ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA
Sede di Forlì
Corso di Laurea in
INGEGNERIA AEROSPAZIALE Classe L-9

ELABORATO FINALE DI LAUREA
In ING-IND/05 IMPIANTI E SISTEMI AEROSPAZIALI

OTTIMIZZAZIONE DELLE PERFORMANCE DI UN FILTRO DI NAVIGAZIONE MEDIANTE L'USO DI RETI NEURALI

RELATORE:
PAOLO TORTORA

CANDIDATO:
MATTIA TADIOTTO

CO-RELATORE:
AUREL ZEQAJ

Anno Accademico 2022/2023

Sommario

La seguente attività di tesi ha come obiettivo l'implementazione di una rete neurale che funga da supporto ad un filtro di navigazione utilizzato per stimare l'orbita di un piccolo satellite utilizzato per l'esplorazione di corpi minori.

La tesi si compone di due parti, una prima dove viene testato il filtro per ricavare i parametri che ne ottimizzano le prestazioni e una seconda dove viene sviluppata una rete neurale che analizza i dati di input del filtro, i.e. le osservabili, ed estrapola la quantità di rumore che contengono.

Indice

Lista delle figure

Lista delle tabelle

Lista degli acronimi

ADAM:	Adaptive Moment Estimation
AI:	Artificial Intelligence
CNN:	Convolutional Neural Network
CPU:	Central Processing Unit
EKF:	Extended Kalman Filter
GPU:	Graphic Processing Unit
GUI:	Graphic User Interface
IA:	Intelligenza Artificiale
KF:	Kalman Filter
LSTM:	Long-Short Term Memory
MSE:	Mean Square Error
NASA:	National Aeronautics and Space Administration
NLP:	Natural Language Processing
NN:	Neural Network
OSIRIS-REX:	Origins, Spectral Interpretation, Resource Identification, Security, Regolith Explorer
PINN:	Physics-Informed Neural Network
PM:	Point Mass
ReLu:	Rectified Linear Unit
RMS:	Root Mean Square
RNN:	Recurrent Neural Network
RSS:	Residual Sum of Squares
SGD:	Stochastic Gradient Descent

SPH:	Spherical Harmonics
SRP:	Solar Radiation Pressure
STD:	Standard Deviation
SW:	Software
YORP:	Yarkovsky-O'Keefe-Radzievskii-Paddak

1. Introduzione

Lo spazio è sempre stato un punto fermo nella vita degli esseri umani, a partire dall'antichità dove si utilizzavano le stelle nella notte per orientarsi e muoversi sulla Terra, fino ad arrivare ai giorni nostri, dove per navigare satelliti e navicelle spaziali nel sistema solare facciamo uso ancora di stelle e pianeti.

L'esplorazione spaziale, in modo particolare, è stata una delle sfide più ambiziose dell'umanità, tanto da far nascere una vera e propria "corsa" per dimostrare la superiorità di una nazione sulle altre; l'ardua impresa di esplorare l'universo che ci circonda richiede una serie di ingredienti che devono saper cooperare in maniera coesa e precisa, quali tecnologie avanzate, alte competenze scientifiche e un profondo desiderio di scoperta.

Tra gli aspetti più critici di qualsiasi missione spaziale, risiede la necessità di dovere navigare con precisione attraverso lo spazio profondo, attraverso il controllo completo del mezzo su cui si sta navigando o che si vuole far navigare, ma anche determinando la posizione, la velocità e altri parametri fondamentali per il successo di una missione.

I filtri di navigazione sono essenziali per stimare i parametri non direttamente misurabili in un ambiente dinamico e incerto, come, ad esempio, lo spazio. La capacità di poter fare delle stime precise è indispensabile per il successo delle missioni e la raccolta di dati scientifici di valore. Tuttavia, nonostante l'affidabilità dell'EKF, o di altri filtri di navigazione, la loro capacità di gestire dati altamente variabili e complessi può essere limitata.

In questo contesto, l'IA emerge come una promettente alleata alla navigazione spaziale. Le reti neurali rappresentano una delle innovazioni più interessanti in questo senso, offrendo la possibilità di analizzare sequenze temporali complesse e apprendere da esse. Questa tecnologia può essere applicata per migliorare le performance degli EKF attraverso la determinazione della deviazione standard del rumore presente all'interno dei dati forniti in input al filtro.

Questa tesi vedrà in dettaglio come l'IA, in particolare la rete LSTM, possa essere utilizzata per affinare le stime di navigazione nello spazio, rappresentando in questo modo un valido supporto al lavoro effettuato dal filtro di navigazione.

2. Studio di un filtro di navigazione

2.1 I filtri di navigazione

Nel mondo dell'ingegneria e della navigazione, un filtro di navigazione è un componente fondamentale per determinare con precisione la posizione, la velocità e altri parametri che definiscono lo stato del sistema che si sta studiando in condizioni in cui le informazioni in input possono essere incomplete o affette da errori.

I filtri di navigazione rappresentano una classe di algoritmi e tecniche progettate per stimare lo stato attuale di un sistema dinamico in base alle osservazioni e alle misurazioni disponibili.

Questi algoritmi aggiornano costantemente le stime dello stato del sistema man mano che nuove informazioni vengono acquisite. Con "stato del sistema" di un filtro di navigazione si intendono l'insieme di variabili o parametri che stiamo monitorando.

Tra i filtri di navigazione più utilizzati si trova la famiglia dei filtri di Kalman; ne esistono di diversi tipi, tra cui il Linearized Kalman Filter, l'Extended Kalman Filter, oppure l'Unscented Kalman Filter.

2.2 Filtro di Kalman

Il filtro di Kalman è un algoritmo che combina le informazioni a priori fornite dal modello dinamico con i dati misurati, solitamente affetti da rumore, in un algoritmo sequenziale per determinare lo stato attuale del sistema e quelli futuri.

Analizziamo la matematica che regola il funzionamento di un KF andando a seguire il processo di predizione, misurazione e aggiornamento che il filtro esegue ogni qual volta riceve nuovi dati.

Questi tre step si basano su un modello definito a priori, che nel caso di un sistema che deve determinare la traiettoria di uno spacecraft, consistono nelle equazioni differenziali descriventi la dinamica di quest'ultimo.

Definiamo gli stati iniziali e le relative derivate come:

$$\boldsymbol{X}(t_0) = \boldsymbol{X}_0 \qquad \dot{\boldsymbol{X}} = f(\boldsymbol{X})$$
$$\boldsymbol{Y}(t_0) = \boldsymbol{Y}_0 \qquad \dot{\boldsymbol{Y}} = f(\boldsymbol{Y}) \qquad (1)$$

con f che indica una funzione dello stato del sistema. In seguito, descriviamo lo stato predetto come scostamento dal valore vero in ingresso, in quanto la stima effettuata sarà affetta da un errore.

Perciò, dato il termine $\delta \boldsymbol{x}$:

$$Y = X + \delta x \qquad \dot{Y} = f(X + \delta x) \tag{2}$$

Essendo, poi, $f(\boldsymbol{X} + \delta \boldsymbol{x})$ una funzione non lineare, andiamo ad attuare un'espansione in serie di Taylor:

$$\dot{\boldsymbol{Y}} = \dot{\boldsymbol{X}} + \delta \dot{\boldsymbol{x}} = f(\boldsymbol{X}) + \frac{\partial f(\boldsymbol{X})}{\partial \boldsymbol{X}} \delta \boldsymbol{x} + \frac{\partial f^2(\boldsymbol{X})}{2! \partial \boldsymbol{X}^2} \delta \boldsymbol{x}^2 + \cdots \tag{3}$$

Riunendo tutti i termini trascurati maggiori del primo grado in $\boldsymbol{z}$, possiamo dunque descrivere la dinamica del sistema in maniera linearizzata:

$$\delta \dot{\boldsymbol{x}} = \frac{\partial f(\boldsymbol{X})}{\partial \boldsymbol{X}} \delta \boldsymbol{x} + \boldsymbol{z} = \boldsymbol{F}(t) \delta \boldsymbol{x} + \boldsymbol{z} \tag{4}$$

Dove $\boldsymbol{F}(t)$ rappresenta la matrice Jacobiana, definita come:

$$F(t) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \tag{5}$$

Ponendo ora $\boldsymbol{z} = 0$, scriviamo la matrice della derivata parziale dello stato t rispetto allo stato precedente t_0 come $\boldsymbol{\Phi}(t, t_0)$, e definiamo le seguenti proprietà:

$$\begin{aligned} \dot{\boldsymbol{\Phi}}(t, t_0) &= \boldsymbol{F}(t) \boldsymbol{\Phi}(t, t_0) \\ \boldsymbol{\Phi}(t_0, t_0) &= \boldsymbol{I} \\ \Phi(t, t_0)^{-1} &= \Phi(t_0, t) \end{aligned} \tag{6}$$

Definiamo poi le seguenti relazioni:

$$\delta \boldsymbol{x} = \boldsymbol{\Phi}(t, t_0) \delta \boldsymbol{x}_0$$

$$\boldsymbol{v} = \int_{t_0}^{t} \boldsymbol{\Phi}(t, t_0)\, \boldsymbol{u}\, dt \tag{7}$$

Definito come $\boldsymbol{v}$ il rumore di processo, ovvero l'incertezza del modello dinamico usato nel processo di propagazione, nel nostro caso $\boldsymbol{v}$ è preso a media nulla ma potrebbe avere un termine di deviazione; con $\boldsymbol{u}$ è indicata la misura statistica dell'incertezza nella dinamica data dal modello utilizzato e da eventuali errori esterni agenti che fanno discostare la predizione dal valore reale.

Per completare l'introduzione sulla matematica che fa da fondamenta al funzionamento del filtro di navigazione in esame, ci manca da definire il processo di stima della matrice di covarianza $\boldsymbol{P}_k$, che determinerà la confidenza che ha il filtro a riguardo dei dati in ingresso, e la definizione del guadagno del filtro.

Partendo dalla propagazione del vettore che mi definisce l'errore sullo stato del sistema:

$$\delta x_{k+1} = \Phi \delta x_k + \nu \tag{8}$$

Descritto utilizzato la matrice dell'errore sullo stato $\boldsymbol{\Phi}$.

Successivamente, per propagare la matrice di covarianza in modo tale da valutare la bontà dello stato al tempo $t + 1$, si può scrivere $\boldsymbol{P}_{k+1}$ come il valore atteso dall'errore quadratico dello stato:

$$\boldsymbol{P}_{k+1} = E(\delta \boldsymbol{x}_{k+1}\, \delta \boldsymbol{x}_{k+1}^{T}) \tag{9}$$

Da cui, dopo una serie di passaggi matematici, possiamo riscrivere come:

$$\boldsymbol{P}_{k+1} = \boldsymbol{\Phi} \boldsymbol{P}_{k} \boldsymbol{\Phi}^{T} + \boldsymbol{Q} \tag{10}$$

Con $\boldsymbol{Q}$ momento del secondo ordine del rumore di processo, che tiene conto dell'errore di propagazione sulla matrice di covarianza nel tempo.

Infine, definiamo con $\boldsymbol{K}$ la matrice di guadagno del filtro, scritta come:

$$\boldsymbol{K} = \boldsymbol{P}\boldsymbol{H}^{T}\boldsymbol{W} \tag{11}$$

Con $\boldsymbol{W}$ matrice dei pesi ($\boldsymbol{W}^{-1} = \boldsymbol{R}$, $\boldsymbol{R}$ matrice che mi indica l'incertezza sulle misure), e $\boldsymbol{H}$ matrice delle derivate parziali sulle osservabili rispetto al tempo corrente.

$$H = \frac{\partial\, observations_{k+1}}{\partial x_{k+1}} \tag{12}$$

2.3 EKF

La prima differenza che caratterizza l'EKF rispetto al KF tradizionale è il calcolo del nuovo stato di riferimento, che avviene ad ogni nuova osservazione in modo da rappresentare sempre la stima migliore, ponendo perciò il vettore dell'errore sullo stato predetto uguale a zero $\delta \boldsymbol{x}_k = 0$.

Così facendo si ottiene un successivo vettore, secondo la formula vista nella sezione precedente, che sarà dunque: $\delta \boldsymbol{x}_{k+1} = \boldsymbol{\nu}$ **,** con $\boldsymbol{\nu}$ assunta a media nulla.

Facendo un esempio, se volessimo stimare il valore "vero" di una traiettoria date in ingresso le osservabili, ad ogni nuovo dato in input l'algoritmo andrebbe ad aggiornare lo stato per approssimare in modo quanto più preciso la traiettoria reale (vedi Figura 1).

Il risultato del metodo di calcolo dello stato appena descritto si può tradurre in una più alta adattabilità a problemi di natura non lineare.

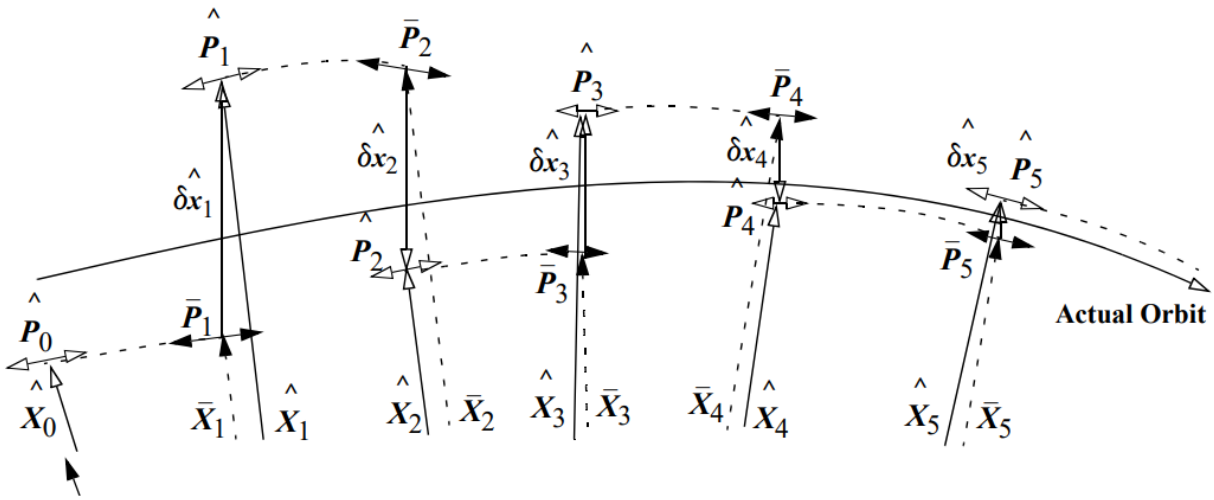

Figura 1: Rappresentazione di un filtro di Kalman esteso [1]

Grazie alle sue caratteristiche, l'EKF viene spesso utilizzato in applicazioni più complesse rispetto al KF, quali navigazione inerziale, tracciamento di obiettivi in movimento, localizzazione e controllo di velivoli autonomi, dove sono presenti modelli non complessi e spesso non lineari.

Uno svantaggio, però, che si porta dietro questo metodo è indubbiamente il costo computazionale più elevato rispetto alla sua versione lineare dato, ad esempio, dal fatto che la matrice delle derivate parziali $\boldsymbol{F}(t)$ dev'essere ricalcolata ad ogni iterazione e non è più una costante come sarebbe stata nel filtro di Kalman linearizzato.

2.4 Esempio pratico di funzionamento

Per capire meglio il significato fisico che risiede a valle di tutta la trattazione matematica vista in precedenza, di seguito è riportato un esempio generico preso da una delle varie simulazioni effettuate, nella quale vengono raffigurate due traiettorie, in rosso è rappresentata l'orbita "reale", mentre in azzurro è raffigurata l'orbita stimata dal filtro dopo aver ricevuto le osservabili. Dalla Figura 2 si può apprezzare il lavoro che effettua il filtro, dai momenti iniziali nei quali riceve i primi dati in ingresso, a quando si stabilizza sul valore finale. È chiaro come all'inizio della traiettoria stimata ci sia un transitorio nel quale il filtro cerca di bilanciare l'incertezza sul modello dinamico e quella sulle osservabili:

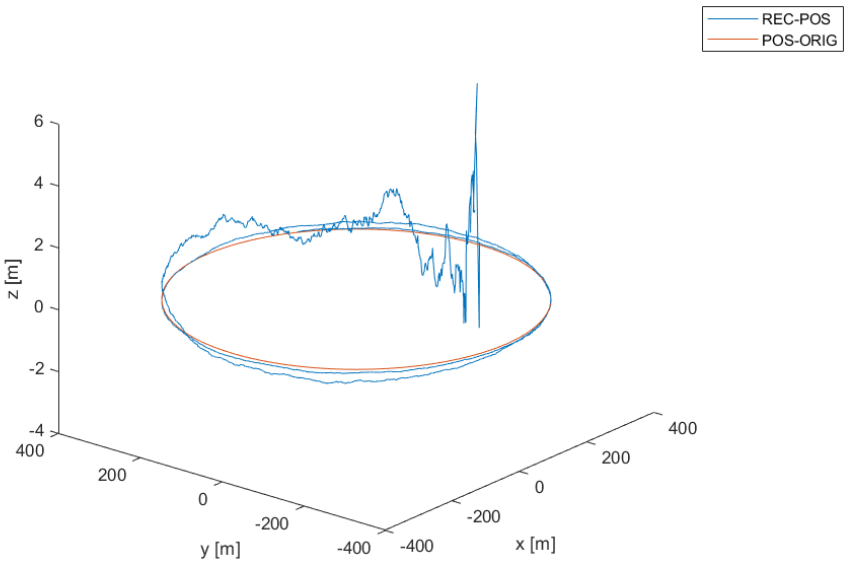

Figura 2: Rappresentazione simulazione filtro di Kalman esteso

Inoltre, può risultare utile al fine di comprendere più in dettaglio il comportamento del filtro, graficare il processo secondo il quale, dopo aver preso in ingresso dei dati contenenti rumore, il filtro convergerà avvicinandosi ai valori della traiettoria sintetica in x, y e z.

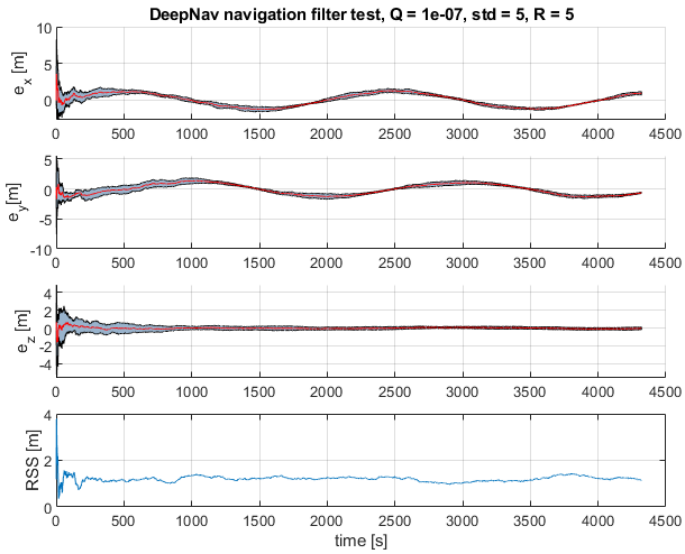

Figura 3: Grafici rappresentanti la convergenza del filtro sulla posizione

Questo plot (Figura 3) preso come esempio, è tratto dalla medesima prova effettuata per la Figura 2, dove i primi tre grafici rappresentano l'errore che presenta la traiettoria stimata da quella "ideale" nelle tre coordinate del sistema di riferimento utilizzato; mentre il quarto grafico rappresenta l'errore totale della traiettoria calcolato come la radice quadrata della somma dei quadrati (Root Sum Squared o RSS) dei residui – differenza tra valore calcolato e predetto - per ogni istante di tempo.

3. Reti Neurali

3.1 Funzionamento di una Rete Neurale

Una rete neurale (neural network, NN) è una struttura non-lineare che elabora i dati presi in input e li processa tramite delle modellazioni matematiche, riportando come output una previsione. Sono chiamate "reti neurali" perché ispirate al funzionamento del cervello umano, infatti, presentano diversi layer nei quali vengono processati i dati, ed ognuno di questi è formato da diversi "neuroni".

I layer sono di tre tipi:

- Input: utilizzati per introdurre nella rete i dati da processare;
- Hidden: chiamati così perché rappresentano la parte "nascosta" della rete neurale, ovvero quella che effettua le computazioni non lineari nei dati. L'hidden layer può essere singolo, come ad esempio nelle reti neurali più semplici, oppure ne possono essere presenti diversi, come nelle reti neurali più complesse o nel caso di "deep learning";
- Output: l'ultimo livello della rete, restituisce la previsione a seguito dei passaggi precedenti.

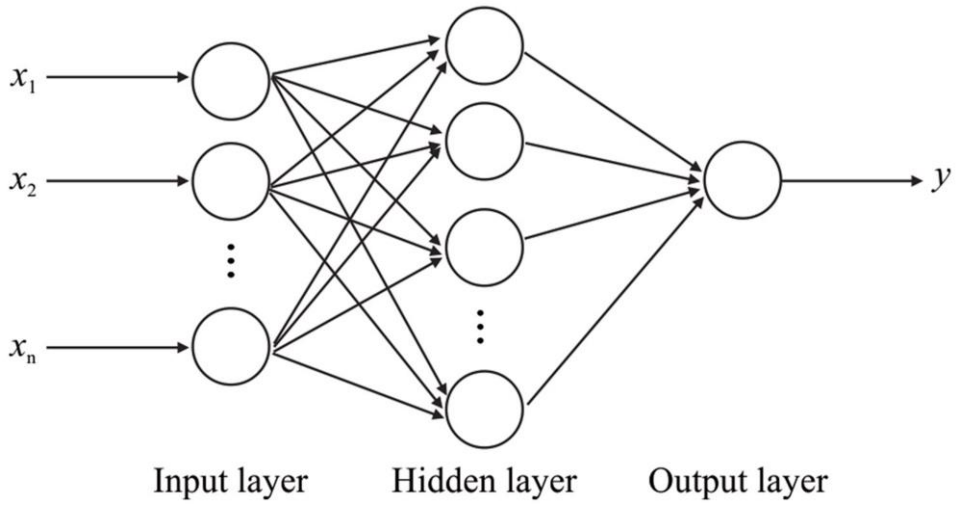

Figura 4: Architettura generale di una rete neurale [2]

Come detto prima, ogni layer è formato da diversi neuroni, all'interno dei quali vengono incanalati i dati provenienti dal livello precedente. Ad ogni dato è applicato un peso, un bias ed una funzione di attivazione; il bias è un termine che viene sommato ai dati di input moltiplicati per il relativo peso, e che serve ad aumentare la flessibilità del modello nelle sue predizioni, mentre la funzione di

attivazione serve a trasformare matematicamente il valore della computazione avvenuta nel neurone prima di passarlo allo strato successivo.

Approfondiamo di seguito il ruolo di ogni elemento dividendo l'analisi nelle due principali fasi di training di una rete neurale:

- Forward Propagation
- Backward Propagation

3.1.1 Forward Propagation

La prima di queste due fasi è quella di Forward Propagation, nel quale l'algoritmo genera delle previsioni sulla base dei dati in input.

Analizzando la struttura di un neurone, (Vedi Figura 5), possiamo notare come ad ogni singolo dato in input venga applicato un "peso" che serve a valutare quanta influenza avrà nel layer. Solitamente i pesi ad inizio training vengono inizializzati in maniera uniforme, ma con valori randomici vicini allo zero.

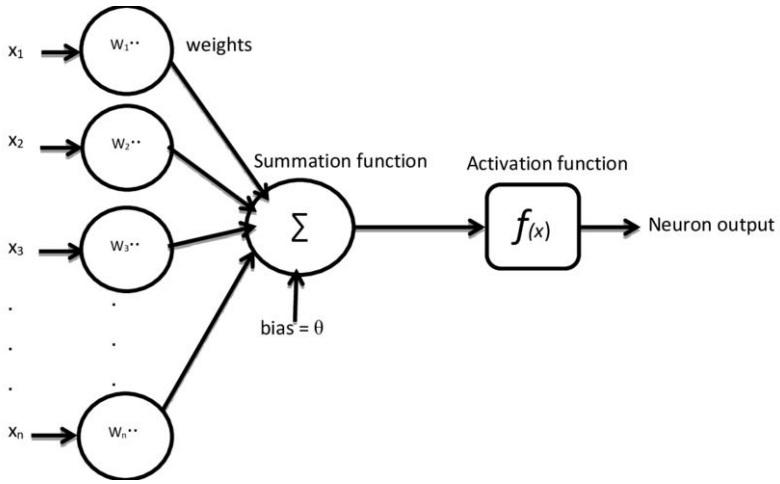

Figura 5: Rappresentazione di un neurone[1]

Insieme agli input con i relativi pesi, all'interno di un neurone viene sommato un valore chiamato "bias" che ha la funzione di shiftare il risultato. Con questa operazione finisce la parte di computazione lineare descritta dalla seguente equazione:

[1] Rappresentazione di un neurone artificiale disponibile qui: https://www.researchgate.net/figure/The-structure-of-the-artificial-neuron_fig2_328733599 (data: 30/09/2023)

$$z = \sum_{i=1}^{n} x_i w_i + b \quad (13)$$

All'output lineare dev'essere applicata una funzione non lineare chiamata "funzione di attivazione" per consentire alla rete neurale di imparare e svolgere le operazioni non lineari atte ad associare l'input all'output:

$$\hat{y} = \sigma(z) \quad (14)$$

Ci sono svariate tipologie di funzioni di attivazione in base alle diverse esigenze; per esempio, la ReLu è una delle più utilizzate e consiste nel mantenere il valore in uscita dalla parte lineare invariato se questo è maggiore o uguale a zero, altrimenti i valori vengono portati a zero, oppure la funzione di attivazione, o ancora la funzione sigmoidea, la tangente iperbolica, ecc.

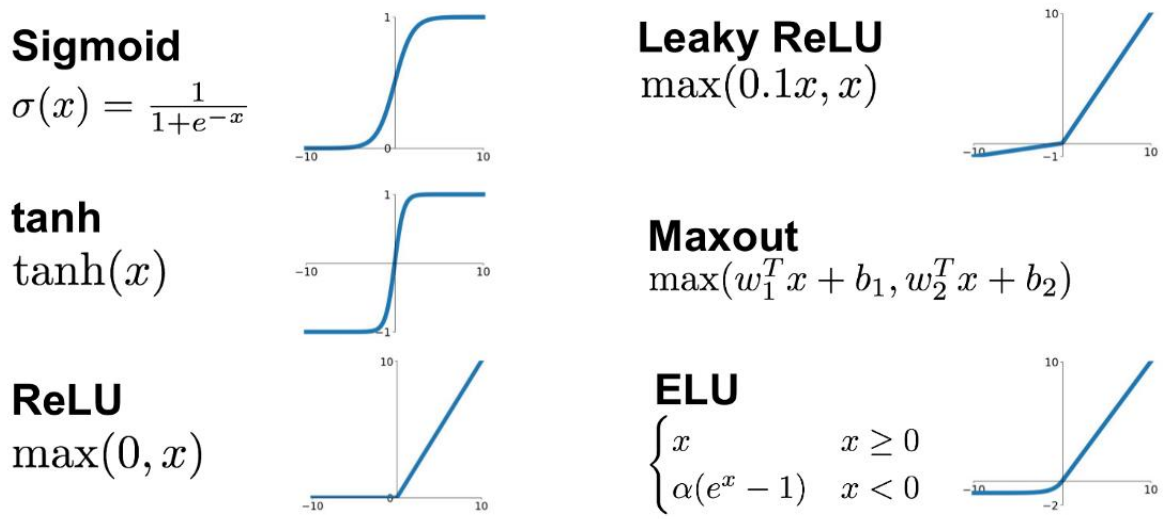

Figura 6: Funzioni di attivazione applicabili ad un neurone[2]

Quanto visto avviene in ogni singolo neurone della rete, indipendentemente che sia una rete semplice ad un solo layer e con pochi neuroni, od una sviluppata per il deep learning composta da molti neuroni e layers.

Infine, per completare l'analisi della prima parte di Forward propagation, si deve confrontare la predizione in output dalla rete con il valore reale tramite una "loss function", in modo tale che la rete possa comprendere di quanto sia sbagliata la predizione che ha ottenuto alla fine di ogni ciclo.

Come per le activation functions, esistono molteplici tipi di loss functions per soddisfare esigenze diverse in base al tipo di dati che stiamo trattando o in base al tipo di predizione che vogliamo ottenere. Per esempio, ci sono delle loss functions studiate per i problemi di classificazione come la

[2] Immagine funzioni di attivazione disponibile qui: https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c (data: 30/09/2023)

“Cross Entropy Loss” o la “Hinge Loss”, altre più adatte ai problemi di regressione come la “MSE Loss” o la “MSA Loss”, e così via.

Per comprendere meglio quanto detto prima facciamo un piccolo excursus sui problemi di classificazione e regressione. Le reti progettate per risolvere il primo tipo di problema hanno come obiettivo l’assegnazione di un’istanza di input ad una delle classi definite, l’output sarà dunque una variabile discreta; discorso opposto per la regressione, dove l’output sarà un valore numerico continuo o una serie di valori numerici.

3.1.2 Backward Propagation

Proseguendo poi con la seconda fase, quella di Backward Propagation, che costituisce il cuore della fase di training, all’interno della rete vengono modificati tutti i pesi in modo tale minimizzare l’errore. L’ottimizzazione dei pesi avviene attraverso l’ausilio un componente chiamato “optimizer”, il cui ruolo è quello di ridurre le perdite e adeguare il learning rate; proprio quest’ultimo è un parametro importante che regola la dimensione dei “passi” che la rete compie in direzione dell’ottimo globale nel gradiente della cost function. Un learning rate troppo altro potrebbe far oscillare l’ottimizzazione o farla convergere ad un punto lontano dall’ottimo globale, al contrario, se il learning rate fosse troppo basso la funzione potrebbe procedere con l’ottimizzazione in maniera troppo lenta o fermarsi in un minimo locale.

Loss function e cost function sono concetti leggermente differenti in quanto la loss function si riferisce alla singola iterazione, mentre la cost function, come detto prima, è definita globalmente e ciò vuol dire che si riferisce alla rete in generale.

Le fasi di forward propagation e backward propagation vengono eseguite ogni volta che la rete deve processare degli input, in particolare, per allenare la rete neurale vengono creati nel codice due cicli iterativi; il più interno effettua i cicli sulla lunghezza del dataset che viene fornito alla rete, mentre il secondo (quello più esterno) effettua i cicli sul numero di “epoche” scelte dal programmatore in modo arbitrario; il numero di epoche rappresenta quante volte la rete si allena sull’intero dataset di training.

Al termine della fase di training può essere presente una parte del dataset originale con cui la rete neurale non si è allenata che viene fornita per la “validazione” della rete, ovvero per valutare se la rete si è allenata nel modo corretto; in questa fase chiamata appunto “validation”, la rete effettua i cicli su questi nuovi dati con la possibilità, però, di andare a cambiare gli iperparametri della rete attraverso la backward propagation.

Infine, è presente la fase di “testing” che si differenzia dalle due precedenti perché elabora i dati, effettua delle predizioni, ma queste ultime non vengono messe a confronto con nessun dato corretto e non è presente la fase di backward propagation.

A questo punto la rete ha completato tutte le fasi necessarie per l'allenamento ed è pronta ad essere utilizzata.

Due problemi che possono verificarsi e a cui bisogna prestare particolare attenzione durante la realizzazione di una rete neurale, sono:

- Underfitting
- Overfitting

3.1.3 Underfitting

Il fenomeno dell'underfitting è un problema di apprendimento che consiste nella scarsa capacità della rete di fare previsioni accurate nella fase di allenamento, e di conseguenza scarse previsioni in fase di test; questo può essere dovuto da un'architettura troppo semplificata.

Per poter risolvere il fenomeno descritto è necessario rendere più complessa la costruzione del modello utilizzato.

3.1.4 Overfitting

L'overfitting, invece, è il problema inverso in quanto si verifica quando la previsione si basa su troppi parametri e quindi la rete è troppo complessa. In questo caso il modello diventa troppo sensibile ai dati di addestramento, il che produce bassi errori in fase di addestramento, ma in fase di test l'algoritmo non sarà in grado di adattarsi ai nuovi dati.

Mediante la semplificazione dell'architettura è possibile risolvere l'overfitting in una NN.

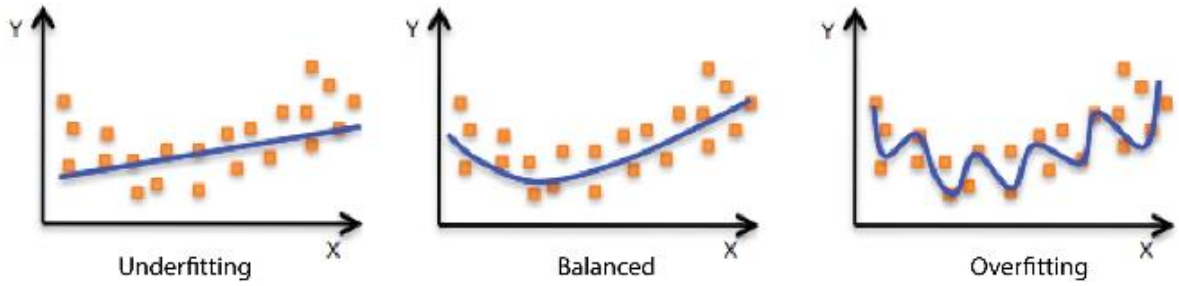

Figura 7: Raffigurazione problema di underfitting e overfitting[3]

[3] Informazioni riguardati underfitting e overfitting disponibili qui: https://docs.aws.amazon.com/it_it/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html (data: 30/09/2023)

3.2 LSTM

Una rete neurale LSTM è di base una recurrent neural network (RNN) strutturata, però, in maniera più complessa.

Queste reti sono state create per risolvere il problema che affligge comunemente le reti neurali ricorrenti, ovvero la scomparsa del gradiente ("vanishing gradient problem" in inglese) che consiste nell'assunzione di un valore nullo o asintotico allo zero da parte del gradiente, impedendo così l'aggiornamento dei pesi della rete.

Per capire la differenza tra RNN e LSTM, partiamo dal descrivere l'architettura di una rete neurale ricorrente.

Una RNN è una classe di rete neurale che è formata da neuroni collegati tra loro in un ciclo, perciò i valori in uscita da un layer, sono portati in ingresso al livello precedente. Questa interconnessione tra i vari layer, permette alla rete di acquisire una "memoria di stato" che rende le predizioni successive dipendenti non solo dal dato ricevuto in input dalla rete in quell'istante, come in qualsiasi altra rete neurale, ma anche dalla predizione effettuata all'istante precedente.

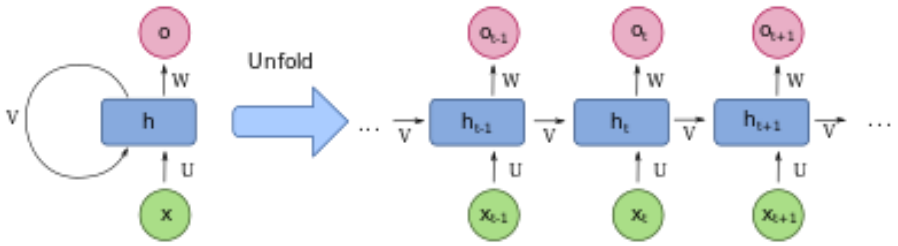

Figura 8: Architettura RNN

È per questo motivo che le reti neurali ricorrenti sono le più indicate per problemi di natura temporale, ovvero per il processing e predizione dei dati basati su serie temporali come, ad esempio, problemi di traduzione linguistica, riconoscimento vocale, NLP (Natural Language Processing), meteorologia, economia, e molti altri ancora.

Analizzando, invece, la struttura di una rete Long-Short Term Memory, possiamo subito delineare una differenza importante, ovvero che questa rete ha una struttura a "cella", chiamata così perché all'interno di una singola cella sono presenti quattro sezioni differenti per l'elaborazione dei dati in input:

- Forget Gate: la funzione di questo primo passaggio è quella di decidere se tenere o dimenticare l'informazione in ingresso al gate, che è composta dalla parte proveniente dall'hidden layer precedente e quella derivante dal nuovo input. Questi dati passano

attraverso una funzione sigmoidea, che scarta il dato se ha un valore che si avvicina a zero, oppure lo tiene se ha un valore più vicino ad uno;

- Input Gate: questo gate contribuisce alla fase di aggiornamento della Cell State. L'input alla cella e le informazioni dello stato precedente passano attraverso una funzione sigmoidea da un lato, e dall'altro attraverso una funzione tangente iperbolica, e i risultati delle relative computazioni vengono moltiplicati per poi essere indirizzati verso la Cell State. L'utilità di moltiplicare insieme gli output delle due funzioni, è che grazie alla funzione sigmoidea si possono identificare le informazioni importanti in uscita dalla funzione tanh;

- Cell State: grazie ai passaggi precedenti, ora possiamo andare a calcolare lo stato attuale della cella. Come prima operazione viene moltiplicato lo stato della cella dell'iterazione precedente con l'output ottenuto in questa iterazione dal Forget Gate, successivamente, il risultato viene sommato con l'output del Input Gate. Questo risultato andrà, poi, a costituire lo stato della cella definitivo di questa iterazione che corrisponderà all'input per il calcolo del Cell State dell'iterazione successiva;

- Output Gate: l'hidden state per l'iterazione successiva viene definito in quest'ultimo passaggio. Per trovare le informazioni dell'hidden state dobbiamo andare ad applicare una funzione sigmoidea all'input e all'hidden state dell'iterazione precedente, come è stato fatto anche per il Forget Gate e per l'Input Gate, solo che ora il risultato va moltiplicato con l'output proveniente dal Cell State, a cui è stato precedentemente applicata una funzione tangente iperbolica. L'output di questa computazione finale determinerà l'hidden state che entrerà nella cella all'iterazione dopo.

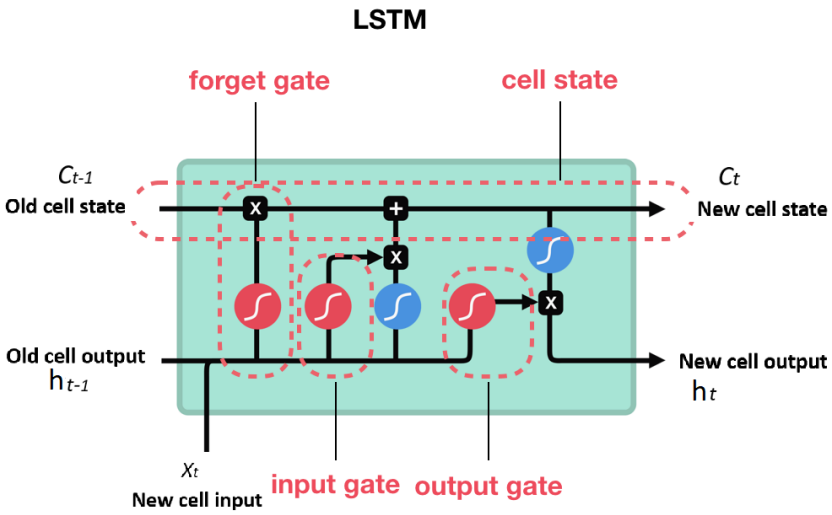

Figura 9: Cella di rete neurale LSTM[4]

Nella Figura 9, sono contrassegnate in fucsia le funzioni sigmoidee (σ), e in azzurro le funzioni di tangente iperbolica (tanh).

[4] "Beginner's Guide to RNN & LSTMs" disponibile qui: https://medium.com/@humble_bee/rnn-recurrent-neural-networks-lstm-842ba7205bbf (data: 30/09/23/2023)

4. Bennu

Tutte le simulazioni nel codice MATLAB dell'EKF sono state effettuate utilizzando l'asteroide Bennu come corpo centrale attorno al quale il CubeSat, oggetto dello studio, stava orbitando.

Bennu, la quale denominazione ufficiale è "101955 Bennu", è un piccolo asteroide near-Earth dal diametro medio di circa 500 m che ogni sei anni circa effettua un passaggio orbitale nelle vicinanze della Terra.

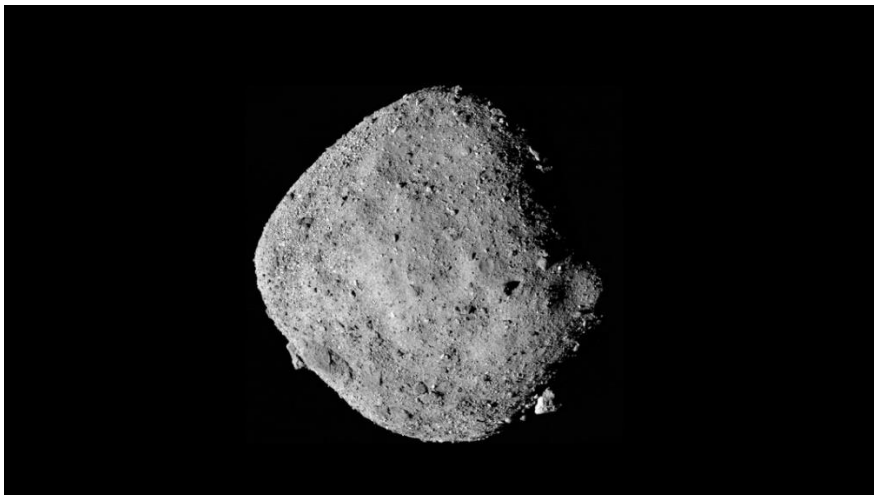

Figura 10: Asteroide 101955 Bennu[5]

Nonostante sia stato scoperto nel 1999, il nome ufficiale, che richiama la divinità egizia, gli fu assegnato solamente nel 2013 quando fu scelto come obiettivo della missione OSIRIS-Rex della NASA, il cui lancio è avvenuto l'8 settembre 2016 alle ore 23:05 UTC dal Launch Complex 41 del Cape Canaveral Air Force Station a bordo del lanciatore United Launch Alliance Atlas V 411.

La missione mirava a:

- Raccogliere un campione di regolite incontaminata dalla superficie
- Mappare le proprietà globali, chimiche e mineralogiche di un asteroide primordiale carbonaceo
- Documentare la distribuzione, la morfologia, la geochimica e le proprietà spettrali della regolite al sito di prelievo

[5] Informazioni e foto riguardanti Bennu disponibili qui: https://solarsystem.nasa.gov/asteroids-comets-and-meteors/asteroids/101955-bennu/overview/ (data: 30/09/2023)

- Misurare l'effetto YORP di un asteroide potenzialmente pericoloso. L'effetto YORP è una variazione indotta dalla radiazione solare sull'orbita e sulla rotazione degli asteroidi

La missione della NASA ha effettuato il prelievo di 250 grammi di materiale dalla superficie dell'asteroide il 20 ottobre 2020 alle ore 20:12 UTC, ed è terminata il 24 settembre 2023 alle 14:53 UTC riportando con successo i campioni sul suolo terrestre.

Figura 11: Capsula contenente i sample della missione NASA OSIRIS-Rex

Bennu, inoltre, fa parte del gruppo Apollo, un insieme di asteroidi near-Earth il cui principale asteroide che dà il nome al gruppo è stato scoperto il 24 aprile 1932 da un astronomo tedesco, Karl Reinmuth. All'interno di questa classe di asteroidi sono compresi oggetti potenzialmente pericolosi per la Terra a causa della possibilità di un impatto catastrofico.

Per concludere la descrizione di 101955 Bennu, elenchiamo le sue caratteristiche principali.

Tabella 1: Elenco caratteristiche orbitali e fisiche dell'asteroide Bennu

PARAMETRI ORBITALI	
Semiasse maggiore	168'434'580 km
Periodo orbitale	436,36 giorni (1,19 anni)
Inclinazione sull'ellittica	6,03428°
Argomento del perielio	66,30371°
Anomalia media	87,64100°
DATI FISICI	
Dimensioni	565 m x 535 m x 508 m
Diametro medio	0,492 km
Superficie	0,7874 km^2
Volume	0.061354 km^3
Periodo di rotazione	4h 17min 16s

5. Attività svolte su MATLAB

5.1 Simulatore EKF e codici MATLAB

La prima parte del lavoro di tesi è stata svolta su un algoritmo sviluppato in ambiente MATLAB, realizzato per simulare un computer di navigazione che stima la traiettoria di un CubeSat orbitante intorno a Bennu. Tale simulatore combina le osservabili ottiche e la dinamica dell'ambiente attorno all'asteroide, all'interno di un filtro di navigazione.

Sono state svolte simulazioni con diversi tipi di modelli dinamici per indagare il funzionamento del filtro in situazioni differenti con l'obbiettivo di trovare i valori dei parametri del filtro che minimizzino l'errore sulla predizione della traiettoria a seguito del processo di una data osservabile.

Prima di approfondire quanto svolto, andiamo a descrivere come è composto il codice che simula l'Extended Kalman Filter.

Il codice è costituito da un file principale, dove si trova il computer di navigazione (compreso il filtro) e da diversi file secondari contenenti diversi tipi di funzioni, come aprire file contenenti dati astronomici e svolgere tutte le operazioni che non sono direttamente collegate all'attività di tuning del filtro.

La struttura del file principale può sostanzialmente essere divisa in tre parti:

- Una prima parte di inizializzazione, nella quale vengono definite le variabili locali e globali, il sistema di riferimento utilizzato, il modello dinamico, e viene aggiunto alla traiettoria usata come riferimento del rumore bianco per simulare i dati in ingresso;

- Una parte centrale che rappresenta il cuore del filtro, dove sono definiti tutti i suoi parametri, come, ad esempio, il Kalman Gain, la matrice di propagazione dello stato, la matrice dell'incertezza sul modello dinamico, ecc. In questa parte, avviene anche il processo di stima della traiettoria, che si divide nelle due fasi di propagazione della traiettoria, effettuato usando il metodo di Runge-Kutta di quinto ordine, e la fase di correzione del filtro che si alternano ciclicamente in base alle osservabili ricevute in input;

- Infine, l'ultima parte è dedicata alla creazione dei plot contenenti i risultati più importanti da visualizzare e successivamente salvare.

Il codice ha subito alcune piccole variazioni incentrate sulla struttura dello stesso, mantenendo però, invariato il funzionamento del filtro. Oltre alle attività di test e tuning, che consiste nel trovare i valori

della matrice di covarianza ottimali tali da minimizzare l'errore, si è infatti lavorato al fine di migliorare l'organizzazione del codice, andando ad automatizzare il processo di inizializzazione delle variabili del SW, in quanto alcune di esse prima era necessario inserirle "hard coded", ovvero nei punti del codice dove venivano utilizzate, senza inizializzazione a monte. Questo ha permesso una maggiore chiarezza e controllo delle variabili all'interno del programma.

Infine, per quanto concerne l'attività di modifica del codice originale, si è provveduto a calcolare le matrici di propagazione dello stato che sarebbero state usate per i nuovi modelli dinamici in previsione dei test effettuati in seguito.

Successivamente, sono stati sviluppati due ulteriori codici in MATLAB per supportare la nostra ricerca. Il primo codice è stato creato con l'obiettivo di generare nuove traiettorie che possono essere personalizzate attraverso uno script. In particolare, queste traiettorie differiscono dalle precedenti per la loro estensione temporale, che è stata ampliata fino ad una durata di qualche giorno. Le traiettorie generate sono state poi utilizzate come input per il secondo codice.

Il secondo codice ha il compito di aggiungere rumore bianco con media nulla alle traiettorie precedentemente create. Questo passaggio è cruciale poiché ciò che otteniamo sono delle osservabili sintetiche che costituiranno il dataset utilizzato per addestrare e testare la rete neurale. È importante notare che le reti neurali richiedono dataset ampi e diversificati per un addestramento efficace.

In questo secondo codice, è possibile specificare il numero di osservabili da generare e l'intervallo di deviazione standard a cui queste osservabili saranno soggette. Ad esempio, è possibile generare 1000 osservabili diverse, a cui viene associato del rumore bianco con un valore di std scelto nell'intervallo [0, 100]. Questa flessibilità consente di creare un dataset di addestramento e test su misura per le esigenze del nostro modello di NN.

5.2 Modelli dinamici

Le tipologie di prove nelle quali si sono maggiormente concentrati i nostri test in questa fase preliminare sono state la stima orbitale attraverso il modello dinamico del punto materiale con l'aggiunta di effetti gravitazionali secondari, quali:

- SRP
- SPH
- SRP + SPH

Queste due famiglie di test sono state determinanti per capire il funzionamento alla base del filtro.

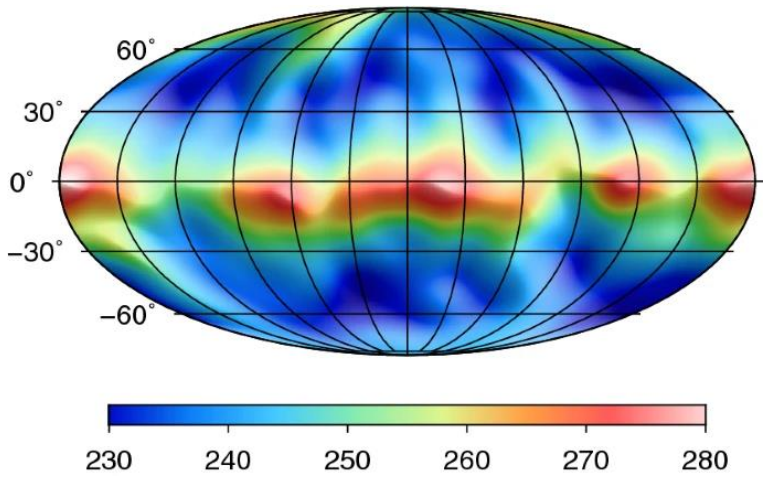

Figura 12: Rappresentazione delle armoniche sferiche sull'asteroide Bennu [3]

Nel modello point mass sono già tenuti in considerazione gli effetti del terzo corpo agenti sullo spacecraft dovuti dalla presenza del Sole e di Giove, dato che sono i corpi celesti che hanno un'influenza maggiore nella dinamica del CubeSat.

Si sono effettuati dei test preliminari per determinare l'accuratezza del filtro quando nel modello viene propagato l'effetto di SRP, SPH o ALL, ma utilizzando la matrice Jacobiana $\boldsymbol{F}(\boldsymbol{t})$ che tiene conto solamente del modello PM. Questo è stato fatto per studiare il margine di miglioramento del filtro una volta implementata la matrice $\boldsymbol{F}$ riferita al modello dinamico utilizzato.

$$F = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -4.89143 * \frac{R^2-3*x^2}{R^5} & -4.89143 * \frac{-3*x*y}{R^5} & -4.89143 * \frac{-3*x*z}{R^5} & 0 & 0 & 0 \\ -4.89143 * \frac{-3*x*y}{R^5} & -4.89143 * \frac{R^2-3*y^2}{R^5} & -4.89143 * \frac{-3*y*z}{R^5} & 0 & 0 & 0 \\ -4.89143 * \frac{-3*x*z}{R^5} & -4.89143 * \frac{-3*y*z}{R^5} & -4.89143 * \frac{R^2-3*z^2}{R^5} & 0 & 0 & 0 \end{pmatrix} \quad (15)$$

Matrice F del modello PM

$$GM = \mu * MBennu = 4.89143\ [m^3 s^{-2}]$$
$$\mu = 6.6743 * 10^{-11}\ \left[m^3\ kg^{-1}\ s^{-2}\right]$$
$$MBennu = (7.329 \pm 0.009) * 10^{10}\ \left[kg\right] \quad (16)$$

Con μ = costante di gravitazione universale, e $MBennu$ = Massa dell'asteroide Bennu.

Per quanto concerne la costruzione della matrice $\boldsymbol{F}$ che descrive anche gli effetti di SRP o SPH, questa sarà una matrice 7x7 aventi nelle posizioni $\boldsymbol{F}(4,7)$, $\boldsymbol{F}(5,7)$ e $\boldsymbol{F}(6,7)$ i contributi delle relative accelerazioni SPH o SRP, in base al modello con il quale si effettuano le simulazioni; di conseguenza la matrice del modello completo sarà una 8x8 contenente entrambe le accelerazioni.

Un altro parametro fondamentale del filtro che varia al variare del modello simulato e sul quale si è concentrato il processo di tuning, è la matrice $\boldsymbol{Q}$ che descrive l'incertezza che si ha nei confronti del modello dinamico adottato; in particolar modo è necessaria una dimensione 6x6 per la matrice che tiene conto solamente degli effetti di PM, con i primi tre valori nella diagonale che descrivono l'incertezza quadratica sulla posizione $[m^2]$ e gli ultimi tre valori sulla diagonale che descrivono l'incertezza quadratica sulla velocità $[m^2\ s^{-2}]$.

La matrice $\boldsymbol{Q}$ viene presa diagonale nel nostro studio, semplificando il problema in fase preliminare mediante la descrizione di ogni parametro come indipendente dagli altri, ma generalmente essa non è diagonale, visto che esiste una correlazione tra i vari termini.

Dopo varie simulazioni si sono trovati i valori ottimali per la matrice $\boldsymbol{Q}$, che corrispondono a 10^{-09} per l'incertezza sulla posizione e 10^{-12} per l'incertezza sulla velocità; l'utilizzo di questi valori permette al filtro di effettuare le stime con un livello di accuratezza più alto quando viene utilizzato il modello PM. Questi valori sono quindi stati presi come riferimento per le simulazioni con i modelli dinamici più complessi (quali SRP, SPH e ALL).

Per le simulazioni effettuate con i differenti modelli dinamici si è deciso di utilizzare delle osservabili con una standard deviation di 100 m, mentre il valore tenuto in considerazione nella matrice di covarianza del rumore sulle misure è stato sottostimato, per verificare il comportamento del filtro.

Prima di esaminare i risultati trovati, è importante introdurre due matrici che hanno un ruolo chiave nelle simulazioni:

- $\boldsymbol{Q}$: matrice di covarianza del disturbo sullo stato. Rappresenta la variabilità statistica del valore dei disturbi sullo stato
- $\boldsymbol{R}$: matrice di covarianza del rumore sulle misure. Rappresenta la variabilità statistica del vettore dei disturbi di misura

5.2.1 SRP

Il primo modello dinamico che siamo andati ad analizzare è quello che tiene conto della SRP (Solar Radiation Pressure).

Come anticipato precedentemente, il primo test effettuato è stato quello in cui abbiamo la SRP nelle osservabili ma non all'interno del modello dinamico propagato, con la matrice $\boldsymbol{Q}$ che tiene conto solo del modello dinamico PM. Abbiamo utilizzato i valori della matrice trovati in precedenza, quindi 10^{-09} per l'incertezza sulla posizione e 10^{-12} per l'incertezza sulla velocità, calcolando la differenza tra i valori stimati e quelli "reali" si è trovato un RMS^2 = 47,63 m^2.

Proseguendo con la simulazione rifatta mediante l'uso della matrice $\boldsymbol{Q}$ che tiene conto della Solar Radiation Pressure, sono stati effettuati diversi tentativi prima di trovare i valori che hanno dato il risultato con l'errore più basso.

I valori che sono stati trovati sono 10^{-09} per l'incertezza sulla posizione, 10^{-14} per la velocità e 10^{-12} per l'accelerazione data dalla radiazione solare.

Includendo nella matrice di transizione anche la dinamica relativa alla Solar Radiation Pressure, il valore medio della RMS^2 si è abbassato notevolmente arrivando ad un valore di 33,79 m^2; ciò delinea una precisione maggiore da parte del filtro di navigazione che riesce a stimare con più precisione quella che è l'orbita "vera".

Per comprendere il lavoro effettuato dal filtro durante una simulazione, nel grafico a pagina successiva (Figura 13) è raffigurato l'andamento dell'errore medio sulla posizione nelle tre coordinate spaziali con le due bande laterali che rappresentano il valore di std al variare del tempo.

La linea rossa con la banda azzurra raffigura l'errore medio e la std facendo uso della matrice $\boldsymbol{Q}$ che propaga il modello dinamico PM, mentre la linea blu con la banda rosa raffigura la $\boldsymbol{Q}$ SRP.

Si può notare come il modello SRP sia generalmente più preciso rispetto a quello PM.

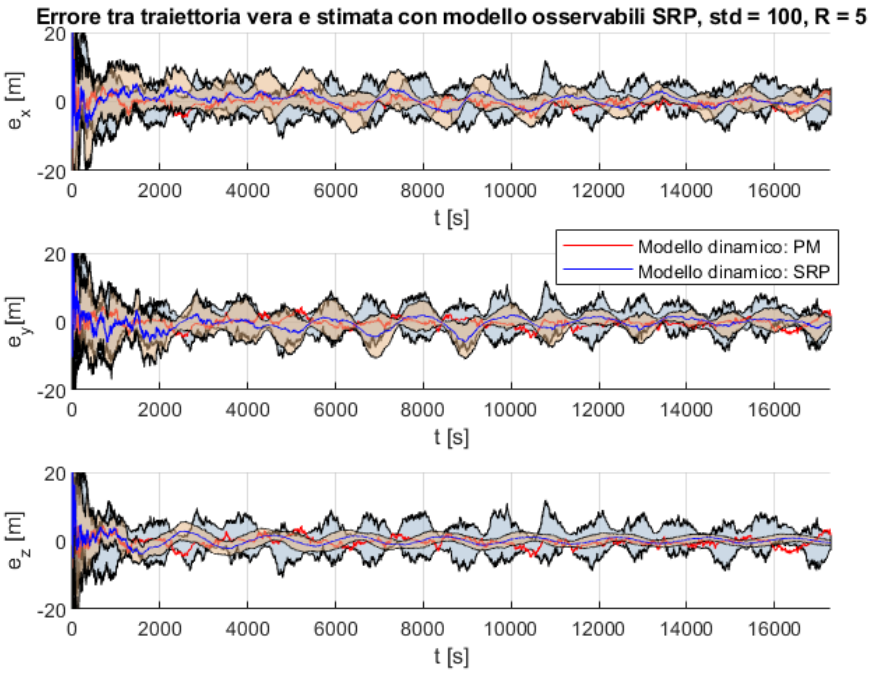

Figura 13: Stime sulla traiettoria effettuate con la **Q** dei diversi modelli dinamici.
Le linee rappresentano l'errore medio sulla posizione per ogni coordinata spaziale, mentre le due bande laterali raffigurano il valore di std al variare del tempo.

Per poter apprezzare in maniera migliore la differenza negli andamenti della linea media, si può plottare il grafico precedente senza raffigurare le std dei risultati, che rappresentano la deviazione dal valore medio dell'errore delle simulazioni durante i diversi cicli iterativi, che quindi non è correlata alla std delle osservabili in ingresso al filtro di navigazione descritta nel titolo del plot:

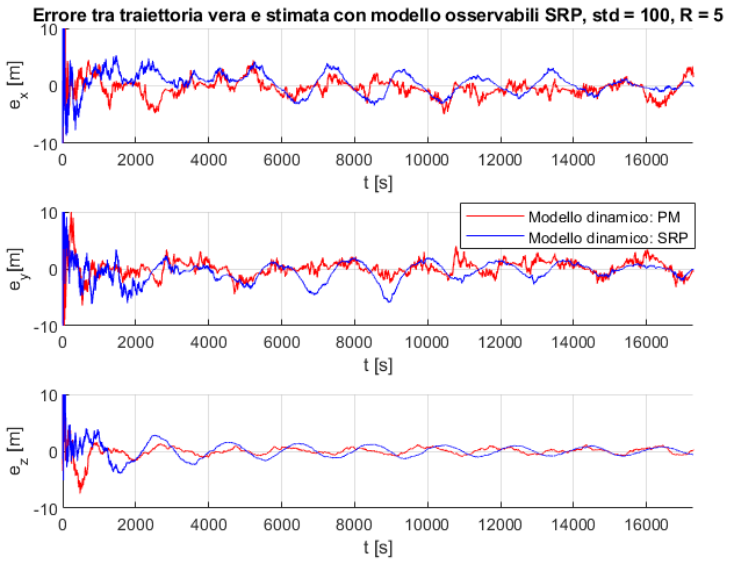

Figura 14: Errore medio sulla posizione con la **Q** dei diversi modelli dinamici

Infine, un altro plot interessante per il confronto dei due modelli è quello raffigurante la Residual Sum of Squares (RSS), dal quale si può osservare come, soprattutto nella parte finale della simulazione, il modello SRP assuma valori più piccoli rispetto al PM:

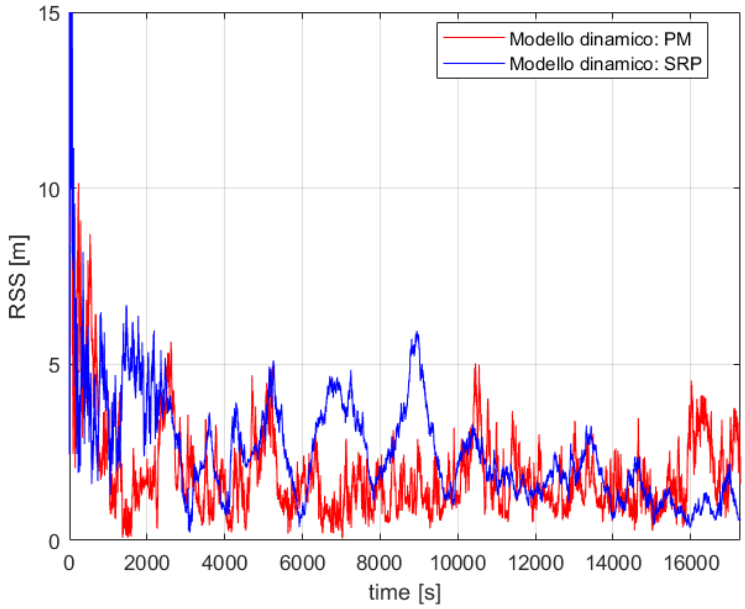

*Figura 15: Andamento RSS della **Q** PM e SRP*

5.2.2 SPH

Per quanto concerne le simulazioni con le osservabili comprendenti l'effetto SPH (Spherical Harmonics), partendo sempre dalla $\boldsymbol{Q}$ che propaga la dinamica PM, il risultato migliore ottenuto è stato quello di RMS^2 = 46,17 m^2.

Andando a valutare i risultati ottenuti, invece, mediante l'uso di $\boldsymbol{Q}$ con il modello SPH, che ha dato i risultati migliori nella configurazione:

$$\boldsymbol{Q} = \begin{pmatrix} 10^{-09} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10^{-09} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10^{-09} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^{-13} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10^{-13} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10^{-13} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10^{-10} \end{pmatrix} \tag{17}$$

si è potuto notare che, anche in questo caso, c'è stato un netto miglioramento nel valore del RMS^2 medio, il quale ha fatto registrare un valore di 35,72 m^2.

Valutando l'andamento durante la simulazione dei due modelli di matrice $\boldsymbol{Q}$ nei due seguenti grafici raffiguranti l'errore medio nella posizione e la std per la Figura 16 con l'errore medio messo in risalto nella Figura 17, si apprezza come anche in questo caso l'utilizzo del modello più completo porti a dei risultati più accurati

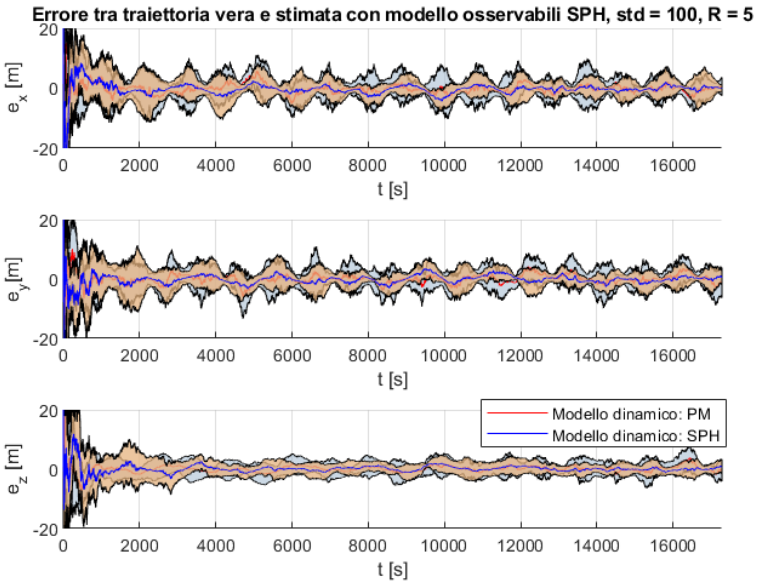

Figura 16: Stime sulla traiettoria effettuate con la **Q** dei diversi modelli dinamici.
Descrizione approfondita in *Figura 13*

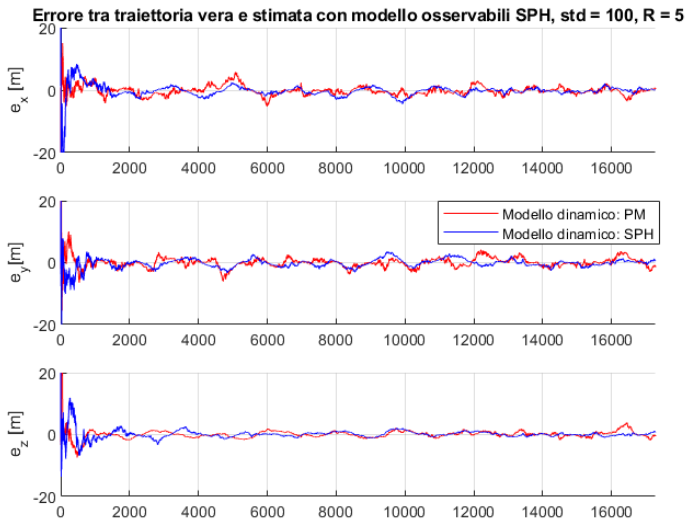

Figura 17: Errore medio sulla posizione con la **Q** dei diversi modelli dinamici

Risultato che si può apprezzare in modo più chiaro dal grafico RSS:

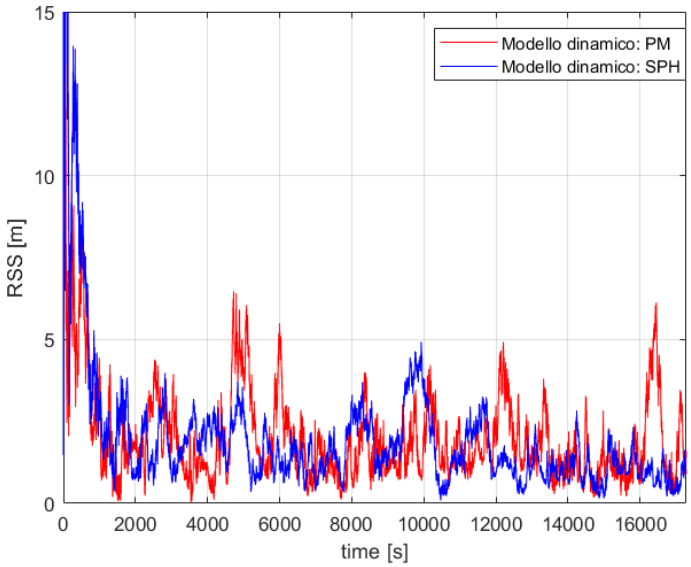

Figura 18: Andamento RSS della ***Q*** *PM e SPH*

5.2.3 ALL

Per concludere i test sul filtro di navigazione sono rimasti da esaminare i risultati relativi alle osservabili ALL, che includono entrambi gli effetti di SRP e SPH.

Partendo dalla matrice $\boldsymbol{Q}$ rappresentante il modello PM, il valore di RMS^2 migliore ottenuto è stato di 44,21 m^2, utilizzando i medesimi valori nella diagonale della matrice (10^{-09} e 10^{-12}).

Mentre la matrice $\boldsymbol{Q}$ del modello ALL che ha riportato il livello di accuratezza maggiore è composta da 10^{-09} Sull'incertezza della posizione, 10^{-13} per la velocità, 10^{-12} per accelerazione SRP ed infine 10^{-10} per accelerazione SPH, riuscendo ad ottenere un RMS^2 = 36,29 m^2.

Finiamo la nostra analisi andando a plottare, anche per questa simulazione, i tre grafici raffiguranti errore medio e std (Figura 19), errore medio isolato (Figura 20) ed infine il grafico RSS (Figura 21).

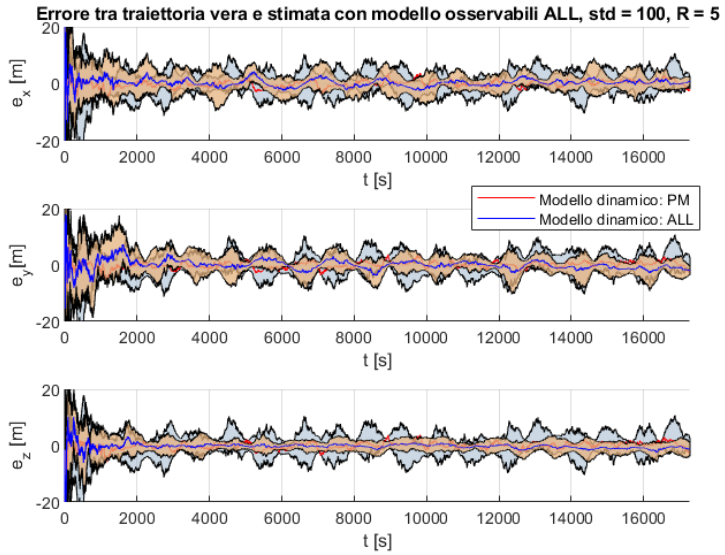

Figura 19: Stime sulla traiettoria effettuate con la **Q** dei diversi modelli dinamici.
Descrizione approfondita in *Figura 13*

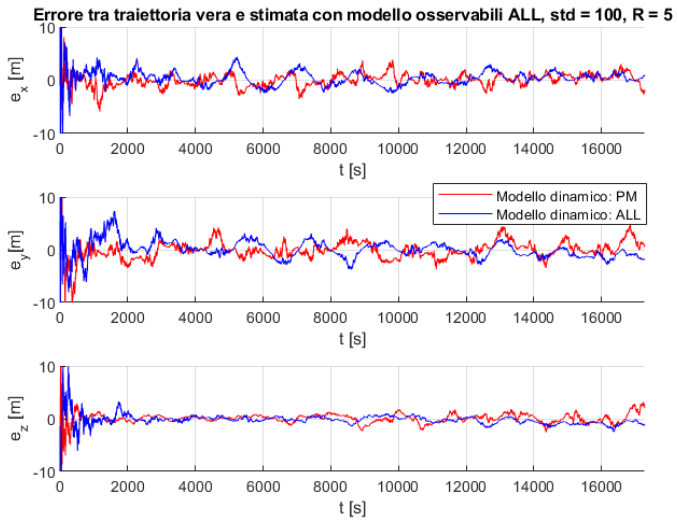

Figura 20: Errore medio sulla posizione con la **Q** dei diversi modelli dinamici

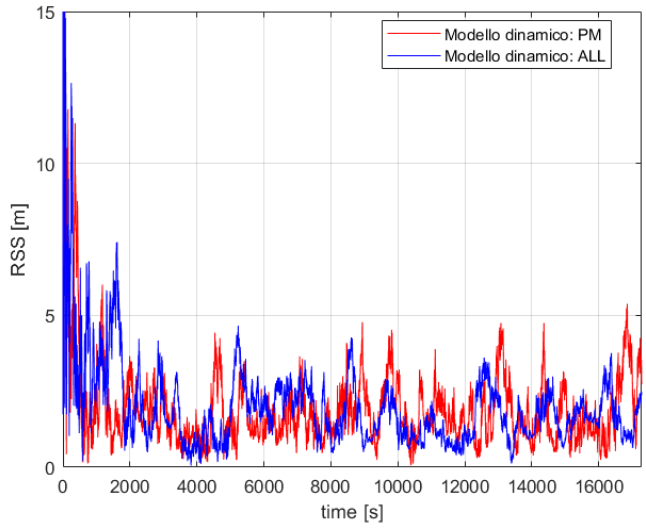

Figura 21: Andamento RSS della ***Q*** *PM e ALL*

Dalle tre simulazioni si può notare come con l'aumentare della complessità del modello dinamico adottato si ha un miglioramento via via meno marcato rispetto alle simulazioni con la matrice $\boldsymbol{Q}$ PM, ma nonostante questo, con un corretto tuning del filtro, è possibile avere una stima della traiettoria a partire dalle osservabili più precisa.

6. Attività di sviluppo della rete neurale

6.1 Configurazione dell'ambiente Python

Per lo sviluppo della rete neurale si è utilizzato il linguaggio di programmazione Python, in quanto rappresenta uno dei linguaggi più utilizzati nell'ambito dell'intelligenza artificiale, con la presenza di librerie sviluppate ad hoc per questo tipo di applicazioni.

Ciò che contraddistingue Python rispetto ad altri linguaggi di programmazione, sono la velocità, la semplicità di utilizzo, e la portabilità, ovvero la compatibilità con praticamente ogni sistema; per questo motivo è scelto per le più svariate applicazioni quali programmazione GUI (Graphical User Interface), sviluppo web, creazione di script, analisi dati, e machine learning.

Il linguaggio di programmazione da solo non basta, ma è necessario creare un ambiente di sviluppo completo se si vuole progettare una NN; per questo motivo è stato necessario implementare i seguenti tool addizionali, indispensabili per la realizzazione del progetto.

6.1.1 Conda

Conda è un sistema open source per la gestione di pacchetti e creazione di ambienti virtuali in Python. L'utilità principale di Conda è la possibilità di avere installate diverse versioni di Python nei diversi ambienti virtuali (environments) senza che si creino conflitti; per ogni environment, poi, è possibile installare solo i pacchetti di proprio interesse.

Esistono due versioni di Conda:

- Anaconda
- Miniconda

Anaconda è la versione più corposa e pesante in termini di pacchetti standard installati, comprende di base oltre 1500 pacchetti scientifici e non è necessario dunque andarli ad installare individualmente, mentre Miniconda è la versione "lightweight" di Conda che non ha pacchetti preinstallati se non quelli strettamente necessari.

Per la creazione dell'ambiente virtuale della rete neurale è stato scelto l'utilizzo di Miniconda in quanto i pacchetti utilizzati per la creazione della rete sarebbero stati poco numerosi, così da risparmiare in termini di memoria.

Mediante l'uso di questo programma è stato possibile anche installare ed operare Jupyter Notebook, uno strumento nato per il data science che permette di creare dei file aventi diverse celle di codice separato, in modo tale da poter eseguire ogni cella in modo indipendente e nell'ordine desiderato.

Questo sistema è utile quando si ha la necessità di analizzare le diverse parti del codice singolarmente, soprattutto a fronte di numerose modifiche, come accade, ad esempio, durante il processo di creazione di una rete neurale.

6.1.2 Visual Studio Code

Visual Studio Code è un editor di codice sorgente molto potente che integra un sistema di debugging, IntelliSense (ovvero un sistema atto al completamento automatico delle istruzioni al fine di velocizzare il processo di scrittura del codice), controllo Git (per lavorare direttamente sui repository di GitHub), e inoltre offre molte estensioni grazie alle quali è possibile ampliare in modo significativo le funzionalità offerte dal programma.

A monte delle simulazioni effettuate su Google Colab, di cui parleremo in seguito, si è utilizzato Visual Studio Code per la stesura dei codici. In questa maniera è risultato più veloce il processo di scrittura e modifica della rete, data la possibilità di effettuare il debugging delle porzioni di codice interessate da errori; una volta completata la rete neurale e testato il suo funzionamento, questa veniva importata su Google Colab insieme al dataset per velocizzare il processo di training.

6.1.3 PyTorch

Per la creazione di una rete neurale ci sono diverse librerie come TensorFlow, Keras, o frameworks come PyTorch. Nello specifico, è stato utilizzato il framework PyTorch per la stesura del codice della NN, questo perché, nonostante una complessità iniziale maggiore nella fase di apprendimento del linguaggio, successivamente risulta più semplice il processo di debugging rispetto a TensorFlow.

6.1.4 Google Colab

Infine, l'ultimo tool utilizzato come ausilio per la creazione della rete, è stato Google Colab.

Google Colab è una piattaforma in cloud sviluppata appositamente per il machine learning, che offre una notevole potenza di calcolo attraverso le GPU (graphics processing unit) messe a disposizione gratuitamente da Google. Per sfruttare questo servizio, che frutta di default Python con Jupiter Notebook, è sufficiente importare il proprio codice con i file e il dataset corrispondenti, implementare nel codice il tool Cuda di Nvidia, tramite il quale sarà poi possibile indirizzare le variabili e il modello della rete neurale nella GPU Tesla T4 che viene fornita con il pacchetto gratuito (altrimenti di default l'esecuzione avverrebbe nelle CPU sempre messe a disposizione da Google), e una volta finita questa fase preliminare l'ambiente è configurato e pronto all'utilizzo.

L'unica limitazione di Google Colab è che, nella versione gratuita, è possibile utilizzare le GPU per un tempo limitato di 12 ore al giorno, le quali vengono ridotte man mano se si fa un uso frequente del

servizio, e contemporaneamente aumenta il tempo di attesa prima di poter riutilizzare le GPU. Questo nel tempo può portare ad un rallentamento importante dei test.

6.2 Studio preliminare e scelta della rete

La realizzazione di una rete neurale non è un processo lineare, si hanno diverse possibilità tra cui scegliere e non esiste una soluzione unica ad un dato problema poiché non esiste un solo modo per risolverlo od una sola architettura di NN che possa svolgere un determinato compito.

Per questo motivo è stata necessaria un'approfondita attività di ricerca per comprendere in primis il principio di funzionamento su cui si basa una rete neurale, e poi per cercare di capire tra le tante architetture esistenti quale fosse la più calzante al nostro problema.

Le reti neurali più comuni sono le convolutional neural networks (CNN), spesso usate per problemi di classificazione, riconoscimento delle immagini, rilevamento degli oggetti, riconoscimento facciale e così via; in queste applicazioni le CNN hanno alte performance, ma sono meno indicate per i casi in cui si ha la necessità di analizzare dati sequenziali e temporali, che invece interessano il nostro ambito di sviluppo. Per questo motivo si è deciso di scartarle in quanto si è ritenuto non rappresentassero il tipo di architettura più adatto.

Tra le altre architetture la scelta è ricaduta sulle recurrent neural networks (RNN), dato che sono appositamente studiate per i problemi dipendenti dal tempo, come visto anche nel 3.2, in particolar modo si è deciso di optare per una rete Long-Short Term Memory, ovvero la versione migliorata della RNN che punta a risolvere i problemi di vanishing gradient e di exploding gradient; inoltre, le reti LSTM performano meglio per pattern temporali anche molto lunghi grazie all'introduzione della struttura a cella e questo si sposa perfettamente con l'obbiettivo della rete neurale che è stata sviluppata per la realizzazione di questa tesi.

6.3 Il codice della rete neurale

Lo scopo della rete neurale sviluppata è quella di stimare il rumore presente all'interno delle osservabili fornite al filtro di navigazione.

In questa sezione si descrive tutto il codice della rete in modo dettagliato per capirne il funzionamento.

Il primo passaggio è quello che vede l'importazione di tutte le librerie che saranno poi utilizzate durante l'esecuzione dello script, come la libreria per importare il modulo di PyTorch, quella per creare il dataset con cui dovrà allenarsi la rete neurale, quella per il caricamento del dataset, ecc.

Successivamente avviene la fase di controllo della GPU, ovvero si verifica che ci sia la disponibilità di una GPU nella quale andare a caricare il modello della rete e le variabili per aumentare la potenza di calcolo (controllo eseguito indipendentemente dal fatto che il codice venga eseguito in locale o su Colab), in caso contrario viene eseguito sulla CPU.

Dopodiché vengono caricate tutte le osservabili generate dal codice MATLAB descritto nel 5.1 con i relativi valori di std. Prima di creare il dataset, vengono normalizzati i valori di std nell'intervallo [0,1].

- I dati usati per il training, insieme ai corrispettivi label, vengono poi divisi nei subset di training (solitamente l'80% dei dati totali) e testing (il rimanente 20%) (vedi Codice 1).

Viene, inoltre, impostato il valore di batch size, che determina il numero di istanze che la NN deve processare nella stessa iterazione del ciclo, in modo tale da velocizzare il processo di allenamento dato che il database comprende una grande quantità di osservabili generate. In questo modo la rete neurale lavorerà con un numero di osservabili pari al batch size, e, di conseguenza, restituirà in output tante istanze quante quelle fornite all'ingresso.

Per evitare di allenare la rete dando in input le osservabili sempre nello stesso ordine, si utilizza la funzione di "shuffle" che permette di mescolare i dati in input.

Proseguendo con il codice, si trova la parte relativa al corpo della rete neurale (vedi Codice 2), costituita da una parte di inizializzazione e da una parte che esegue la forward propagation.

Nella parte di inizializzazione vengono definiti gli iperparametri della rete, quali numero di parametri in input, numero dei neuroni per hidden layer, e numero di parametri in output. Gli input sono 3 in quanto si ha un tensore in ingresso alla rete composto come segue

$$[batch\ size, lunghezza\ osservabile\ in\ secondi, coordinate\ osservabile\ per\ ogni\ secondo]$$

mentre l'output ha dimensione 1 perché in uscita dalla classe avremo un singolo vettore di lunghezza $[batch\ size]$, dato che vogliamo la predizione della std per ogni osservabile che viene processata in un determinato ciclo.

Sempre nella parte di inizializzazione vengono create le parti della rete neurale che verranno poi utilizzate nella fase di forward, come i layer LSTM, in cui viene specificato anche il numero, la percentuale di dropout (ovvero quanti neuroni per strato vengono disattivati ad ogni iterazione, questo aiuta l'apprendimento della rete e riduce il problema dell'overfitting), e i layer lineari, ognuno con la propria dimensione di input e output. Per completare la parte di inizializzazione c'è una funzione che serve a resettare i pesi della rete, così per ogni nuovo allenamento non rimangono in memoria i pesi dell'allenamento precedente.

Successivamente viene definita la sezione di forward, che comprende come primo passaggio l'inserimento dei dati ricevuti in input dalla rete nelle celle LSTM, che li processano come visto nel 3.2; lo step seguente prevede di prendere l'ultima predizione effettuata dalle celle per ogni relativa osservabile e darle in input al primo layer lineare. I layer lineari hanno un numero che può essere variabile e che viene deciso sulla base delle performance mostrate in fase di allenamento dalla rete;

vengono poi applicati ai primi due layer lineari delle funzioni di attivazione ReLu, mentre nell'ultimo di norma non si applicano funzioni di attivazione.

Come step finale del processo di forward, è presente un "flatten", ovvero una funzione che ha il compito di ridimensionare l'output in maniera tale da renderlo compatibile con il vettore delle std reali con cui dovrà poi confrontarsi.

La parte di codice successiva (vedi Codice 3) esegue l'inizializzazione della rete, ovvero la creazione del modello nel quale verranno poi passati i dati; quest'ultimo viene, poi, allocato nella GPU (o CPU, in base alla disponibilità) per effettuare le computazioni.

Dopodiché viene definita la loss function utilizzata, che nel nostro caso è il Mean Squared Error (MSE), l'optimizer scelto, ovvero l'Adaptive Moment Estimation (ADAM), e il valore di inizializzazione del learning rate, nel nostro caso 0.001.

ADAM è un algoritmo di ottimizzazione molto potente che può essere usato al posto della versione più classica, la Stochastic Gradient Descent (SGD). La peculiarità di questo optimizer risiede nella sua capacità di adattare i tassi di apprendimento per ciascun peso della rete, diversamente dalla SGD che mantiene un unico tasso di apprendimento per tutti gli aggiornamenti dei pesi. Questa adattabilità si traduce in prestazioni notevolmente migliori rispetto ad altri optimizer.

La porzione di codice che segue è quella che concerne la fase di training del modello (vedi Codice 4).

Il primo parametro da impostare è il numero di epoche di training, ovvero il numero di cicli nel quale la rete si allena sull'intero dataset di training, seguono i due cicli iterativi, il primo sul numero di epoche e il secondo sul database. Quest'ultimo effettua un numero di cicli differenti in base al valore impostato di batch size, poiché al suo variare il codice ci impiegherà più o meno iterazioni a completare una epoca di training.

All'interno del secondo ciclo, la prima linea di comando che viene eseguita richiama la classe della rete e le manda in input i dati, i quali vengono processati, e successivamente, restituiti come valori predetti; dopodiché questi ultimi vengono passati alla loss function, nella quale si esegue un confronto con i valori veri corrispondenti alle osservabili a cui fanno riferimento, e si riceve in output l'errore. Segue il comando per azzerare il gradiente (altrimenti ogni iterazione verrebbe sommato ai valori precedenti), il comando per effettuare la backward propagation e lo step dell'ottimizzatore; per finire, ogni 100 epoche viene stampato il valore in output dalla loss function.

La parte finale del codice consiste nella fase di testing della rete, fase molto importante in quanto viene valutata l'accuratezza della rete neurale senza effettuare più gli aggiornamenti dei pesi (vedi Codice 5).

Successivamente vengono creati dei vettori che serviranno a valutare l'accuratezza della rete; questi sono posti prima del ciclo for che itera sul numero di osservabili che compongono i dataset di testing.

All'interno del ciclo for è presente un altro comando molto importante che imposta temporaneamente il flag "requires_grad" delle variabili a "False", specializzandole per il training.

A questo punto vengono prese le osservabili e vengono passate alla rete neurale che restituirà in output una previsione di std, la quale verrà comparata con il valore vero e successivamente verrà calcolata l'accuratezza della previsione in maniera manuale. Si è scelto di valutare l'accuratezza della rete attraverso il conteggio del numero di previsioni che si discostassero al massimo di ±5 m dal valore reale valutate sul numero di osservabili totali.

6.4 Risultati

Dalle simulazioni effettuate con il modello di rete neurale descritto precedentemente, il risultato ottenuto in fase di training è soddisfacente, in quanto, si può notare come la loss tenda a diminuire all'aumentare del numero di epoche simulate.

La rete in generale ha mostrato una buona accuratezza sia in fase di training, sia nella successiva fase di testing nella quale si è riusciti a raggiungere un'accuratezza, che ha raggiunto valori del 95%.

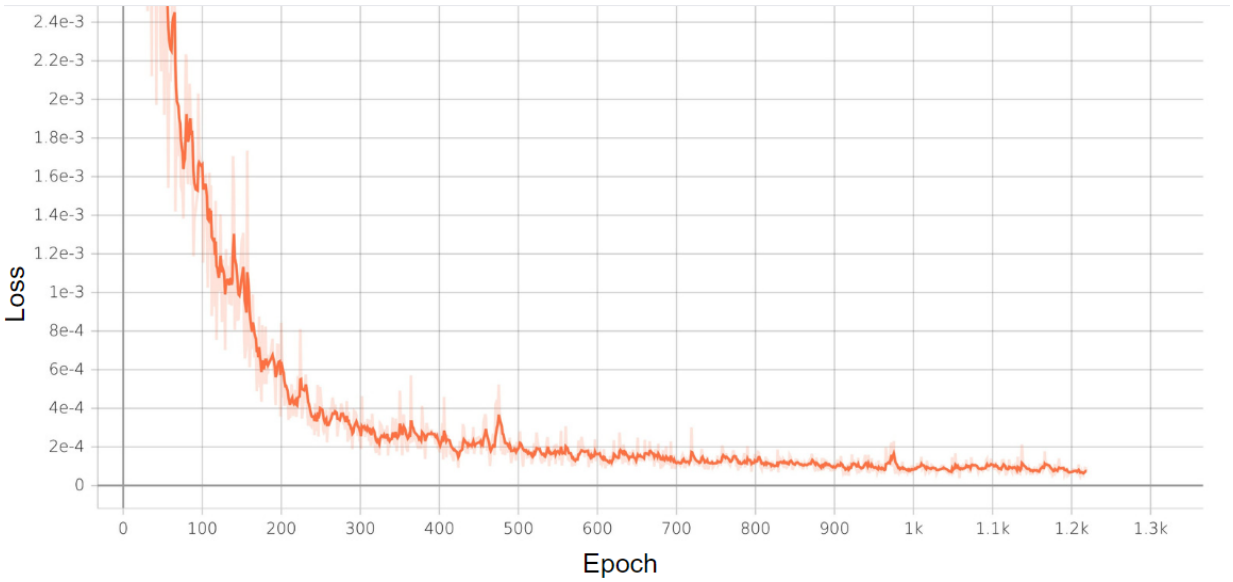

Figura 22: Rappresentazione della loss della rete neurale in fase di training

Per stabilire quali valori utilizzare negli iperparametri sono state effettuate alcune prove e sono stati confrontati i grafici ottenuti.

Partendo dall'analisi relativa al numero di hidden size, si è trovato che questo non andava a variare molto la velocità di convergenza del modello, in quanto, per i valori testati, le curve risultano pressoché sovrapposte (vedi Figura 23).

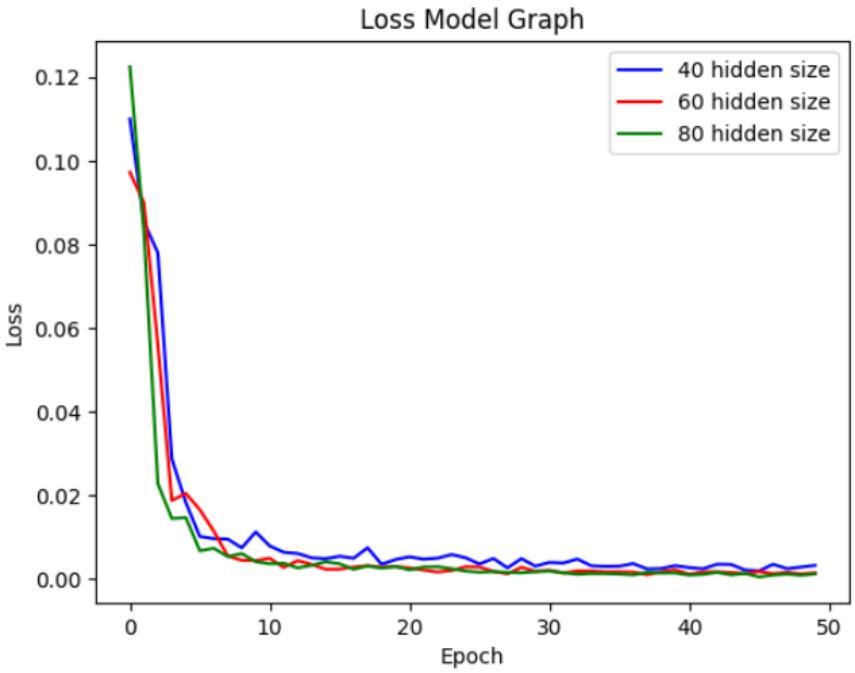

Figura 23: Plot della loss della rete neurale al variare degli hidden size

Un parametro che, invece, varia in modo più apprezzabile la velocità di convergenza del modello, è il numero di layers della rete LSTM.

In particolar modo sono stati testati 4 valori di layers, e si è visto come con 2 e 3 layers la rete aveva delle performance maggiori in termini di velocità di convergenza rispetto alla rete con 1 o 4 layers (vedi Figura 24 alla pagina successiva).

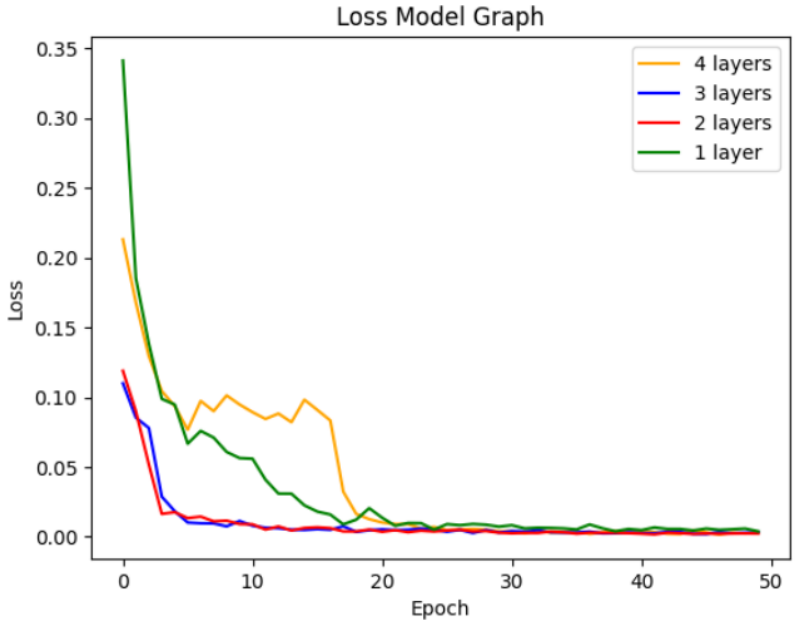

Figura 24: Plot della loss della rete neurale al variare dei layers

Come anticipato precedentemente, il modello che è meglio riuscito a predire le std delle osservabili di testing, è quello che ha raggiunto l'accuracy del 95%.

Per visualizzare di quanto si sono discostate le predizioni effettuate dalla rete rispetto ai valori reali delle relative osservabili, si è plottato l'errore rispetto alla n-esima osservabile.

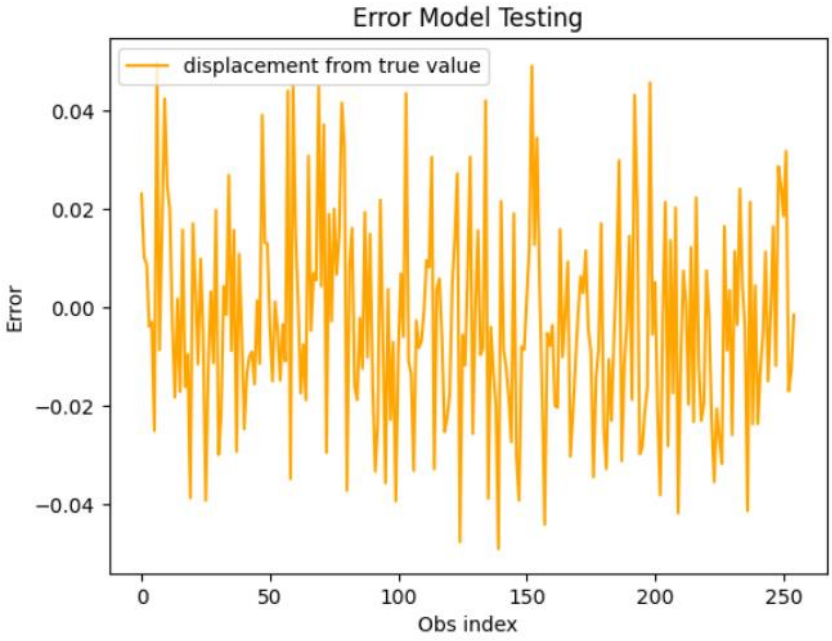

Figura 25: Plot del discostamento del valore predetto dal valore vero in fase di testing

L'asse x del plot (Figura 25) corrisponde al numero della singola osservabile, dato che nel dataset di testing le osservabili erano 267, ogni numero dell'asse x si riferisce all'indice della relativa

osservabile. L'asse y, invece, descrive l'errore della predizione, in particolare, avendo normalizzato le standard deviations tra 0 e 1, ed avendo allenato la rete neurale con valori di std compresi tra 0 e 100, i numeri nell'asse y se moltiplicati per 100 indicano l'esatto valore di scostamento dal valore reale. Per esempio, se si prende un'osservabile che ha come errore di stima sulla std un valore di 0.02, questo corrisponde ad un valore reale di 2 m.

La rete neurale è stata testata con diversi modelli dinamici, ovvero con osservabili aventi l'orbita di base che risente delle diverse accelerazioni dovute, per esempio, dalla radiazione solare o dall'effetto delle armoniche sferiche.

I modelli descritti sopra erano tutti modelli dinamici con cui la rete non era stata allenata, dato che il nostro dataset è diviso in una parte di training, composta solo da osservabili che includono solo il modello PM del corpo, e una di testing, con invece anche gli altri modelli dinamici. Questo è stato fatto per testare l'accuratezza delle predizioni della rete in caso dovesse trovarsi a maneggiare modelli dinamici mai visti prima.

7. Conclusioni

Il lavoro effettuato in questa tesi è suddivisibile in due parti, quella inerente ai test sul filtro di navigazione, che sono serviti a studiarne il comportamento a fronte di osservabili e modelli propagati aventi dinamiche differenti, e la seconda parte che ha visto la configurazione, l'allenamento e il testing della rete neurale che serve ad analizzare i dati di input del filtro ed estrapolare la quantità di rumore a cui sono affetti.

I test sul filtro sono stati articolati come segue, in primo luogo si è scelta la complessità del modello dinamico utilizzato per generare le osservabili (SRP, SPH oppure SRP + SPH), e il livello di rumore gaussiano bianco (definito con una certa standard deviation ed una media pari a zero). Per ognuna di queste tre tipologie di traiettorie si è effettuato un test con la matrice di covarianza del disturbo sullo stato ($\boldsymbol{Q}$) che teneva in considerazione l'incertezza del solo modello Point Mass, nel quale veniva inglobato anche l'effetto di terzo corpo dovuto dal Sole e da Giove, e un test che andava a tenere in considerazione le incertezze corrispondenti al modello dinamico delle osservabili in input.

L'approccio scelto descritto poc'anzi è stato dettato dall'esigenza di capire quale fosse il margine di miglioramento della stima effettuata dal filtro nel caso si fosse andati ad utilizzare un modello più accurato.

I risultati ottenuti hanno confermato quello che ci si aspettava; infatti, per le osservabili che tengono conto della Solar Radiation Pressure si è passati da un RMS^2 di 47,63 m^2 per il modello PM, ad un valore di RMS^2 di 33,79 m^2 per il modello SRP. Per le altre due simulazioni il trend è stato il medesimo, dato che per SPH l'errore è calato da 46,17 m^2 a 35,72 m^2 e infine per il modello completo ALL, il decremento è stato da un iniziale valore di RMS^2 = 44,21 m^2 fino ad arrivare a 36,29 m^2.

Per quanto concerne la rete neurale, invece, che ha costituito la seconda parte dell'attività di tesi, è stato deciso di implementare una Long-Short Term Memory (LSTM), poiché si è rivelata essere la più adatta alla tipologia di dati con i quali si doveva operare, ovvero dati composti da sequenze temporali.

È stato creato un dataset di osservabili a cui è stato associato del rumore bianco con un valore di std scelto nell'intervallo [0, 100] per il training e il testing della rete; a questo è seguito un periodo nel quale è avvenuta la costruzione della rete, esplorando diverse configurazioni al fine di trovare quella che desse i risultati più attendibili.

L'ultimo step è stato l'allenamento della LSTM, attraverso il quale si è riusciti ad ottenere un accuracy massima del 95%.

Ovviamente il modello sviluppato non è completo ed è sicuramente migliorabile. Per questioni di tempo, alcune delle idee che sono venute fuori durante il lavoro, non sono state realizzate, ma le riportiamo comunque di seguito come spunto per eventuali progetti futuri.

Uno dei primi miglioramenti attuabili potrebbe essere quello di ampliare il database in modo da poter dare la possibilità alla rete di allenarsi con diverse tipologie di rumore (e non solo con rumore gaussiano bianco), con modelli dinamici e una matrice di transizione più complessi, ecc.

Un'altra soluzione interessante potrebbe essere l'utilizzo delle Physics-Informed Neural Networks (PINN) che permettono all'interno dei propri layer di definire leggi fisiche. Questa soluzione può risultare molto interessante perché permetterebbe di integrare il Kalman Filter stesso all'interno della rete neurale.

8. Appendice

- Codice 1: Creazione dei dataset

```
n = len(trajectories)
percentuale_allenamento = 0.8
print(n)
train_data = trajectories[0:int(n*percentuale_allenamento)]
train_noise = noise_values[0:int(n*percentuale_allenamento)]

test_data = trajectories[int(n*percentuale_allenamento):]
test_noise = noise_values[int(n*percentuale_allenamento):]
print(len(train_data))
print(len(test_data))


from torch.utils.data import DataLoader, TensorDataset
dataset = TensorDataset(train_data, train_noise)

batch_size = 100
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

#osservabili di test:
dataset_obs = TensorDataset(test_data, test_noise)
batch_size = 100
dataloader_obs = DataLoader(dataset_obs, batch_size=batch_size, shuffle=True)
```

- Codice 2: Classe della rete neurale

```
class LSTM(nn.Module):
    def __init__(self, input_size=3, hidden_layer_size=40, output_size=1):
        super().__init__()
        self.hidden_layer_size = hidden_layer_size

        self.lstm = nn.LSTM(input_size, hidden_layer_size, dropout=0.5,
num_layers=3, batch_first=True)

        self.linear = nn.Linear(hidden_layer_size, 20)
        self.linear2 = nn.Linear(20, 10)
        self.linear3 = nn.Linear(10, output_size)

        self.linear.reset_parameters()

    def forward(self, input_seq):

        lstm_out, self.hidden_cell = self.lstm(input_seq)
        predictions = torch.relu(self.linear(lstm_out[:, -1, :]))
        predictions = torch.relu(self.linear2(predictions))
        predictions = self.linear3(predictions)
        predictions = torch.flatten(predictions, start_dim=0)

        return predictions

    def weights_init(self):

        nn.init.xavier_uniform(self.lstm.parameters)
```

- Codice 3: Inizializzazione rete neurale

```
model = LSTM()
model.to(device)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

print(model)
torch.cuda.empty_cache()
```

- Codice 4: Fase di training

```
epochs = 2000

for i in range(epochs):
    model.train()
    for seq, labels in dataloader:

        y_pred = model((seq.float()).to(device))

        single_loss = loss_function(y_pred, (labels.float()).to(device))

        optimizer.zero_grad()
        single_loss.backward()
        optimizer.step()

    if i%100 == 0:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {(i+1):3} loss: {single_loss.item():10.10f}')
```

- Codice 5: Fase di testing

```
model.eval()

vec_acc = 0
vec_test = test_data.size()[0]
for seq, labels in dataloader_obs:
  with torch.no_grad():
          std_pred = model((seq.float()).to(device))
          acc = std_pred - (labels.float().to(device)
          #print(std_pred)
          #print(labels)
          for j in range(len(acc)):
            if abs(acc[j]) <= 0.05:
              vec_acc += 1

print(f"Accuracy: {(vec_acc/vec_test)*100}%")
```

9. Bibliografia

1. “Fundamentals of Astrodynamics and Applications” fourth edition, David A. Vallado, Wayne D. McClain, 29 March 2013.

2. “Physical Side-Channel Attacks on Embedded Neural Networks: A Survey”. Maria Méndez Real, Rubén Salvador, 2021.

3. “Gravitational field modelling near irregularly shaped bodies using spherical harmonics: a case study for the asteroid (101955) Bennu”, Blazej Bucha, Fernando Sansò, 2021.

4. “Statistical Orbital Determination”, Byron D. Tapley, Bob E. Schutz, George H. Born, 10 June 2004.

5. “Heterogeneous mass distribution of the rubble-pile asteroid (101955) Bennu”, D. J. Scheeres, A. S. French.

6. “Satellite Orbits: Models, Methods, Applications”, Oliver Montenbruck, Eberhard Gill.

7. “Deep Learning from Scratch: Building with Python from First Principles”, Seth Weidman.

Ringraziamenti

Volevo iniziare i miei ringraziamenti rivolgendomi al Prof. Tortora e Prof. Zannoni che mi hanno concesso la possibilità di svolgere questa tesi presso il laboratorio di Radio Science and Planetary Exploration, poter svolgere la tesi triennale presso questo laboratorio è sempre stato un mio desiderio.

Ringrazio l'Ing. Zeqaj per avermi accompagnato durante tutta la mia esperienza, per avermi insegnato tanto, essere sempre stato presente e paziente nei miei confronti anche nei momenti di difficoltà.

Vorrei poi ringraziare tutte le persone che mi hanno accompagnato durante questo percorso, sia che esse siano state un passaggio sia che facciano tutt'ora parte del viaggio.

Un ringraziamento ai miei amici Jonathan, Francesco, Alessandro, Samuele, Jacopo e Michael che sono sempre al mio fianco, Mattia che mi è stato molto vicino in questo ultimo periodo, un ringraziamento speciale a Daniele che durante questa esperienza triennale ha condiviso con me sacrifici, sfoghi, sconfitte, vittorie, ma anche tante risate. Per me è stato ed è tutt'ora un fratello più che un amico.

Per ultimi, ma non per importanza, Elena e Maurizio, i miei genitori, che nonostante tutto sono sempre stati come le stelle per un navigante. A voi devo tutto.