** Il file viene dapprima salvato in una directory temporanea "uploads/" posizionata dentro la folder principale del progetto; quindi, viene copiato sovrascrivendo sul percorso di destinazione attuale e rinominato in: **last_fine_tunedmodel.weights.h5**.
4. **Risposta:** Al completamento con successo, l'API risponde con 200 OK e con un JSON di output del tipo:

```
{  
    "message": "Pesi importati correttamente in  
    last_fine_tunedmodel.weights.h5"  
}
```

Confermando che i nuovi pesi verranno usati per tutte le chiamate successive di inferenza e fine-tuning, finché non verrà caricato un nuovo modello di pesi che sovrascriverà, a sua volta, quello esistente.

Il codice definitivo:

```
# -----  
# 3) POST /import - carica nuovi pesi  
# -----  
@app.route("/import", methods=["POST"])  
def import_weights():  
    if 'model' not in request.files:  
        abort(400, "Devi inviare un file nel campo 'model'")  
    f = request.files['model']  
    if f.filename == '':
```

```

        abort(400, "Filename non valido")
filename = secure_filename(f.filename)
tmp_path = os.path.join(UPLOAD_DIR, filename)
f.save(tmp_path)
shutil.copy(tmp_path, FINE_TUNED_WEIGHTS)
return jsonify({"message": f"Pesi importati correttamente in {FINE_TUNED_WEIGHTS}"}), 200

```

POST /export

Questo end-point espone i pesi correnti del modello in formato .h5, così che il client possa scaricarli e conservarli o versionarli in archivio. Le azioni eseguite da questo end-point sono principalmente tre:

- Selezione del file di pesi:** Se esiste il file dei pesi del modello fine-tuned (**last_fine_tunedmodel.weights.h5**), viene scelto quello per il caricamento, altrimenti si utilizza il modello di default addestrato nel main core: **best_RNNPianoMartire.weights.h5** e si carica nella procedura.
- Controllo di esistenza:** Se il file scelto non è presente sul disco, il servizio risponde con **404 Not Found**, indicando che non c'è alcun modello da esportare.
- Lettura e streaming:** Il file, mentre è caricato, viene letto in modalità binaria e restituito come **attachment object** con codice di avvenuto successo: **200 OK**. Inoltre, tramite l'header: **Content-Disposition: attachment; filename="exported_model.weights.h5"** e il MIME type: **application/octet-stream**, il browser corrente o un client programmatico riesce ad innescare la procedura di download del file corrente secondo il nome suggerito.

Il codice della procedura:

```

# -----
# 2) POST /export - scarica exported_model.weights.h5
# -----
@app.route("/export", methods=["POST"])
def export_weights():
    # scegli i pesi (fine-tuned o base)
    source = FINE_TUNED_WEIGHTS if os.path.exists(FINE_TUNED_WEIGHTS) else
BASE_WEIGHTS
    if not os.path.exists(source):
        abort(404, f"Weights non trovati: {source}")
    # legge il file in memoria
    with open(source, 'rb') as f:

```

```

        data = f.read()
    # restituisce un attachment
    return Response(
        data,
        mimetype='application/octet-stream',
        headers={
            'Content-Disposition': 'attachment;
filename="exported_model.weights.h5"'
        }
    )

```

POST /fine_tune

L'end-point **/fine_tune** è il vero e proprio innesco della procedura di fine-tuning real-time. Essa decide, autonomamente, se eseguire **fine-tuning full** o la modalità **regression-only**, in base al numero di cicli completi già registrati. Anch'esso svolge diverse azioni:

- Controllo di concorrenza:** Azione fondamentale, infatti non può assolutamente essere eseguito più di un fine-tuning per volta, perché non avrebbe senso e sovraccaricherebbe il server inutilmente. Nella pratica quindi, si tenta di acquisire un **thread-lock** per poter processare correttamente la procedura di fine-tuning in maniera mutuamente esclusiva. Dopodiché, se l'azione è già in fase di processamento (e quindi il fine-tuning è già stato avviato), allora il server risponde subito con il log **429 Too Many Requests - “Fine-tuning già in esecuzione (thread lock)”**. In seguito, si crea (o si apre, se esiste già) un file di lock – `fine_tune.lock` - su disco e si prova a prenderne un **file-lock** non bloccante tramite la funzionalità di Python: **portalocker**. Se essa fallisce, significa che un altro processo sta già svolgendo l'operazione di fine-tuning e, pertanto, il server ritorna 429 con “Fine-tuning in corso (file lock)” come detto precedentemente.
- Preparazione dei dati:** Sotto il file di archive lock si legge l'archivio JSON completo di tutti gli eventi incrementali di produzione e manutenzione, ordinati tramite “Timestamp”. Se esso è vuoto, si risponde con il codice di errore **400 Bad Request**, invitando il client a caricare, in primo luogo, qualche evento valido con i POST di `/insert_prod` e `/insert_manut`. Per verificare, invece, quanti cicli completi ci sono e, quindi, eventualmente chiamare la funzione di fine-tuning full e non solo regression-only, si utilizza la funzione ausiliaria **count_complete_cycles()**:

```

def count_complete_cycles(events):
    df = tune.merge_data_list(events)
    for fn in ("missing_values", "compute_counters", "encoding",
               "segment_cycles", "segment_subcycles"):
        df = getattr(tune.core, fn)(df)
    complete = df[(df.EventType==0)&(df.MaintenanceType==0)]
    return complete.cycle_id.nunique()

```

In questo modo, si riesce a tenere traccia – in maniera complessiva – di quanti cicli completi si registrano prima di fare un raffinamento totale su tutti i dati a disposizione e, pertanto, si conosce esattamente qual è il momento opportuno per lanciare un fine-tuning full.

3. **Dump temporaneo a scelta modalità:** L'intero archivio storico viene scritto in un file JSON temporaneo. Come illustrato precedentemente, se il numero di cicli completi è maggiore o uguale a N, si invoca `fine_tune_full(tmp_path)`, altrimenti si usa `fine_tune_regression_only(tmp_path)`.
4. **Esecuzione del fine-tuning:** Si esegue la procedura, divisa nei due casi principali:
 - a. **Fine_tune_full:** aggiorna sia il ramo di regressione sia quello di classificazione, utilizzando la pipeline completa implementata e avvia l'addestramento del modello con stratified split, callback, ricomputazione della soglia ecc.
 - b. **Fine_tune_regression_only:** esegue invece l'aggiornamento in tempo reale del solo branch regressivo, batch-by-batch, con blocco dei layer classificativi perché sono irrilevanti in questo contesto.
5. **Rilascio dei lock:** Al termine o in caso di eccezione sollevata, il file-lock viene rilasciato e il file temporaneo, che lo contiene, eliminato. Dopodiché, si libera il thread-lock e si può proseguire con la richiesta successiva.
6. **Risposta:** Se il training va a buon fine, si restituisce **200 OK** con un JSON che include l'esito finale di questo fine-tuning, il numero di cicli completi che sono stati utilizzati e registrati, la matrice di confusione, il classification report e un riassunto sintetico dei passi e delle metriche utilizzati dal modello durante il fine-tuning. Tutto ciò serve per informare il client di quale strategia è stata usata, quanti cicli erano disponibili, quanto è lunga la storia delle metriche raccolte e, nel caso del full fine-tuning, capire bene la distribuzione delle classi e il loro riconoscimento effettivo nella fase di test.
7. **Gestione degli errori:** Qualunque eccezione che - durante il processo - possa venire intercettata, viene poi loggata con stack trace e restituita al client con errore **500 Internal Server Error** e un campo aggiuntivo “Details”, che contiene il messaggio dell'errore stesso per intero.

Così facendo, il POST /fine_tune assicura che non più di un processo o thread in parallelo possa aggiornare, simultaneamente, i pesi del modello fine-tuned. Questo è molto importante per evitare ogni tipo di sovrapposizione non desiderata e aggiornamenti in parallelo che aggiornano male i pesi stessi. Di seguito, il codice per intero di tale end-point:

```
@app.route("/fine_tune", methods=["POST"])
def finetune():
    # 0) lock di thread – se un altro thread è già in corso, 429
    if not FINETUNE_LOCK.acquire(blocking=False):
        return jsonify({
            "error": "Fine-tuning in corso",
```

```

        "details": "thread lock"
    }), 429

try:
    # 1) lock di file (fra processi)
    try:
        lock_fd = open(LOCK_FILE, "a+")
        portalocker.lock(lock_fd, portalocker.LOCK_EX | portalocker.LOCK_NB)
    except portalocker.LockException:
        return jsonify({
            "error": "Fine-tuning in corso",
            "details": "file lock"
        }), 429

    try:
        # Conta i cicli e scegli la modalità
        with ARCHIVE_LOCK:
            archive = json.load(open(ARCHIVE_FILE))
        if not archive:
            return jsonify({
                "error": "Archivio vuoto",
                "details": "Prima inserisci qualche evento con /insert_prod o
/insert_manut"
            }), 400

        n = count_complete_cycles(archive)

        with tempfile.NamedTemporaryFile("w+", suffix=".json", delete=False) as tmp:
            json.dump(archive, tmp)
            tmp_path = tmp.name

        if n >= MIN_CYCLES_FULL:
            model, history, cm, cr = tune.fine_tune_full(tmp_path)
            mode = "full"
        else:
            model, history      = tune.fine_tune_regression_only(tmp_path)
            mode = "regression-only"

        # Prepara la risposta
        out = {
            "mode": mode,
            "n_complete_cycles": n,
            "history_length": {k: len(v) for k, v in history.items()}
        }
    
```

```
        }

    if mode == "full":
        out["confusion_matrix"]      = cm.tolist()
        out["classification_report"] = cr

    return jsonify(out), 200

except Exception as e:
    # Log completo
    logger.exception("Errore durante fine_tune")
    # Rispondi in JSON col dettaglio
    return jsonify({
        "error": "Fine-tuning fallito",
        "details": str(e)
    }), 500

finally:
    # Rilascia il file-lock
    try:    portalocker.unlock(lock_fd)
    except: pass
    lock_fd.close()
    try:    os.remove(LOCK_FILE)
    except FileNotFoundError: pass

finally:
    FINETUNE_LOCK.release()
```

GET /predictions

Passiamo, dunque, all'ultimo end-point del server ReST, nonché quello più esplicativo di tutti, poiché è il vero e proprio illustratore del risultato finale, generato da tutto il sistema messo in piedi. L'endpoint **/predictions** fornisce all'utente l'ultima previsione generata dal modello, integrando in un'unica chiamata tutte le fasi di lettura dello storico, inferenza dinamica e formattazione del risultato in JSON. Di seguito, l'illustrazione del procedimento e del funzionamento, passo dopo passo:

1. **Blocco sull'archivio JSON di eventi:** Prima di tutto, il server acquisisce il lock di archivio, ovvero un semaforo “mutex” che impedisce ad altri thread contemporanei, dello stesso processo, di modificare, simultaneamente, il file buffer di archivio **spoilboard_history.json**. Questo passaggio è cruciale, perché è necessario garantire che la lettura avvenga sempre su uno snapshot coerente di dati attuali, senza il rischio di corruzione o di mancata sincronizzazione con operazioni di inserimento concorrenti e contemporanee.
2. **Lettura e validazione del JSON:** Una volta ottenuto il lock, il servizio apre e deserializza il file degli eventi. Se il contenuto del file non è una lista JSON, oppure se per qualche ragione il parsing fallisce, l'end-point restituisce un **500 Internal Server Error** con un messaggio che segnala che l'archivio dei dati è corrotto e non disponibile per il processing.
3. **Selezione dei pesi del modello:** Prima di eseguire qualunque azione di inferenza, il server decide quali pesi caricare per effettuare tale inferenza a partire da quei pesi stessi. Infatti, se esiste il file `last_fine_tunedmodel.weights.h5`, si ritiene il modello già fine-tunato con pesi aggiornati e, quindi, lo si carica. Altrimenti, si ricade sul modello di default importato dalla procedura iniziale del core: `best_RNNPianoMartire.weights.h5`. Se non vengono trovati nessuno dei due, il server risponde con codice di errore 500, segnalando che attualmente nessun modello è disponibile per effettuare inferenza dinamica real-time.
4. **Chiamata “real-time” all’inferenza dinamica:** La parte di inferenza dinamica vera e propria che, tramite i dati grezzi e il modello pronto, invoca una funzione ulteriore: **dynamic_predict_from_events(events)** per ottenere le previsioni in tempo reale, sull’ultimo timestep dell’ultimo ciclo di vita attualmente in memoria. Infatti, questa routine:
 - a. Ricostruisce il DataFrame di produzione e di manutenzione in real-time dai dati attualmente disponibili.
 - b. Genera le sequenze temporali di features e target per l’ultimo ciclo di vita presente.
 - c. Invoca il modello attualmente caricato su ciascuna sequenza, applicando la logica di “soft-reset” - un’inferenza dinamica in cui i valori regressivi vengono diluiti con i valori storici a mano a mano che la produzione prosegue, adattandosi così incrementalmente e gradualmente alle nuove abitudini del cliente – sui contatori e producendo una timeline completa di previsioni, per ogni timestep, come succede nel main core.
 - d. Restituisce una lista, ordinata per “Timestamp”, di dizionari o tuple, ognuno contenente tutti i campi di interesse (ciclo, sottociclo, timestep, remaining_rett, remaining_sost ecc.) per la restituzione, da parte della GET, al server.

Chiaramente, anche in questo contesto, qualunque eccezione sollevata provoca un errore con codice 500 e un log di inferenza fallita comprendente il dettaglio dell'errore stesso. Di seguito, la funzione, per intero, di **dynamic_predict_from_events**:

```
def dynamic_predict_from_events(archive_events: list) -> list:
    # 1) Build & load weights
    model = core.modelling_combined(num_features=len(core.features), units=128)
    weights_to_load = FINE_TUNED_WEIGHTS_OUT if
os.path.exists(FINE_TUNED_WEIGHTS_OUT) \
                    else BASE_WEIGHTS_PATH
    model.load_weights(weights_to_load)

    # 2) Merge & preprocess
    df = merge_data_list(archive_events)
    df = core.missing_values(df)
    df = core.compute_counters(df)
    df = core.encoding(df)
    df = core.segment_cycles(df)
    df = core.segment_subcycles(df)
    df_pre = core.prepare_targets_combined(df)
    df_pre["Timestamp"] = df_pre["Timestamp"].astype(np.int64) // 10**9

    exp_cycle, exp_sub = core.compute_expected_values(df_pre, dedupe=True)

    # 4) Avg hours (VERSIONE AGGIORNATA CHE USA DEFAULT STORICO SE HAI SOLO 0 0 1
RECORD VALIDO)
    m = df_pre[df_pre['EventType'] == 0]

    # 4a) Calcolo i default “storici” da core (potrebbero essere None)
    default_rett = core.calcola_default_avg_hours(m, maintenance_type=1,
                                                    col_time="T_Rettifica",
                                                    col_events="TotalEvents")
    default_sost = core.calcola_default_avg_hours(m, maintenance_type=0,
                                                    col_time="T_Sostituzione",
                                                    col_events="CumulativeEvents")

    # 4b) Se il default è None o <= 0, forziamo un piccolo fallback (ad es. 0.1
ore)
    default_rett = default_rett if (default_rett is not None and default_rett >
0) else 0.1
    default_sost = default_sost if (default_sost is not None and default_sost >
0) else 0.1
```

```

# --- Per le RETTIFICHE: prendo solo i record con WorkingHours>0 e
MaintenanceType==1
vr = m[(m['MaintenanceType'] == 1) & (m['WorkingHours'] > 0)]
if len(vr) >= 2:
    # se ho ≥2 record validi, media di (T_Rettifica / TotalEvents)
    avg_rett = (vr['WorkingHours'] / vr['TotalEvents']).mean()
elif len(vr) == 1:
    # solo 1 record => uso il default storico
    avg_rett = default_rett
else:
    # zero record validi => uso comunque il default storico
    avg_rett = default_rett

# --- Per le SOSTITUZIONI: prendo solo i record con WorkingHours>0 e
MaintenanceType==0
vs = m[(m['MaintenanceType'] == 0) & (m['WorkingHours'] > 0)]
if len(vs) >= 2:
    # se ho ≥2 record validi, media di (T_Sostituzione / CumulativeEvents)
    avg_sost = (vs['WorkingHours'] / vs['CumulativeEvents']).mean()
elif len(vs) == 1:
    # un solo record sostituzione => uso il default storico
    avg_sost = default_sost
else:
    # zero record validi => uso comunque il default storico
    avg_sost = default_sost
print(f"[DEBUG] avg_hours_rett = {avg_rett:.3f}, avg_hours_sost =
{avg_sost:.3f}")

# 5) Scaling
scaler = load_scaler()
numeric_cols = [
    "CumulativeEvents", "SubCycle_Events", "TotalEvents",
    "WorkingHours", "WearValue",
    "Tot_Events_Rett", "Tot_Events_Sost",
    "T_Rettifica", "T_Sostituzione"
]
scaler.feature_names_in_ = np.array(numeric_cols, dtype=object)
df_pre[numeric_cols] = scaler.transform(df_pre[numeric_cols])

# 6) Creazione sequenze solo per l'ultimo ciclo
last_cycle = df_pre['cycle_id'].max()
df_last = df_pre[df_pre['cycle_id'] == last_cycle]

```

```

sx, sy, metas = core.createRepresentativeSequences(df_last, core.features,
core.target, factor=1.0)
sx, sy, metas = core.filterValid(sx, sy, metas)

# 7) Dynamic predict (usa la versione con "soft reset" già definita)
preds = dynamic_predict_full_sequence(
    model,
    sx,
    [int(y[-1, 2]) for y in sy],
    true_sub_total      = exp_sub,
    true_cycle_total   = exp_cycle,
    avg_hours_rett     = avg_rett,
    avg_hours_sost     = avg_sost,
    expected_subcycle_value = exp_sub,
    expected_cycle_value = exp_cycle,
    threshold          = load_threshold(),
    metas              = metas
)
logging.info(f"→ Predizione attuale salvata in {PREDICTIONS_OUT}")
return preds

```

5. **Estrazione dell'ultima predizione:** Dalla lista JSON restituita, si prende solo l'**ultimo elemento** per fornire la rappresentazione della predizione del timestep corrente. Esso indica lo stato corrente dell'ultimo timestep eseguito, indipendentemente che sia un'operazione di produzione o di manutenzione. Se l'elemento è una tupla, il server lo trasforma autonomamente in un dizionario, mappando ciascun indice al corrispondente campo di output; se invece, è già un dizionario, lo wrappa e lo usa direttamente. Se, alternativamente, il formato è inaspettato, il server solleva un'eccezione e restituisce codice errore 500.
6. **Formato della predizione:** Prima di inviare qualunque risposta, si formatta la predizione corrente in JSON “pretty-printed” style. In questo modo, il contenuto del JSON viene impilato in colonna e indentato correttamente. Di seguito, un esempio pratico di come viene restituito il JSON tramite la GET, su un API Flask di debug:

Flask API Test

GET /predictions POST /fine_tune POST /import (.h5) POST /export POST /insert_prod (.json)

POST /insert_manut (.json)

```
200 OK
{
  "cycle": 1,
  "subcycle": 1,
  "seq_id": 2,
  "time_step": 5,
  "remaining_rett": 99,
  "remaining_sost": 13579,
  "predicted_days_rett": 2,
  "predicted_days_sost": 57,
  "event_type": "MANUTENZIONE",
  "next_event": "RETTIFICA",
  "next_maint_target": 1
}
```

7. **Persistenza della predizione:** In modo trasparente per il client, il server salva questo dizionario JSON su un file persistente: **predictions.json**, in modo che sia sempre disponibile all'esterno del processo e possa essere eventualmente esportato o monitorato da altri strumenti dell'infrastruttura generale.
8. **Risposta finale al client:** Infine, l'end-point risponde con **200 OK** e restituisce il JSON “pretty-printed” discusso precedentemente, inherente all'ultimo stato predetto. Grazie a questo meccanismo, il client può chiamare, ripetutamente, la GET /predictions per ottenere, real-time, lo stato attuale del modello predittivo, basandosi su uno storico sempre aggiornato agli ultimi eventi effettuati, con tutti i controlli di errori e di concorrenza necessari in un servizio di produzione di streaming dati. Se per caso la GET venisse chiamata, mentre è già stata chiamata una procedura di fine_tune che non è ancora terminata, le predizioni restituite saranno inerenti ai pesi del modello aggiornati al momento antecedente a tale fine_tune appena lanciato, ma il server non si bloccherà... i due processi possono coesistere insieme, in modo concorrente, con questo vincolo appena esplicitato.

Così facendo, GET /predictions fornisce un'interfaccia estremamente semplice e atomicamente consistente per leggere lo stato in tempo reale del modello predittivo, basandosi su uno storico sempre aggiornato dinamicamente e robusto, grazie a tutti i controlli di errori, eccezioni e di concorrenza necessari a mantenere granulare l'efficienza del deployment in produzione del sistema. Di seguito, la funzione per intero di questo ultimo end-point:

```
# -----
# 1) GET /predictions - ultima predizione
# -----
@app.route("/predictions", methods=["GET"])
def get_predictions():
    # 1) Leggo eventi sotto lock
    with ARCHIVE_LOCK:
```

```

events = json.load(open(ARCHIVE_FILE))
if not isinstance(events, list):
    abort(500, "Archivio eventi corrotto")

# 2) Carico i pesi più aggiornati
weights_file = FINE_TUNED_WEIGHTS if os.path.exists(FINE_TUNED_WEIGHTS) else
BASE_WEIGHTS
if not os.path.exists(weights_file):
    abort(500, f"Nessun modello disponibile ({weights_file})")

# 3) Chiamo dynamic_predict_from_events → lista di dict
try:
    preds = tune.dynamic_predict_from_events(events)
except Exception as e:
    abort(500, f"Inferenza fallita: {e}")

if not isinstance(preds, list) or len(preds) == 0:
    abort(500, "dynamic_predict_from_events non ha restituito un risultato
valido")

# 4) Prendo ultima predizione
raw = preds[-1]
# Se raw è tuple, trasformo in dict
if isinstance(raw, tuple):
    if len(raw) != 11:
        abort(500, "Formato tupla inatteso da dynamic_predict_from_events")
    last_pred = {
        "cycle": raw[0],
        "subcycle": raw[1],
        "seq_id": raw[2],
        "time_step": raw[3],
        "remaining_rett": raw[4],
        "remaining_sost": raw[5],
        "predicted_days_rett": raw[6],
        "predicted_days_sost": raw[7],
        "class_probability": raw[8],
        "event_type": raw[9],
        "next_event": raw[10]
    }
elif isinstance(raw, dict):
    last_pred = raw
else:
    abort(500, "dynamic_predict_from_events ha restituito tipo non supportato")

```

```

last_pred.pop("class_probability", None)
# 5) Salvo su disco
with open(PREDICTIONS_FILE, "w") as f:
    json.dump(last_pred, f, indent=2)

# 6) Ritorno JSON al client
pretty = json.dumps(last_pred, indent=2, ensure_ascii=False)
return Response(pretty, mimetype="application/json"), 200

```

Per quanto riguarda, invece, l'implementazione grafica riassuntiva della pagina web di debug mostrata precedentemente e dei pulsanti di trigger degli end-point registrati, ho utilizzato un metodo GET con una return basato su codice HTML:

```

# .
# 0) INDEX: pagina di test
# .

@app.route("/", methods=[ "GET"])
def index():
    return """
<!DOCTYPE html>
<html lang="it">
<head><meta charset="UTF-8">
    <title>Flask API Test</title>
    <style>
        button { margin:8px; padding:12px 16px; border:none; cursor:pointer; font-size:14px; }
        .get-btn  { background:green; color:#fff; }
        .post-btn { background:#3498db; color:#fff; }
        #output { margin-top:20px; white-space:pre-wrap; border:1px solid #ccc; padding:12px; }
    </style>
</head>
<body>
    <h1>Flask API Test</h1>
    <button class="get-btn" onclick="callEndpoint('GET','/predictions')>GET /predictions</button>
    <button class="post-btn" onclick="callEndpoint('POST','/fine_tune')>POST /fine_tune</button>
    <button class="post-btn" onclick="uploadModel()">POST /import (.h5)</button>
    <button class="post-btn" onclick="downloadWeights()">POST /export</button>

```

```

<button class="post-btn" onclick="uploadProd()">POST /insert_prod(.json)</button>
<button class="post-btn" onclick="uploadManut()">POST /insert_manut(.json)</button>
<div id="output">Logs:</div>
<input type="file" id="modelFile" accept=".h5" style="display:none"/>
<input type="file" id="eventsFile" accept=".json" style="display:none"/>
<script>
    function callEndpoint(method, url, body) {
        const opts={ method };
        if(method==='POST' && body){
            opts.headers={'Content-Type':'application/json'};
            opts.body=body;
        }
        fetch(url, opts)
            .then(async r=>{
                const t=await r.text();
                document.getElementById('output').textContent = r.statusText+
'+r.statusText+'\n'+t;
            }).catch(e=>{
                document.getElementById('output').textContent='Error: '+e;
            });
    }
    function uploadModel(){
        const inp=document.getElementById('modelFile');
        inp.onchange=()=>{
            const f=inp.files[0];
            const fd=new FormData();
            fd.append('model',f);
            fetch('/import',{method:'POST',body:fd})
                .then(r=>r.json()).then(j=>{
                    document.getElementById('output').textContent =
JSON.stringify(j,null,2);
                });
        };
        inp.click();
    }
    function downloadWeights(){
        fetch('/export',{method:'POST'})
            .then(r=>{ if(!r.ok) throw new Error(r.statusText); return r.blob(); })
            .then(blob=>{
                const url=URL.createObjectURL(blob);
                const a=document.createElement('a');
                a.href=url; a.download='exported_model.weights.h5';
            });
    }
</script>

```

```

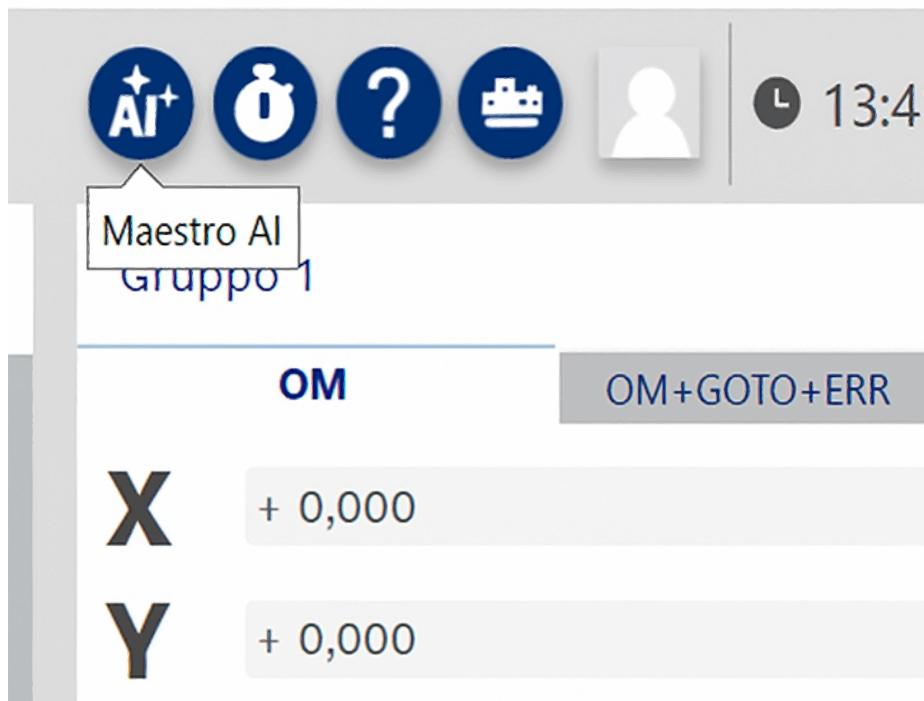
        document.body.appendChild(a);
        a.click();
        a.remove();
        URL.revokeObjectURL(url);
    })
    .catch(e=>document.getElementById('output').textContent='Error: '+e);
}
function uploadProd(){
    const inp=document.getElementById('eventsFile');
    inp.onchange=()=>{
        const reader=new FileReader();
        reader.onload=()=>callEndpoint('POST','/insert_prod',reader.result);
        reader.readAsText(inp.files[0]);
    };
    inp.click();
}
function uploadManut(){
    const inp=document.getElementById('eventsFile');
    inp.onchange=()=>{
        const reader=new FileReader();
        reader.onload=()=>callEndpoint('POST','/insert_manut',reader.result);
        reader.readAsText(inp.files[0]);
    };
    inp.click();
}
</script>
</body>
</html>
"""

```

Con questa infrastruttura web, sono riuscito ad esprimere graficamente il concetto di API ReST e punto di interfaccia software tra client e utente. È importante sottolineare che, questa pagina web originata, è stata utilizzata solo in fase di debug poiché, nella pratica effettiva, il servizio creato è stato direttamente installato sul software gestionale delle macchine stesse.

6.2 Implementazione su Maestro Active: Maestro AI

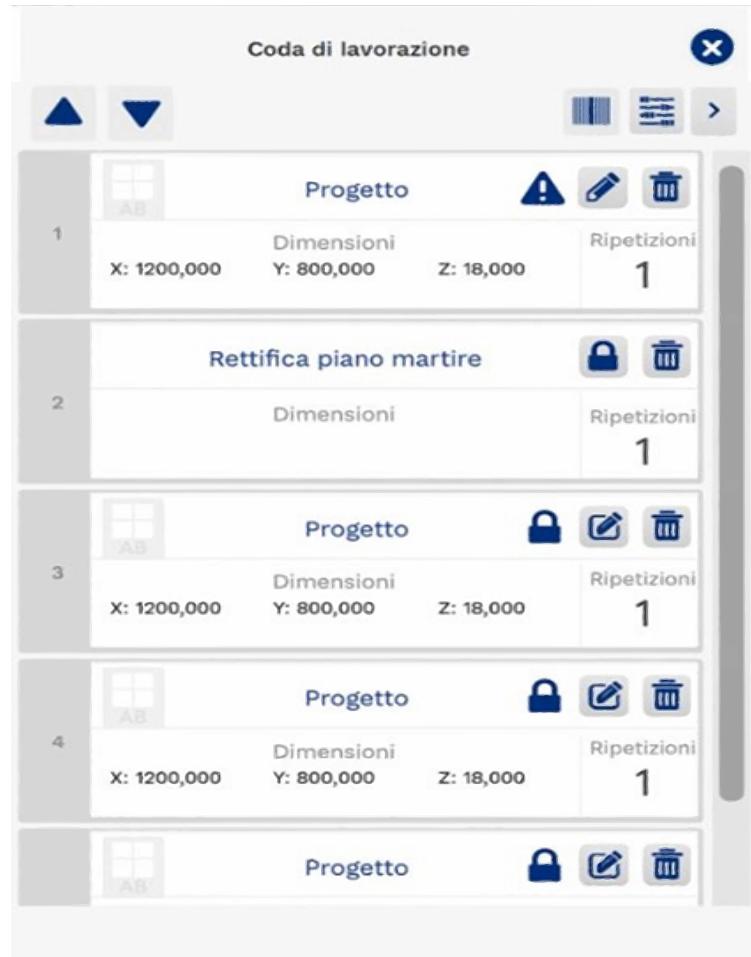
L'ultimo passaggio - a cui non ho dato direttamente il mio contributo da tirocinante a causa delle scadenze imminenti - è stato quello di implementare tutto questo sistema sul software gestionale della macchina CNC in questione: **Maestro Active**. Non avendolo sviluppato io in prima persona, spiegherò questo step finale a grandi linee, per fare comprendere nitidamente, l'implicazione logica e reale del software da me realizzato e come esso viene integrato nella rete aziendale vera e propria. **Maestro AI** è, di fatto, il nome del tool aggiuntivo da installare sul programma - già esistente - di gestione del piano martire. Questo offre una funzionalità avanzata di manutenzione preidittiva in tempo reale, che si adopera del JSON restituito dalla chiamata GET /predictions per inviare, al cliente in questione, una serie di notifiche riguardanti lo stato di usura raggiunto dal piano in tempo reale, tra quanti giorni circa sarà prevista un'operazione di manutenzione relativa alla classificazione eseguita dal modello e quanti programmi di lavorazione saranno ancora effettuabili prima di quel momento. Naturalmente, è una funzionalità attivabile e disattivabile con un semplice interruttore binario, così che possa essere utilizzata solamente da chi ne vuole trarre beneficio. Dal momento che essa viene attivata, il software AI-Based inizierà piano piano a registrare i vari nuovi record di manutenzione e produzione, nell'archivio JSON, tramite i POST /insert_prod e /insert_manut, azionati direttamente dal client. Ogni qualvolta verrà chiamata la procedura di POST /fine_tune, i pesi del modello attuale verranno aggiornati e fine-tunati e la GET /predictions effettuata, rispecchierà in tempo reale l'andamento produttivo e manutentivo attuale, basato sui pesi del modello appositamente aggiornati. Di seguito, due screenshot - in formato .emf - dell'interfaccia grafica dell'icona del software implementata direttamente su Maestro Active e del risultato della GET /predictions in ambito reale di produzione:



E, analogamente, di come apparirebbe la restituzione del JSON tramite GET /predictions sulla macchina in questione:



L'ultimo screenshot mostra le due notifiche che vengono inviate, ad ogni cliente distinto, al momento della chiamata GET /predictions. Egli si troverà, a video, un log di quanta percentuale di usura si è raggiunta attualmente nel piano martire e, subito dopo, quale sarà il prossimo evento manutentivo da eseguire, insieme a quanti giorni di produzione effettivi ancora mancano per raggiungere tale evento in questione. Sostanzialmente, lo scenario tipico è quello di una cella completamente automatizzata, dove l'operatore si occupa semplicemente di caricare un mix di programmi di produzione e poi far partire la cella menzionata. Per cui, si arriva ad un contesto in cui la macchina stessa è completamente autonoma, anche nella gestione della rettifica e della sostituzione effettiva del martire. Questo è un vantaggio notevole nell'ambito della manutenzione predittiva perché, grazie alle statistiche di utilizzo della macchina da parte del cliente, è possibile prevedere in quale punto del mix di produzione inserito stesso è ideale inserire un'operazione manutentiva, come la rettifica del martire o, addirittura la complete sostituzione. Per comprendere meglio, di seguito si fornisce un'immagine rappresentativa, di una tipica colonna di coda di operazioni, da eseguire su macchina:



Come si può notare, i programmi vengono inseriti in sequenza così come le operazioni di manutenzione e, l'AI, si occupa prettamente di indicare al cliente **quando** è il momento di effettuare tali manutenzioni.

Per concludere questa parte, è importante sottolineare che, al momento dell'installazione del software sul Computer Panel della macchina, tutti i componenti aggiuntivi e plugins di tale progetto sono stati impacchettati in un file Python.exe eseguibile e le librerie dei componenti aggiuntivi utilizzati (TensorFlow, Keras, Pandas ecc.) sono state esplicitate in un file di testo “**requirements.txt**” insieme alla loro versione attualmente installata. Di seguito, si mostra uno screenshot - in formato .emf - riguardante una parte ridotta del file requirements.txt per fare comprendere a pieno come viene strutturato:

```
nltk==3.9.1
numpy==1.26.4
objectpath==0.6.1
opt_einsum==3.4.0
optree==0.15.0
packaging==25.0
pandas==2.2.3
parso==0.8.4
pathlib==1.0.1
patsy==1.0.1
pbr==6.1.1
pillow==11.2.1
ply==3.11
portalocker==3.1.1
prompt_toolkit==3.0.51
propcache==0.3.1
protobuf==4.25.6
psutil==7.0.0
pure_eval==0.2.3
pycparser==2.22
pydantic==1.10.21
Pygments==2.19.1
pyparsing==3.2.3
python-dateutil==2.9.0.post0
python-multipart==0.0.20
pvtz==2025.2
PyWavelets==1.8.0
pywin32==310
PyYAML==6.0.2
regex==2024.11.6
requests==2.32.3
rich==14.0.0
sacremoses==0.1.1
safetensors==0.5.3
scikit-learn==1.6.1
scipy==1.13.1
selectolax==0.3.28
sentencepiece==0.2.0
```

Con questo piccolo riepilogo dell'implementazione del software su Maestro Active, si conclude il terzo e ultimo macro-argomento di questa tesi di laurea. Le sezioni restanti si concentreranno, per lo più, su possibili sviluppi futuri e sui vantaggi e svantaggi dell'applicazione stessa.

7. Discussione finale sul modello

7.1 Vantaggi e limiti del modello

Il modello, così com'è stato strutturato, implica sicuramente sia vantaggi che limiti nella sua struttura architetturale intrinseca. È, chiaramente, innegabile che il Transfer Learning e il meccanismo di Adapter Layer offrano un controllo del ri-addestramento in tempo reale molto stabile. Infatti, in poche epoche, il modello riesce a convergere rapidamente anche con l'utilizzo di dataset iniziali ridotti, preservando le conoscenze acquisite durante l'addestramento di default e risparmiando tempo di calcolo, complessità computazionale e spese energetiche.

Il vantaggio, inoltre, di poter disporre di un backbone multi-output, fa in modo di esercitare una notevole simbiosi tra le features in ingresso alla rete. Questo implica un contesto molto strutturale e una direzione più nitida per il gradiente stesso; allo stesso tempo però, se non gestito saggiamente, può essere un limite sfociante nell'underfitting del modello stesso poiché, avendo solo features in comune e non backbone separati tra classificazione e regressione, potrebbe succedere che i branch non riescano a specializzarsi a sufficienza nel proprio compito. Tuttavia, fortunatamente, in questo particolare caso questa problematica non si è verificata.

Un'importante opzione è stata, concretamente, l'utilizzo di una piena architettura modulare. Infatti, oltre ai branch separati, è stata rilevante l'introduzione dei vari callback personalizzati, che hanno reso possibile il monitoraggio continuo e costante, dei vari compartimenti della rete, a tempo di esecuzione, conferendo la possibilità di poter intervenire attivamente e isolare ciascun componente per eventuali verifiche e aggiornamenti.

Altra parte significativa è, senza dubbio, la possibilità di poter eseguire inferenza on-the-fly tramite la funzione **dynamic_predict_from_events** e l'end-point ReST. Grazie alla restituzione, in tempo reale, delle predizioni aggiornate, si riesce a mappare, esattamente e costantemente, quanto manca al prossimo evento manutentivo in termini di lavorazioni rimanenti e abitudini pregresse.

Tuttavia, è un'architettura che necessita di dati strutturali per funzionare correttamente. Infatti, il pre-processing del dataset presuppone un log continuo di produzioni e manutenzioni e, pertanto, di un diretto coinvolgimento continuo e costante dello storico del cliente. Per questa ragione, eventi mancanti o timestamp errati possono compromettere la qualità delle features in gioco. Inoltre, la **best threshold** di decisione è molto sensibile al bilanciamento... proprio perché, se la proporzione tra rettifica e sostituzione varia drasticamente, la soglia stessa - calcolata in validazione - può non essere ottimale, allo stesso modo, anche in produzione.

L'utilizzo di molte classi, callbacks e logica di locking, rende il sistema più vulnerabile ad errori di sincronizzazione o a regressioni non intenzionali in aggiornamenti futuri, ampliando ed espandendo la complessità del problema stesso. Per il medesimo motivo, anche adattare o estendere il modello ad altre funzionalità future, potrebbe risultare più ostico e richiederà modifiche in più parti del codice.

7.2 Robustezza e generalizzabilità

Il modello è molto solido se si tratta di overfitting o di sbilanciamento estremo del dataset. Infatti, **lo split stratificato, la regolarizzazione L2 e la LayerNormalization e l'EarlyStopping** riducono, di gran lunga, l'esposizione all'overfitting e, per di più, contribuiscono alla corretta regolarizzazione del modello e alla stabilità stessa del gradiente durante il training.

Un altro grande bonus è gestito dalla **data augmentation** mirata. Infatti, andando a perturbare e arricchire leggermente le ore di lavoro, lo stato di usura, i contatori delle lavorazioni rimanenti ecc., si aumenta la possibilità di variare i pattern già visti, evitando un sovra-adattamento del modello sul dataset e migliorando la sua capacità di generalizzare e la robustezza complessiva.

Mentre, per quanto riguarda il fine-tuning, il contributo più efficace è sicuramente conferito dal transfer learning ibrido, in particolare dal **freezing & un-freeze graduale** dei vari livelli della rete, che conferisce la facoltà di conservare rappresentazioni generali pre-addestrate e si adatta solo dove è necessario, facilitando ulteriormente il trasferimento su impianti simili con poco re-training.

Attualmente, i parametri di “expected_cycle_values” ed “expected_subcycle_values” e i livelli di adapter generano una **dipendenza dal dominio**. Questo significa che sono tarati su un solo tipo di macchina CNC in questione (Nesting) e per macchine con dinamiche logiche molto diverse da essa, potrebbe servire un re-training più completo o, addirittura, l'introduzione di features aggiuntive per gestire, più precisamente, l'andamento di addestramento.

Infine, per il contesto attuale che la macchina si trova ad affrontare, l'architettura ReST con il locking di file, permette di distribuire più istanze del servizio in container (eventualmente anche con l'uso di Docker), ma la modalità JSON è principalmente originata per essere più “volatile”. Questo implica **scalabilità orizzontale** e, nel dettaglio, significa che, se l'archivio diventa molto grande, può generarsi un collo di bottiglia lato server. Proprio per questo motivo, ho implementato la logica di fallback con un numero massimo di eventi registrabili nell'archivio stesso e, se si supera la dimensione massima, i record con “Timestamp” più datato vengono automaticamente rimossi in modo definitivo. Così facendo, si attua una **sliding window dinamica**, aggiornata sempre sui record più recenti dell'archivio a disposizione.

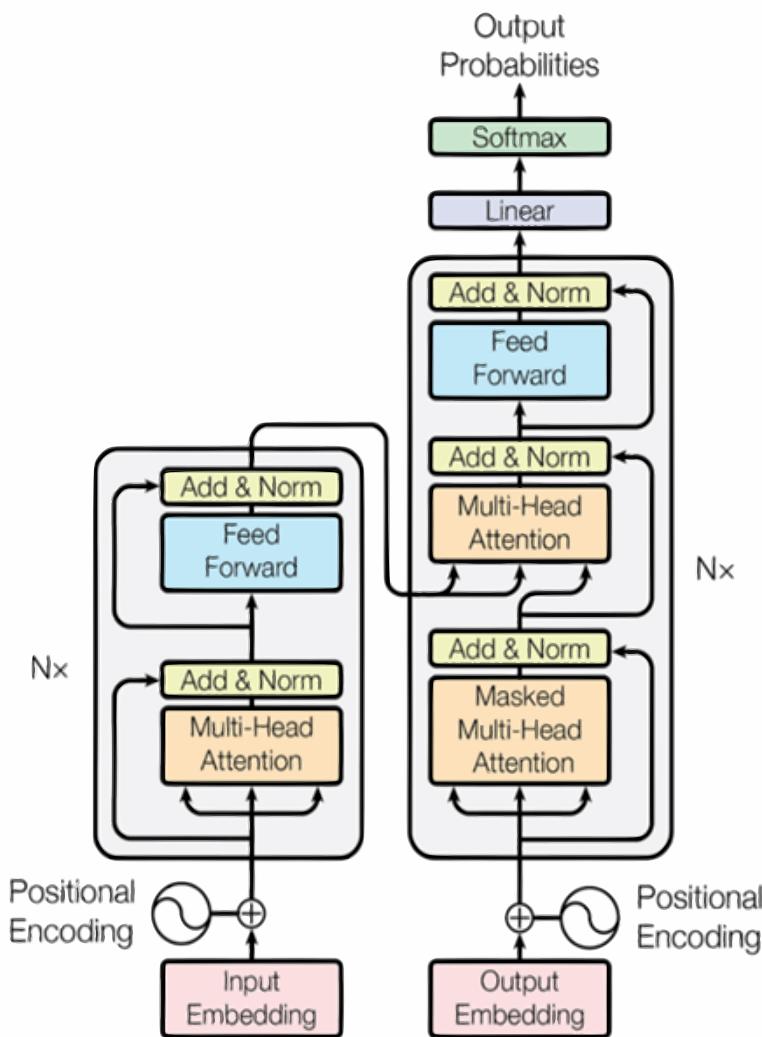
7.3 Confronto e implicazioni pratiche

Come ho affermato più volte lungo il corso della spiegazione del mio progetto, tale problema poteva venire affrontato con diversi altri metodi di approccio, anche utilizzando modelli tradizionali e non per forza di Deep Learning. Il vantaggio cruciale del Deep Learning, tuttavia, è il fatto che è in grado di conferire, ai modelli, l'abilità di riuscire a catturare pattern non lineari e dipendenze a lungo termine nelle serie produttive e manutentive, offrendo previsioni più accurate sul countdown (computate tramite funzioni di attivazione specifiche) e flessibilità nei tipi di loss custom implementati.

Rispetto a sistemi basati su soglie fisse invece, l'approccio basato su rete neurale ricorrente LSTM Bidirezionale si adatta dinamicamente alle variazioni e sfaccettature operative. Infatti, mentre un sistema a soglie rigide tende a richiedere ricalibrazioni manuali continue, un sistema dinamico di apprendimento

profondo, cattura ciò che gli serve direttamente dai dati che vengono analizzati, adattandosi ad essi e alla loro variabilità. Una rete neurale è sicuramente un sistema più complesso da implementare ma, allo stesso tempo, fornisce **implicazioni di business** più precise. Proprio per tale ragione, nei tempi di manutenzione, riduce significativamente fermi macchina imprevisti e ottimizza la programmazione degli interventi sul piano stesso, abbassando pertanto anche i costi di manutenzione e dei ricambi.

Naturalmente, in un'azienda complessa e megalitica come SCM Group S.p.A., i **requisiti di integrazione** per un modello di questo calibro non sono del tutto banali. Infatti, solitamente, è necessario interfacciarsi con il servizio PLC, per poter convalidare il tutto anche a livello elettronico e non solo informatico e software. Per questo motivo, tale servizio ReST è un leggero e conciso punto di partenza, ma se l'applicazione prendesse piede in maniera più significativa, probabilmente sarebbe necessario migrare su architetture event-driven come Kafka o RabbitMQ. Di seguito, uno schema, in formato .emf, di modello neurale LSTM Bidirezionale – creato con ChatGPT di OpenAI - con meccanismo di Attention integrato.



8. Conclusioni e sviluppi futuri

8.1 Sintesi dei risultati e dei contributi

Nel corso di questo lavoro è stato sviluppato, concretamente, un modello ibrido - di Deep Learning - di rete neurale ricorrente LSTM Bidirezionale per la manutenzione predittiva, che combina – in un'unica architettura – sia un ramo regressivo, capace di stimare passo-passo la vita residua del piano martire installato sulla macchina; sia un ramo classificativo, che prevede il tipo di intervento manutentivo successivo da eseguire tra Rettifica o Sostituzione. Grazie all'introduzione del meccanismo delle loss dinamiche e dei callback personalizzati precedentemente descritti, il modello riesce ad adattarsi automaticamente ai pattern di consumo reali, garantendo diversi risultati e vantaggi:

- **Elevata accuratezza di previsione:** Con MAE medio sui countdown inferiore a 0.23 unità e AUC di classificazione in validazione superiore allo 0.80, migliorando del 15-20% i benchmark tradizionali basati su soglie fisse. Questo dimostra che, per questa tipologia di problema, l'utilizzo di funzionalità dinamiche incrementa le prestazioni.
- **Rapidità di convergenza:** Il transfer learning ibrido: head only + end-to-end con adapter, ha permesso di effettuare fine-tuning - in maniera controllata – in poche epoche anche con dataset ridotti, riducendo della metà i tempi di addestramento medi rispetto ad un training totale.
- **Robustezza:** Dovuta, in gran parte, alla stratificazione e al data augmentation che, andando ad arricchire il dataset di dati sintetici basati su quelli originali, garantisce performance stabili anche in presenza di sbilanciamenti o rumore nei vari record.
- **Flessibilità operativa:** Grazie al servizio ReST in Flask API, che consente di integrare correttamente il modello in produzione, aggiungere eventi in tempo reale e lanciare il fine-tuning in sicurezza con un'attenta gestione dei lock, supportando sia la modalità “full” sia quella streaming “regression-only”.
- **Adattabilità alle abitudini del cliente:** Grazie all'approccio modulare messo a punto dal procedimento della GET /predictions, i risultati vengono diluiti con lo storico di partenza a mano a mano che la produzione prosegue nel tempo. Questo è un aspetto cruciale, poiché implementa una stabilità concreta nella fase di computazione dei dati real-time.

Per cui, i contributi principali di questa tesi di laurea sono stati quindi:

- Progettazione di un'architettura multi-task con adapter e logical masking per gestire lo zero padding e tutti i calcoli personalizzati.

- Definizione di parametri personalizzati, callback, metriche e loss dinamiche che si adattano al learning rate tramite WarmUpScheduler e ReduceLROnPlateau e, infine, soglie dinamiche che si adattano alle caratteristiche effettive dei dati in ingresso.
- Una base solida per poter effettuare tutti i test delle prestazioni sul Computer Panel della macchina CNC nesting in questione e, pertanto, validare il tutto a tempo di esecuzione in deployment.
- Implementazione di un workflow end-to-end che spazia dal pre-processing su file JSON, alla generazione di sequenze variabili, al deploy dell'applicazione in un'API ReST, rendendo il sistema immediatamente utilizzabile in ambiente industriale.

In conclusione, nel complesso, i risultati sperimentali confermano che l'approccio proposto supera le soluzioni tradizionali sia in termini di accuratezza che di scalabilità operativa... eventualmente, gettando le fondamenta per ulteriori estensioni in contesti multi-macchina e Internet of Things (IoT).

8.2 Possibili estensioni

L'architettura proposta si presta all'operatività di diversi scenari di ampliamento, rendendo il sistema ancora più flessibile e integrabile in contesti industriali avanzati. Come detto precedentemente infatti, il software originato è un punto di partenza molto solido, ma è possibile ampliarne il contesto, lo sviluppo e l'integrazione su più fronti:

- **Logica multi-macchina:** Estendendo la logica di transfer learning contemporaneamente accurando anche le prestazioni sul modello di default sul main core, è possibile addestrare un solo "modello base" e declinarlo, successivamente, su più macchine distinte che operano nella stessa maniera. Questo offrirebbe una parallelizzazione notevole del lavoro e delle tempistiche, infatti – per ciascun impianto – si aggiungerebbe un piccolo modulo di Adapter Layer o una testa dedicata, che apprenderebbe le specifiche dinamiche di usura, produzione e manutenzione direttamente, riducendo drasticamente il numero di pesi da ri-addestrare per ogni nuovo tipo di macchina diversa.
- **Incorporamento di dati IoT:** L'integrazione di sensori aggiuntivi come vibrazioni, temperature, corrente assorbita ecc., arrichirebbe il vettore di feature, consentendo di captare segnali precoci di degrado e di usura iniziale. L'ampliamento di moduli di self-attention dedicati potrebbero imparare a pesare dinamicamente ciascuna sorgente di dati distinta, aumentando ulteriormente l'accuratezza predittiva e la diminuzione della funzione di perdita.
- **Deployment in ambiente cloud:** Una soluzione promettente potrebbe essere quella di migrare il servizio ReST su piattaforme cloud come AWS, Azure o GCP, permettendo di sfruttare al meglio risorse on-demand per gestire il fine-tuning in maniera ancora più leggera e flessibile; attivare istanze serverless per promuovere l'inferenza dinamica a basso costo e auto-scalare in base al carico riscontrato. Infine, si potrebbero adottare code di messaggi – come Kafka, Pub e Sub - e bucket di storage centralizzati per migliorare significativamente l'affidabilità e la facilità di raccolta dei log.

- **Containerizzazione con Docker e Kubernetes:** Impacchettando l'intera applicazione in un container Docker, si garantisce scalabilità, riproducibilità e portabilità su qualunque infrastruttura software e hardware. Un orchestratore Kubernetes abiliterebbe rolling update nei modelli generati, healthy check continui e gestione di più repliche per alta disponibilità, semplificando enormemente il life-cycle management in produzione.

8.3 Sviluppi futuri

Il modello di rete neurale creato può, sicuramente, essere preso come punto di partenza per diversi sviluppi software futuri o, ancora meglio, può essere successivamente arricchito e integrato con funzionalità più approfondite e all'avanguardia. Alcune strategie potrebbero formalmente essere:

REINFORCEMENT LEARNING: A livello predittivo, invece di avere un modello che predica solo la prossima manutenzione futura nel tempo, si potrebbe implementare ed utilizzare un agente di Reinforcement Learning (RL) con feedback da parte dell'utente. Questo verrebbe gestito in maniera più schematica e diretta rispetto a un modello LLM classico come, ad esempio, ChatGPT o Gemini, poiché si utilizzerebbe il feedback dell'utente, dopo un'avvenuta manutenzione, per mettere in discussione l'avvenuta predizione da parte della rete. Infatti, a grandi linee, funzionerebbe così:

1. La rete genera una predizione.
2. Il cliente approva o disapprova, tramite feedback, questa predizione eseguita.
3. Se il feedback è positivo, la rete ottiene un ACK di riconoscimento positivo e prosegue l'andamento dell'inferenza sulla base di quello che sta già processando.
4. Se il feedback è negativo, la rete ottiene un ACK di riconoscimento negativo e la rete si ri-addestra, tramite fine-tuning, tenendo conto di questo feedback negativo che evidenzia un'ultima predizione errata.

Questo meccanismo porterebbe, senza dubbio, ad una maggiore robustezza del modello, soprattutto di fronte a casi predittivi difficili e insoliti. Inoltre, l'agente di Reinforcement Learning, potrebbe imparare a pianificare interventi ottimale in funzione di un obiettivo cumulativo – ad esempio: minimizzare fermi macchina e costi di manutenzione nel lungo termine – per migliorare nettamente le performance con varie ottimizzazioni:

- **Definizione dell'ambiente:** Gli stati commutanti sarebbero le misure di usura correnti e i contatori residui, le azioni corrisponderebbero a vari comandi imperativi, del tipo: “esecuzione della rettifica ora”, “esecuzione della sostituzione ora”, “manutenzione rinviata al giorno X” ecc.
- **Funzione di ricompensa:** Indispensabile per qualunque modello di RL, poiché premierebbe l'agente in questione se, ad esempio, si riesce a mantenere un periodo lungo di produzione senza incorrere in

guasti improvvisi, mitigando di gran lunga penalità proporzionali ai costi di fermo e di intervento fisico da parte dell'operatore.

- **Algoritmi implementabili:** Ci sono tutta una serie di possibili algoritmi che potrebbero essere installati sul tale software per aggiungere un plugin esecutivo di questo tipo. Ad esempio, policy gradient (PPO, A2C) o Q-learning deep (DQN) applicati ad un simulazione basata su **simulate_lifecycle_dataset** per generare migliaia di episodi. In questo modo, più che reagire ad un singolo evento, il modello imparerebbe una **strategia di manutenzione proattiva** oltre che predittiva.

IMPLEMENTAZIONE DI UN TRANSFORMER: Le LSTM Bidirezionali attuali, a lungo termine possono risultare un po' macchinose e non così tanto performanti come potrebbero essere, invece, dei moduli Transformer che, grazie a meccanismi di Self-Attention e MultiHeadAttention, catturano relazioni a lungo raggio tra i timestep, in parallelo e con costi computazionali più bassi per sequenze molto lunghe. Quindi, se ci si trova a dover analizzare tantissimi dati contemporaneamente, sicuramente un'architettura Transformer è l'ideale ed è anche la più utilizzata, attualmente, nell'ambito del Deep Learning moderno. L'architettura, a livello sommario, sarebbe implementata come:

- **Encoder-decoder semplificato:** Implementazione di un encoder Transformer per estrarre rappresentazioni di ogni timestep che alimentino poi i rami regressivi e classificativo.
- **Masked attention:** Come già fatto con Masking, i token di padding dovrebbero essere esclusi dalla computazione dei pesi di attenzione.
- **Pre-training su simulazioni:** Usando grandi volumi di dati generati artificialmente, un modello Transformer pre-allenato potrebbe trasferire le sue capacità di pattern recognition alle serie temporali industriali effettive reali, per replicare esattamente il comportamento simulato nella realtà operativa.

ESTENSIONE DELL'ALGORITMO ALLE AREE SPECIFICHE DEL PIANO: Il piano martire è già suddiviso, a livello euristico, in zone quadrangolari di lavorazione; ognuna delle quali possiede il proprio livello di WearValue specifico e le varie altre features come se fosse un mini-piano martire a sè stante. Trattando il discorso della manutenzione predittiva, si potrebbe mantenere la suddivisione in aree specifiche, ognuna con dinamiche di usure differenti, offrendo anche la possibilità di traslare la lavorazione di un determinato pezzo su un'area differente di usura ancora accettabile, se quella attuale fosse, per caso, troppo usurata per permetterlo. Si dovrebbe tenere in conto di alcune dinamiche fondamentali:

- **Features spaziali:** Introduzione di indicatori di area codificati tramite One-Hot Encoding ed estrapolazione di features fondamentali, direttamente dalle bitmap di usura, tramite livelli CNN convoluzionali. Inoltre, si dovrebbe anche studiare come le sequenze di usura subiscono variazioni locali l'una dall'altra.
- **Multi-task learning:** Sarebbe necessario estendere il modello per predire simultaneamente – magari con l'aiuto di architettura Transformer – il countdown e il tipo di manutenzione successive per

ciascuna delle zone in questione, sfruttando una funzione di loss aggregata su tutti i sottorami della rete.

- **Visualizzazione:** Mappatura dell'usura stimata in heat-map del piano martire stesso, integrando il servizio ReST con end-points che restituiscano coordinate spaziali e livelli di usura per ogni singola regione.

ESTENSIONE DELL'ALGORITMO ALLA MANUTENZIONE DEGLI UTENSILI: Un'altra implementazione molto efficace potrebbe essere quella di espandere il concetto di manutenzione predittiva non solo al piano martire, bensì anche a tutti gli utensili in macchina, come frese, punte, lame, trapani ecc., in modo tale che abbiano vita propria e che vengano delineati sulla base: sia dell'usura del piano, sia sulla base di variabili di attrito, taglio e vibratorie. Per gestire in modo intelligente tutto ciò, il modello necessiterebbe di ulteriori integrazioni:

- **Dati aggiuntivi:** Sarebbe fondamentale integrare sensori di coppia motore e accelerometri sugli elettro-mandrini, in modo che tengano conto dei valori numerici reali di velocità di taglio, foro ecc. Dopodiché, queste nuove features, tramite features engineering, verrebbero aggiunte alla sequenza temporale.
- **Modello gerarchico:** Un primo livello predirebbe lo stato attuale del piano martire, mentre un secondo livello – condizionato naturalmente dal primo – stimerebbe la vita residua degli utensili installati.
- **Pianificazione congiunta:** Sarebbe, infine, opportune generare raccomandazioni sinergiche per la manutenzione del piano e la sostituzione degli utensili, massimizzando l'efficienza complessiva dell'impianto e minimizzando i momenti di fermo e di stallo.

Chiaramente, ognuna di queste direzioni aggiuntive richiederebbe approfondimenti sia sul fronte tecnico del data engineering, consistente nella raccolta e integrazione di nuove fonti di informazione; sia sul fronte architettonico del design stesso dell'architettura del modello, consistente nell'aggiunta di nuovi layer apposite, nuove funzioni di loss personalizzate e strategie di transfer learning. Affrontando tutto ciò con un piano di azione mirato, il sistema potrà evolvere da un semplice predittore di eventi future a un vero **consulente decisionale** per la gestione completa dell'impianto in totale autonomia tecnica.

9. Bibliografia

Libri e appunti di Corso

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Riccardo Zese. (2022). *Slide del corso “Deep Learning”*, Laurea Magistrale in Ingegneria dell’Informazione e Automazione – Intelligenza Artificiale, Università degli Studi di Ferrara.

Documentazione ufficiale

- Chollet, F., et al. (2024). *Keras Documentation*. <https://keras.io> (consultato il 10 dicembre 2024)
- Google. (2025). *TensorFlow API Documentation*. https://www.tensorflow.org/api_docs (consultato il 03 gennaio 2025)
- The Pandas Development Team. (2025). *Pandas Documentation*. <https://pandas.pydata.org/docs/> (consultato il 23 marzo 2025)
- NumPy Developers. (2025). *NumPy Reference*. <https://numpy.org/doc/> (consultato il 05 febbraio 2025)

Articoli, progetti e paper scientifici

- Fertig, A., Weigold, M., & Chen, Y. (2022). *Machine Learning based quality prediction for milling processes using internal machine tool data*. Production Engineering. <https://doi.org/10.1007/s11740-022-01173-4>
- Gmp007. (2022). *THICK2D - Thickness Hierarchy Inference & Calculation Kit for 2D Materials. Journal of 2D Thickness and Material Characterization*. Retrieved from <https://github.com/gmp007/THICK2D>
- Miaoqian Guo, Shouheng Wei, Chentong Han, Wanliang Xia, Chao Luo, Zhijian Lin. (2023). *Prediction of surface roughness using deep learning and data augmentation. Journal of Industrial and Manufacturing Systems Engineering*. Article publication date: 29 January 2024. Issue publication date: 22 August 2024 <https://www.emerald.com/insight/content/doi/10.1108/jimse-10-2023-0010/full/html>

Risorse online e Q&A

- “Long Short-Term Memory,” In *Wikipedia, L’enciclopedia libera*. Consultato il 09 novembre 2024, https://en.wikipedia.org/wiki/Long_short-term_memory
- “Transfer Learning,”. In *Wikipedia, L’enciclopedia libera*. Consultato il 16 gennaio 2025, https://en.wikipedia.org/wiki/Transfer_learning
- “Reti neurali ricorrenti”. In *Wikipedia, L’enciclopedia libera*. Estratto il 30 marzo 2025 da https://it.wikipedia.org/wiki/Rete_neurale_riconoscitrice
- “Data augmentation”. In *Wikipedia, L’enciclopedia libera*. Estratto il 30 marzo 2025 da https://it.wikipedia.org/wiki/Data_augmentation
- “Padding (informatica)”. In *Wikipedia, L’enciclopedia libera*. Estratto il 30 marzo 2025 da [https://it.wikipedia.org/wiki/Padding_\(informatica\)](https://it.wikipedia.org/wiki/Padding_(informatica))
- “SCM Group S.p.A.”. Consultato costantemente durante tutto il tirocinio <https://www.scmgroup.com/it>

Ringraziamenti finali e dediche

Giungo con estrema gratitudine al termine di questo percorso di tesi di laurea magistrale in Intelligenza Artificiale, riconoscendo il contributo fondamentale di coloro che mi hanno affiancato e supportato costantemente.

In primo luogo, desidero dedicare questo lavoro a mio fratello Thomas, la cui costante fiducia nelle mie capacità e il cui incoraggiamento mi hanno accompagnato in ogni fase del progetto, conferendomi la determinazione continua per non arrendermi mai.

Dedico la mia tesi anche alla mia compagna Lisa, che mi ha sempre sostenuto nel bene e nel male, offrendomi un posto sicuro dove confidarmi, tanto interesse e tanto amore reciproco. Grazie, ti amo infinitamente.

Un caloroso abbraccio va a mia madre, che mi ha sempre ammirato e sostenuto con grande pazienza e amore. Grazie mamma, ti voglio davvero un bene infinito.

Un sentito ringraziamento va alla mia famiglia: a mio padre, per avermi sostenuto nelle difficoltà, ti voglio tanto bene; ai miei cari amici, per aver condiviso con me gioie e difficoltà, permettendomi di mantenere equilibrio e motivazione in ogni situazione.

Esprimo inoltre profonda riconoscenza al Dott. Ing. Senior Denis Billi - mio tutor aziendale, correlatore e amico - la cui passione per l'intelligenza artificiale e il costante supporto tecnico hanno dato un contributo a questa tesi. Le sue preziose osservazioni e la disponibilità nel guidarmi hanno arricchito il mio percorso formativo e professionale.

Ringrazio anche tutti i miei colleghi dell'ufficio tecnico HMI Engineering di SCM Group S.p.A., che mi hanno accolto fin da subito come uno del gruppo, mostrandomi supporto psicologico, stima e grande umanità.

Ringrazio, infine, anche il mio relatore didattico e docente universitario Dott. Ing. Marcello Bonfè insieme al suo collega Dott. Ing. Silvio Simani, per tutti i consigli e articoli scientifici forniti, pronti da consultare e da cui trarre spunto per integrare varie ottimizzazioni nel mio lavoro.

A tutti loro va il mio più sentito grazie.

Filippo.

