Alma Mater Studiorum · Università di Bologna

corso di laurea in informatica per il management

Previsione del prezzo delle azioni di S&P con reti neurali LSTM e GRU

Presentata da:
Paolo Pesci

Relatore:
Elena Loli Piccolomini

Matricola n°:
806580

Sessione unica
Anno Accademico 2020 - 2021

Ringrazio i miei genitori
per l'opportunità che mi hanno dato.
Un grazie va anche ai miei parenti e ai miei amici,
che mi sono stati vicini in questo periodo.

Indice

Elenco delle figure

Introduzione

L'oggetto di questo lavoro di tesi si basa sullo studio delle reti neurali di tipo LSTM e GRU, l'analisi effettuata prende in considerazione lo Standard & Poor's 500 ovvero il più importante indice azionario americano, questo paniere contiene 500 titoli azionari di società quotate a New York, rappresentative dell'80% circa della capitalizzazione di mercato.[11]
La tesi è strutturata in tre macroargomenti:

1. Nel primo macroargomento tratto le reti neurali come strumento per l'analisi del mercato azionario. Affronto a livello teorico i concetti previsione delle serie temporali, stazionarietà, correlazione, classificazione e discriminazione, approssimazione di funzioni e faccio una breve presentazione delle principali tipologie di apprendimento automatico

2. Nel secondo macroargomento analizzo la struttura delle principali reti neurali, facendo una breve introduzione sui neuroni artificiali, andando in seguito a descrivere le reti *feed-forward* e le RNN. Successivamente porto alla luce il problema tipico da cui sono afflitte le reti neurali RNN e come tramite LSTM e GRU riusciamo a risolverlo. In ultima analisi spiego brevemente i concetti di overfitting e underfitting andando ad presentare varie metodologie possibili per evitarli

3. Infine, nel terzo ed ultimo macroargomento effettuo l'analisi e le previsioni del prezzo di S&P 500 utilizzando due tipologie di *layers*, LSTM e GRU.
   Ho creato tre diverse reti neurali, una composta solo da *layers* LSTM, una composta da soli *layers* GRU ed una mista formata dal primo *layer* LSTM ed i due successivi GRU. L'analisi è strutturata in:

   - Preparazione del *dataset*
   - Creazione del modello e addestramento
   - Creazione del *test set*
   - Plot delle predizioni e errori di predizione
   - Predizioni future
   - Risultati raggiunti
   - Approcci alternativi

1 Le reti neurali come strumento per l'analisi del mercato azionario

Prevedere l'andamento del prezzo delle azioni è un problema che ha da sempre attirato il mondo accademico e industriale. Lo studio e l'utilizzo delle reti neurali a questo scopo risale alla fine degli anni 80, inizio anni 90 quando numerosi studiosi ed esperti si cimentarono in questa analisi (White 1988, Kimoto 1990, Kim 2000). [15] Negli ultimi anni il *deep learning* ha raggiunto un grande successo in questo ambito segnando una svolta in tal senso. Predire l'andamento del prezzo delle azioni significa analizzare dati di serie temporali di grandi dimensioni e il *deep learning* ha sviluppato una forte capacità in merito. [15] Considerando che il mercato azionario è un sistema di movimento non lineare, molto complesso e la sua legge di fluttuazione è influenzata da molteplici fattori, prevedere l'indice del prezzo delle azioni risulta un compito molto impegnativo, a questo scopo ci vengono incontro gli algoritmi delle reti neurali che nel tempo sono stati perfezionati ed adattati allo scopo di analizzare serie temporali di grandi dimensioni riuscendo ad ottenere risultati nelle previsioni più che soddisfacenti. [11] Nonostante la complessità di questi sistemi è tuttavia possibile provare a schematizzare gli ambiti applicativi in campo finanziario in 3 diverse categorie: [13]

- Previsione serie temporali
- Classificazione
- Approssimazione di funzioni

1.1 Previsione delle serie temporali

La previsione delle serie temporali, in particolare la previsione degli indici finanziari, che ha come obiettivo quello di fare previsioni sul futuro andamento dei prezzi di mercato, rappresenta un concetto chiave di fondamentale importanza. Una serie storica o temporale(*timeseries*) è un'insieme di dati che traccia un campione nel tempo. In ambito finanziario, una serie temporale tiene traccia del movimento di una determinato indice ad intervalli regolari ed in un periodo di tempo specifico. Vengono quindi raccolti i dati sotto forma di *dataset* in modo tale che l'investitore o analista abbia tutte le informazioni per poter effettuare successivamente analisi e previsioni. Possiamo osservare due tipologie di serie temporali:

- Univariata: una variabile varia nel tempo, perciò, ad ogni passo, si avrà un valore unidimensionale
- Multivariata: più variabili variano nel tempo, quindi ad ogni passo, avremo un valore n-dimensionale in base al numero di variabili

Per poter analizzare una serie temporale ci sono quattro componenti principali da tenere sotto osservazione:

- Stagionalità
- Tendenza
- Ciclicità
- Irregolarità

La componente stagionale osserva la variabilità dei dati nel tempo, spiegando gli alti ed i bassi periodici, vi possono essere più periodi stagionali sovrapposti, la componente di tendenza osserva la direzione(*trend*) che i dati stanno seguendo nel tempo, la componente ciclica non rappresenta un periodo fisso come per le componenti stagionali ma possiamo accostarla ai periodi in cui vi sono fenomeni mondiali che innescano una forte crescita o un forte crollo dei mercati azionari(es. covid, possibili guerre...), la componente irregolare invece osserva e anticipa la variazione casuale nei dati che non possono essere previsti in anticipo. [7]
Attualmente, il *deep learning* è diventato sempre più efficiente nel perfezionare molti aspetti di serie temporali anche complessi, diversi *frameworks* sono stati proposti nella letteratura allo scopo di prevedere il prezzo degli *asset*. [8] Tuttavia, le previsioni a lungo termine di serie temporali finanziarie rimangono ancora di difficile comprensione, a causa della complessa evoluzione temporale e dell'interazione tra i punti temporali di queste ultime. Le serie temporali legate al mondo finanziario, in particolare quelle relative ai prezzi di azioni o *asset* principalmente speculativi, derivano dal fatto che sono

influenzate da “fattori caotici”, che portano a rappresentazioni con bruschi cambiamenti o inversioni inaspettate, considerate come *outliers* che minano l'affidabilità e la capacità di generalizzazione dei modelli di apprendimento.

1.2 Stazionarietà

La stazionarietà è la condizione che assume un processo stocastico nel momento in cui alcune sue proprietà non cambiano rispetto ad una variazione temporale. Possiamo definire una serie storica come stazionaria nel momento in cui possiede una media costante nel tempo, una varianza costante nel tempo, il rapporto tra valori distinti k periodi è influenzato solo da k, non dal punto della serie in cui viene calcolata. [13] Una serie stazionaria non ha *trend*, stagionalità e non presenta dei cicli. Per controllare la stazionarietà di una serie storica esistono diverse metodologie, il primo metodo è quello della visualizzazione del grafico della serie temporale, tramite il grafico possiamo farci un'idea di come si sviluppa l'andamento della serie storica, osservando dunque *trend* e stagionalità. Un' altro metodo è quello di calcolare la media mobile e la deviazione standard mobile, se i due valori descrivono delle linee piatte allora significa che la serie è stazionaria. Infine il test Dickey-Fuller Aumentato (ADF) è uno dei test statistici più utilizzati. La serie temporale è considerata stazionaria se il valore di p (p-value) è basso (minore di una soglia di 0,05) e i valori critici a intervalli di confidenza dell'1%, 5%, 10% sono il più vicino possibile alle statistiche ADF. [13]

1.2.1 Convertire una serie temporale in stazionaria

Poiché la stazionarietà è un presupposto fondamentale in molte procedure statistiche utilizzate nell'analisi delle serie storiche, i dati non stazionari sono spesso trasformati per diventare stazionari. Sopratutto per quanto riguarda l'ambito finanziario, una serie temporale relativa ad un'indice o ad un'azienda, difficilmente potrà essere già stazionaria. Le due tecniche principali usate per rendere una serie temporale stazionaria sono: la trasformazione logaritmica che rende lineare un *trend* esponenziale e aiuta a rendere costante la varianza e la differenziazione, che va a sottrarre il valore corrente dal precedente e può essere utilizzata anch'essa per trasformare una serie temporale in stazionaria.[13] Queste due tecniche possono essere anche utilizzate insieme.

1.3 Correlazione

La correlazione è una misura statistica che esprime la relazione lineare tra due variabili (che cambiano insieme ad una velocità costante) ed è molto usata per descrivere semplici relazioni. La correlazione viene descritta mediante un valore che non è dotato di un'unità di misura specifica, chiamato coefficiente di correlazione, compreso tra -1 e +1 e denotato da 'r': - Più 'r' si avvicina a zero, più la correlazione è debole (le variabili non sono correlate); - Un valore 'r' positivo implica una correlazione positiva, in cui i valori delle due variabili tendono ad aumentare in parallelo (le variabili sono correlate direttamente); [13]

1.4 Classificazione e discriminazione

Applicazioni tipiche in questo campo riguardano la valutazione del rischio di credito come, ad esempio, la suddivisione in classi di *rating* o le decisioni di affidamento. Nei casi di classificazione la rete ha il compito di assegnare gli input ad un certo numero di categorie predefinite cui corrispondono altrettanti output mentre nei modelli destinati alla discriminazione la rete deve anche creare le classi stesse nelle quali suddividere i dati di *input*. [13]

1.5 Approssimazione di funzioni

In questo caso, le reti vengono applicate in tutte le funzioni avanzate di *pricing* e di *risk management* nelle quali manca una forma funzionale precisa per la valutazione degli strumenti. Si pensi, ad esempio, alle opzioni di tipo americano, alle opzioni esotiche e ai portafogli di opzioni. [13]

1.6 Tipologie di apprendimento

L'apprendimento automatico è una parte dell'informatica dove l'efficienza di un sistema migliora eseguendo ripetutamente delle attività(task), invece che utilizzare i dati esplicitamente programmati dai programmatori. In letteratura esistono tre tecniche di *Machine Learning*: apprendimento supervisionato, apprendimento non supervisionato e l'apprendimento per rinforzo. [4]

1.6.1 Apprendimento supervisionato

L'apprendimento supervisionato è una metodologia che crea un modello per prevedere un risultato basato su dati etichettati, ovvero una raccolta di variabili(caratteristiche) ed un *output* specifico che stiamo cercando di prevedere. Per la parte progettuale di questo elaborato di tesi ho utilizzato questa tipologia di apprendimento. Inoltre l'apprendimento supervisionato presuppone che i dati futuri si comporteranno in modo simile ai dati storici. Gli algoritmi "imparano" da un determinato set di dati, il che significa che si adatta a un modello basato su comportamenti ed etichette passati. A volte, quando questi modelli vedono dati aggiornati, non funzionano altrettanto bene. Quando ciò accade, diciamo che il modello è "*overfit*", nel senso che è eccessivamente ottimizzato sui dati storici e quindi non è ben generalizzato. Vedremo i concetti di *overfitting* e *underfitting* di un modello nel prossimo capitolo.

1.6.2 Apprendimento per rinforzo

L'apprendimento per rinforzo può essere considerato come un caso particolare di apprendimento supervisionato, è una tecnica che fornisce *feedback* sull'allenamento, utilizzando un meccanismo di ricompensa. Il processo di apprendimento avviene come una macchina, o agente, che interagisce con un ambiente e prova una varietà di metodi per raggiungere un risultato. L'Agente è ricompensato o punito quando raggiunge uno stato desiderabile o indesiderabile. L'agente scopre quali stati portano a buoni risultati e quali sono disastrosi e devono essere evitati. Il successo viene misurato con un punteggio (indicato come Q, l'apprendimento per rinforzo è talvolta chiamato *Q-learning*) in modo tale che l'agente possa imparare in modo iterativo per ottenere un punteggio più alto.

1.6.3 Apprendimento non supervisionato

L'apprendimento non supervisionato è una tecnica che determina schemi e associazioni nei dati non etichettati. Questa tecnica viene spesso utilizzata per problemi di *clustering* e regole di associazione. A differenza dell'apprendimento supervisionato questo tipo di apprendimento utilizza solamente un set di dati in ingresso senza alcuna indicazione dell'*output* desiderato.

2 Dal neurone alle reti neurali LSTM e GRU

Negli ultimi anni l'intelligenza artificiale ha subito un forte sviluppo. L'utilizzo di reti neurali nella previsione dei prezzi di mercato è diventato un tema molto importante su cui la ricerca ha fatto grandi passi avanti. [2]

2.1 Neurone artificiale

I neuroni sono le unità di base delle reti neurali artificiali e svolgono delle attività semplici come:

- ricevere *inputs* in ingresso
- moltiplicare gli ingressi per il relativo peso
- sommare i risultati dei prodotti
- aggiungere al risultato un *bias*
- applicare al totale finale una funzione di attivazione
- utilizzare il risultato di questa funzione come uscita

Sono disegnati per simulare le funzioni di un neurone biologico. I segnali che ricevono, chiamati *inputs* vengono moltiplicati per i pesi di connessione, vengono sommati e infine passati attraverso un funzione che li trasforma e produce l'output. La funzione di attivazione è la somma pesata degli input del neurone mentre la funzione di trasformazione più comune è la funzione sigmoidea.
Vediamo ora in modo schematico in Figura 2.1 la struttura di un singolo neurone ed il flusso che seguono i dati prima che venga generato l'*output*.

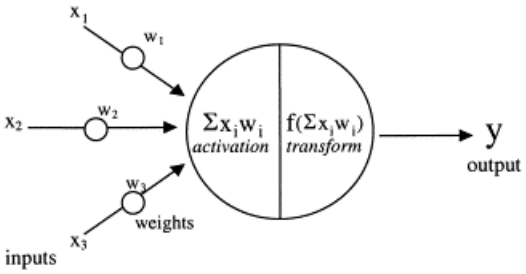

Figura 2.1: Modello di neurone artificiale

I neuroni artificiali formano delle connessioni reciproche andando a determinare il funzionamento di una rete neurale artificiale, ovvero un modello computazionale di ispirazione biologica formato da centinaia di queste unità connesse tramite coefficienti(pesi) che ne costituiscono la struttura neurale. Possiamo vederne la struttura in Figura 2.2

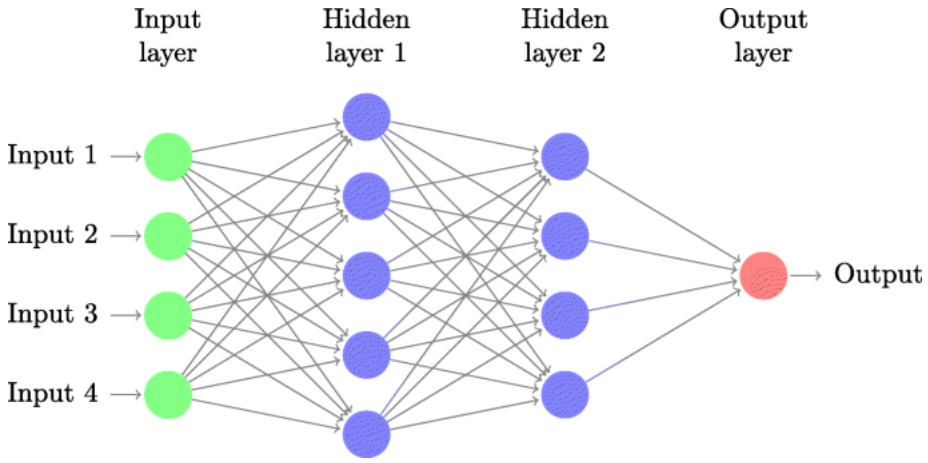

Figura 2.2: Struttura di una ANN con due strati nascosti ed un singolo output

Una ANN(Artificial neural network) è composta da tre strati di (*layers*), nodi principali: lo strato di input composto dai dati di ingresso, uno o più strati nascosti, i quali si occupano dell'elaborazione dei dati e lo strato di output, che conterrà il risultato finale. Un neurone di un singolo strato è collegato a tutti i neuroni di uno strato successivo e possiede un peso ed una soglia correlato. Un nodo si attiva trasferendo i dati al livello successivo della rete, nel momento i cui l'uscita del nodo si trova al di sopra della soglia, specificata dalla funzione di attivazione.
Ogni neurone che compone una rete neurale è composto da dati di *input*, pesi, un *bias* e un *output* come possiamo vedere dalle formule seguenti [1] [6]

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias$$

$$\text{output} = f(x) = \begin{cases} 1 \text{ if } \sum w_1 x_1 + b \geq 0 \\ 0 \text{ if } \sum w_i x_1 + b < 0 \end{cases}$$

2.2 Reti feed-forward FFNN

Una rete neurale *feed-forward* è uno schema di collegamento fra neuroni(nodi della rete) in cui la trasmissione degli impulsi è orientata in avanti. Possiamo quindi definire dei neuroni di input da cui partono gli impulsi, e dei neuroni di output, che costituiscono il loro punto di arrivo. Tra i neuroni di *input* ed *output* possono essere collocati dei neuroni intermedi che ricevono impulsi dai neuroni situati sui livelli precedenti, li elaborano e li trasmettono a quelli dello strato successivo. Vediamo un' esempio di questa tipologia di rete in Figura 2.3

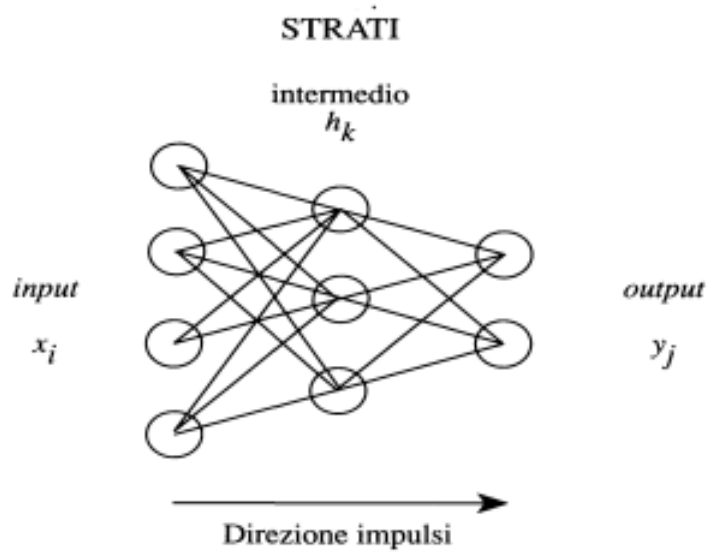

Figura 2.3: Esempio di rete *feed-forward*

La rete è costituita da 4 neuroni *input*, tre neuroni intermedi posti nello strato nascosto(*hidden layer*) e da due neuroni di *output*. [12] Il valore di questi neuroni è determinato dai valori assunti dai 4 neuroni *input*, dall'insieme dei pesi sintattici e dalla legge di attivazione. Possiamo osservare il valore calcolato come:

$$y_j(\boldsymbol{x},\boldsymbol{w}) = f\left(\sum_{k=1}^{3} w_{kj} f\left(\sum_{i=1}^{4} w_{ik} x_i - \theta_k\right) - \theta_j\right)$$

Le reti *feed-forward* non hanno memoria di *input* avvenuti in tempi precedenti, per cui l'*output* è determinato solamente dall'attuale *input*.

2.3 Recurrent Neural Network RNN

Le reti neurali ricorrenti (RNN) sono reti formate da neuroni artificiali ovvero unità interconnesse e collegate tra di loro da archi (sinapsi) aventi un peso. Vengono in contro alla limitazione che si genera con le reti neurali di tipo feed-forward (FFNN), le quali hanno neuroni che pur essendo sullo stesso livello non possono comunicare tra di loro ma possono inviare segnali/informazioni allo strato successivo. Possono essere definite come reti cicliche, ovvero i valori di uscita di uno strato di un livello superiore vengono utilizzati come ingresso per uno strato inferiore. Dalla figura 2.4 vediamo una rete formata da un singolo neurone ricorrente, come si può vedere l'uscita del neurone torna all'ingresso del neurone stesso creando una retroazione, di seguito è possibile vedere il ciclo di retroazione dispiegato lungo la sequenza temporale, questo porta ad evidenziare le correlazioni tra i valori futuri e quelli passati.

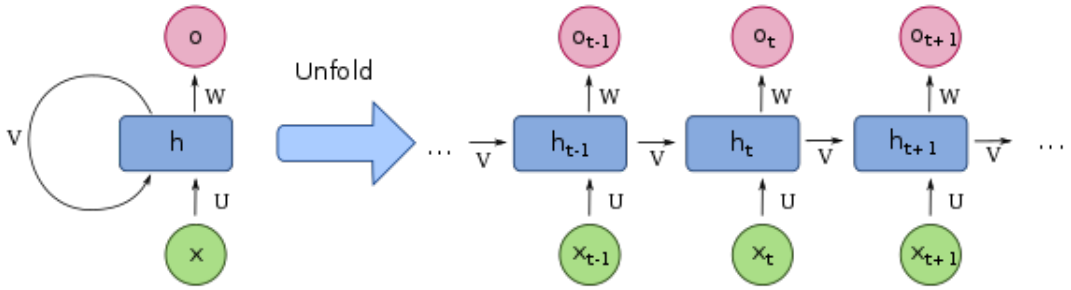

Figura 2.4: ciclo di retroazione dispiegato RNN

La ciclicità di queste reti consente ai neuroni che la compongono di avere memoria degli ingressi passati, quindi di gestire situazioni in cui è necessario mantenere un contesto per elaborare in modo corretto le informazioni, vengono dunque impiegate per svolgere compiti di analisi predittiva su sequenze di dati, quali il riconoscimento della grafia o il riconoscimento vocale.

2.4 Il problema della scomparsa del gradiente

Le RNN soffrono di memoria a breve termine, quando una sequenza di informazioni è troppo lunga fanno molta difficoltà a trasportare le informazioni attraverso i passaggi temporali, quindi, elaborando una grande mole di dati per fare delle previsioni potrebbero tralasciare informazioni importanti.
Il metodo utilizzato durante la fase di addestramento di una rete neurale è quello della *backpropagation*, ovvero la propagazione all'indietro, la quale aggiorna i vari parametri dei neuroni (pesi e *bias*) in modo proporzionale alla derivata parziale della *loss-function* rispetto al parametro stesso. Durante questo processo le RNN soffrono del problema della scomparsa del gradiente (*vanishing or explodient gradient*), i gradienti sono valori che vengono utilizzati per aggiornare i pesi di una rete neurale, il problema si verifica quando il valore del gradiente diminuisce o aumenta mentre si propaga indietro nel tempo.
Quando il valore del gradiente diventa troppo piccolo o troppo grande non contribuisce più all'apprendimento, nelle RNN i livelli che ottengono un piccolo aggiornamento del gradiente smettono quindi di apprendere, dunque una RNN essendo dotata di memoria a breve termine potrebbe dimenticare ciò che ha visto in sequenze più lunghe.
Possiamo dunque delineare, come si vede in Figura 2.5, due equazioni che descrivono i problemi relativi al gradiente:

Figura 2.5: Scomparsa ed esplosione del gradiente

Nel primo caso, il termine converge a zero in modo esponenziale, questo comporta la difficoltà di apprendimento di alcune dipendenze di lungo periodo. Nel secondo caso, il termine converge all'infinito in modo esponenziale ed il valore del gradiente diventa un *NaN(Not a Number)* a causa dell'instabilità del processo.

Per limitare l'effetto della scomparsa o l'esplosione del gradiente, come vedremo nella prossima sezione, ci vengono incontro due tipologie di neuroni, gli LSTM(*Long-Short Term Memory*) ed i GRU(*Gatered Recurrent Unit*). [3]

2.5 Reti neurali LSTM e GRU

Le reti LSTM e GRU sono state create proprio per sopperire al problema della memoria a breve termine, limitare l'effetto del *vanishing/exploding gradient* e consentono di poter lavorare su sequenze di ingresso più lunghe. Vengono utilizzate per problemi tipici come:

- generazione di testo
- riconoscimento vocale
- sintesi vocale
- generazione di didascalie per video
- previsione di indici azionari

Vediamo ora di seguito in Figura 2.6 la struttura di un neurone LSTM ed un neurone RNN.

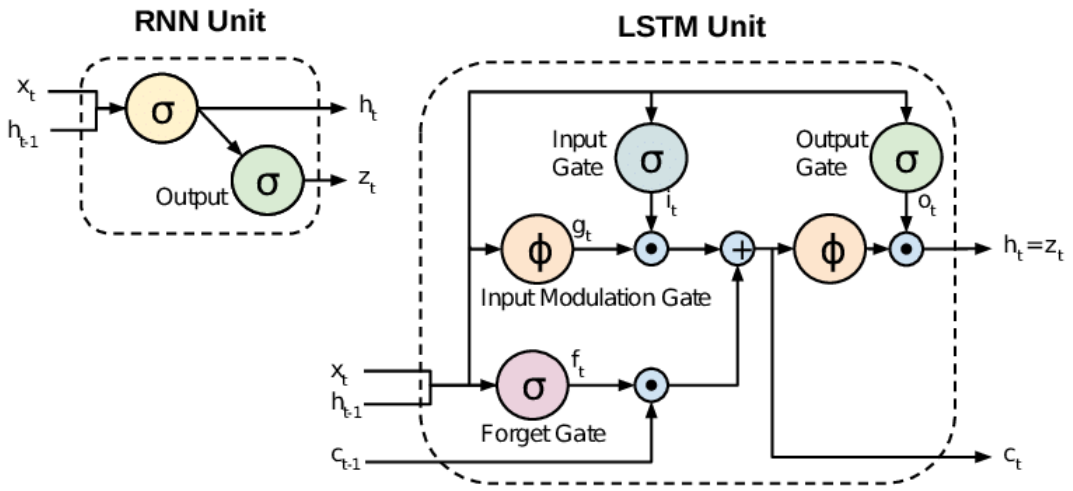

Figura 2.6: Struttura interna di un neurone RNN, LSTM

Come possiamo osservare dalla figura il neurone LSTM ha una struttura più complessa di un neurone ricorrente. Il primo gestisce un'uscita supplementare h(t) per lo stato attuale, un' ingresso h(t-1) per lo stato precedente e possiede due funzioni di attivazione differenti per lo stato h e l'uscita z. Il neurone LSTM invece è composto internamente da diverse porte(*gate*), che gli consentono di decidere durante la fase di training cosa memorizzare o dimenticare, come combinare l'ingresso con lo stato interno e come restituire l'uscita. Il neurone LSTM possiede al suo interno tre porte principali (forget gate, input gate, output gate) di cui vedremo il funzionamento nella successiva sottosezione.

Un neurone GRU come possiamo vedere in Figura 2.7 rappresenta una semplificazione dell'LSTM, esso gestisce solo due gate l'*update gate* e l'*input gate*, vedremo il loro funzionamento in dettaglio nella successiva sottosezione.

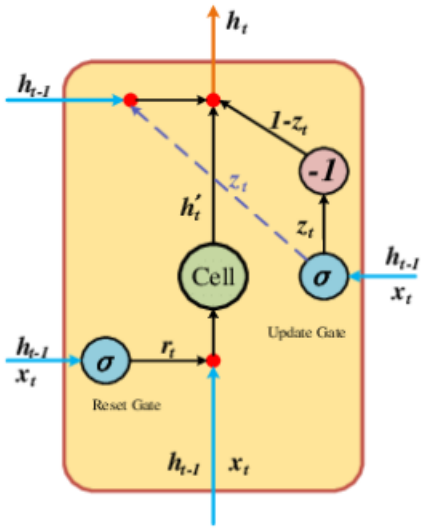

Figura 2.7: Struttura interna di un neurone GRU

2.5.1 Tipologie di *gate* e funzionamento

I *gates* regolano il flusso delle informazioni all'interno dei neuroni, apprendono quali dati in una sequenza eliminare o conservare in modo tale da poter trasmettere informazioni rilevanti lungo la catena di sequenza e fare successivamente delle previsioni. Le operazioni presenti all'interno del neurone LSTM in Figura 2.6 consentono all'unità di conservare o dimenticare le informazioni.
Alla base di una rete LSTM vi è lo stato di una cella ed il funzionamento dei suoi *gates*, all'interno della cella le informazioni vengono trasportate lungo tutta la catena della sequenza dando vita ad una vera e propria "memoria di rete", le informazioni fluiscono attraverso i passaggi temporali precedenti e quelli successivi andando a ridurre l'effetto della memoria a breve termine. I *gates* sono differenti reti neurali che decidono quali informazioni sono ammesse allo stato della cella, imparano quali informazioni sono rilevanti e le mantengono altrimenti le dimenticano durante l'addestramento del modello. Dalla figura 2.6 possiamo vedere che all'interno della cella LSTM sono presenti due tipi di funzioni di attivazione, sigmoidea e tanh, queste funzioni fanno in modo che i dati che vi passano all'interno vengano schiacciati in un range di valori definito dal tipo di funzione, per la sigmoidea i dati vengono compressi tra 0 e 1 mentre la funzione tanh li comprime tra -1 e 1.

Questo meccanismo risulta di aiuto per aggiornare o dimenticare i dati poiché ogni numero che viene moltiplicato per 0 è 0 causando la perdita o scomparsa di valori, mentre ogni numero che viene moltiplicato per 1 rimane lo stesso valore e viene mantenuto. Grazie a queste funzioni la rete neurale può quindi comprendere quali dati sono rilevanti o meno e quindi decidere se conservarli o dimenticarli.
Ogni unità LSTM è denominata "memory unit" ed è composta da tre tipologie di gates:

- Forget gate
- Input gate
- Output gate

Il *Forget gate* come possiamo vedere in Figura 2.8 decide quali informazioni devono essere conservate o scartate, i dati passano attraverso la funzione sigmoidea che li comprime tra 0 e 1, i valori vicini allo zero vengono dimenticati mentre quelli vicini a 1 vengono conservati.

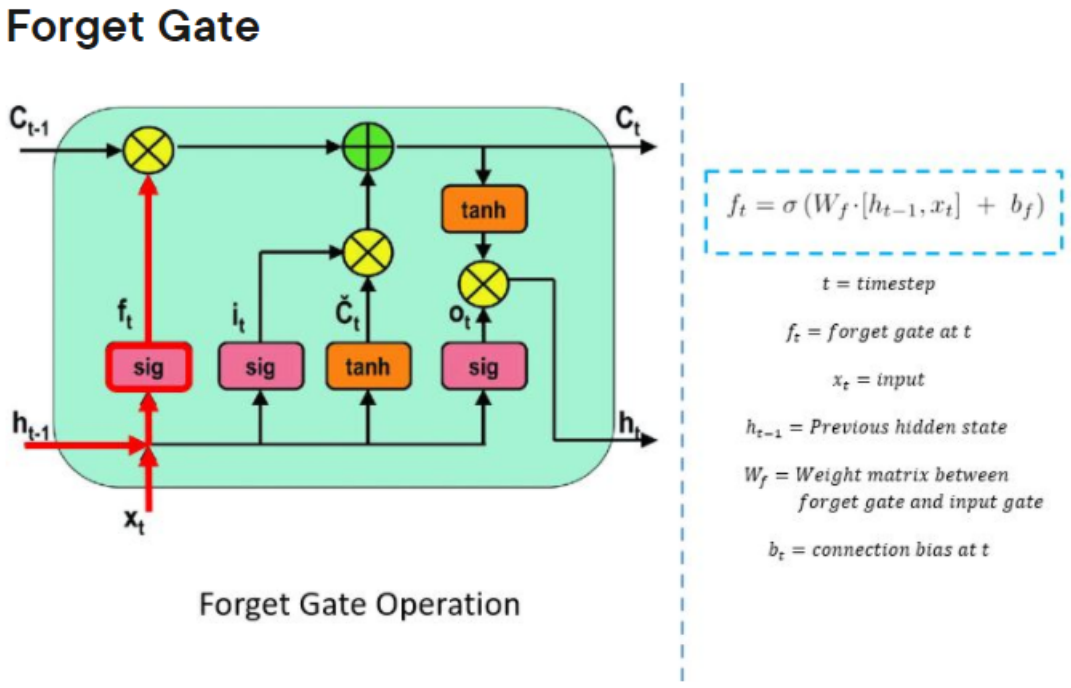

Figura 2.8: Operazioni svolte dal Forget gate

L'*Input gate* si occupa di aggiornare lo stato della cella, viene passato lo stato nascosto precedente e l'input corrente attraverso la funzione sigmoidea che decide quali valori verranno aggiornati o meno comprimendoli tra 0 e 1. I valori vicino allo 0 sono meno importanti da aggiornare rispetto ai valori vicino all'1. Inoltre, i dati dello stato nascosto e dell'*input* corrente vengono passati anche alla funzione tanh, la quale ci viene incontro nella regolazione della rete comprimendoli tra -1 e 1.
Infine, viene moltiplicato l'*output* tanh con l'*output* della funzione sigmoidea, quest'ultima deciderà quali informazioni è importante conservare rispetto all'*output* di tanh.

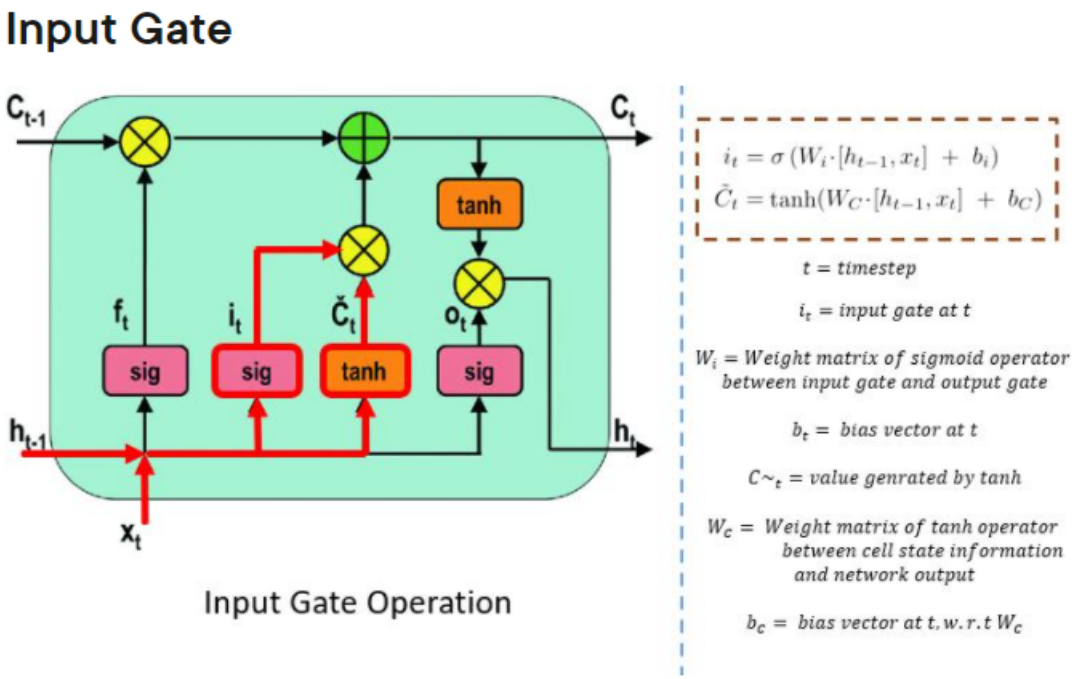

Figura 2.9: Operazioni svolte dall'Input gate

A questo punto abbiamo abbastanza informazioni per calcolare il *cell state*. E' necessario moltiplicare in modo puntuale il *cell state* per l'*output* del *forget gate*. Successivamente prendiamo l'*output* dall'*input gate* e lo sommiamo puntualmente aggiornando lo stato della cella ad un nuovo valore che la rete neurale trova rilevante, questo ci conduce ad un nuovo *cell state*.

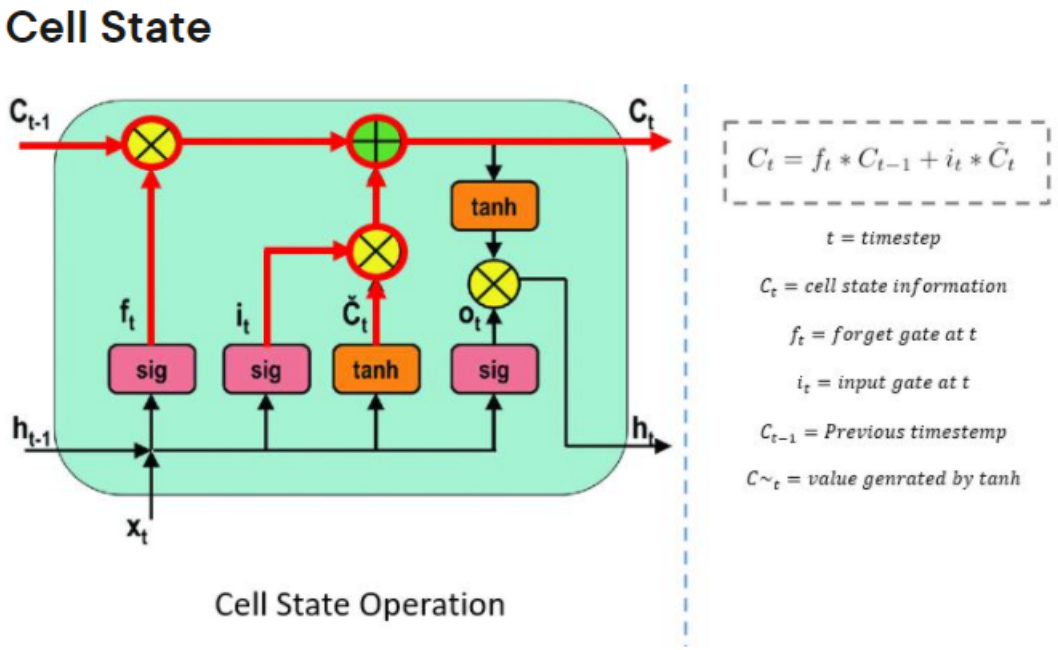

Figura 2.10: Calcolo dello stato della cella

L'*Output gate* decide quale deve essere il prossimo stato nascosto, lo stato nascosto tiene traccia degli *input* precedenti ed è utilizzato per le previsioni. Viene passato lo stato nascosto precedente e l'*input* corrente in una funzione sigmoidea. Successivamente passiamo il *cell state* appena modificato alla funzione tanh e moltiplichiamo il suo *output* con l'*output* della funzione sigmoidea, in modo che vengano trasportate all'*hidden state*. L'*output* è lo stato nascosto. Una volta terminato il processo il nuovo *cell state* ed il nuovo *hidden state* vengono portati al passaggio temporale successivo.
[3] [14]

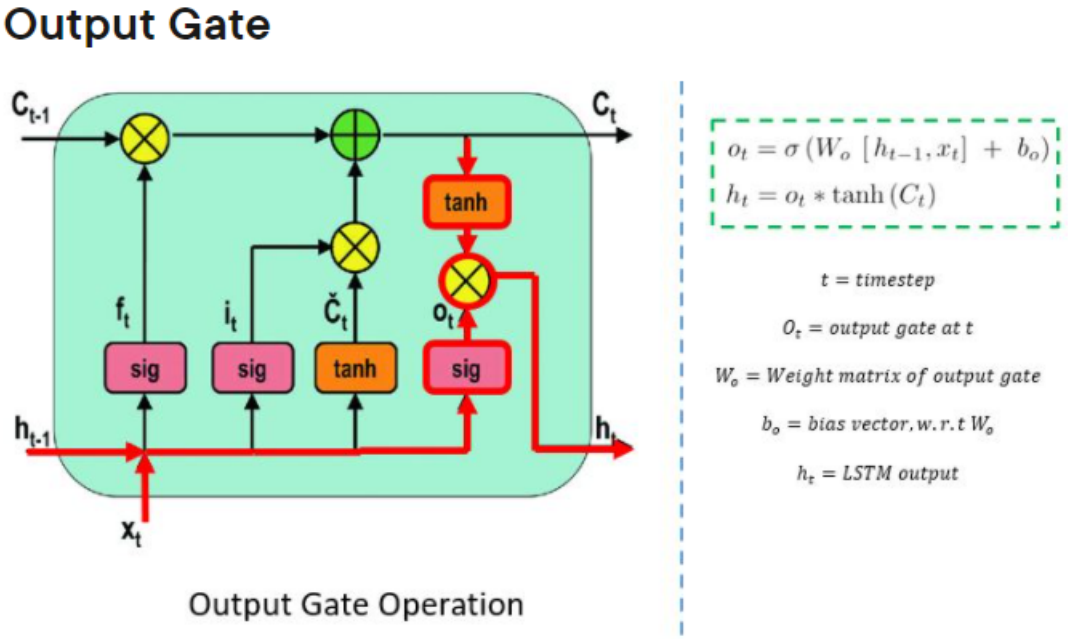

Figura 2.11: Operazioni svolte dall'Output gate

In sintesi il forget gate decide quali dati scartare o conservare dai passaggi precedenti, l'input gate decide quali informazioni è rilevante aggiungere dal passaggio corrente mentre l'output gate determina il nuovo hidden state.

2.6 Overfitting e underfitting

L'*overfitting* è un problema chiave nelle attività di *Machine Learning* supervisionate. Questo fenomeno viene rilevato quando un algoritmo di apprendimento si adatta così bene al set di dati di allenamento da memorizzare il rumore e le peculiarità dei dati di allenamento. In base al risultato degli algoritmi di apprendimento, le prestazioni diminuiscono quando vengono testate in un set di dati sconosciuto. La quantità di dati utilizzati per il processo di apprendimento è fondamentale in questo contesto. I set di dati di piccole dimensioni sono più inclini all'*overfitting* rispetto ai set di dati di grandi dimensioni e, nonostante la complessità di alcuni problemi di apprendimento, i set di dati di grandi dimensioni possono anch'essi essere influenzati dall'*overfitting* (Santos, EM, Sabourin, R. e Maupin,P. 2009). [9] L'eccessivo adattamento dei dati di addestramento porta al deterioramento delle proprietà di generalizzazione del modello, ciò si traduce in prestazioni inaffidabili se applicato a nuove misurazioni. Quindi lo scopo dei metodi che cercano di evitare l'*overfitting* è in qualche modo contraddittorio rispetto all'obiettivo degli algoritmi di ottimizzazione, che mirano a trovare la migliore soluzione possibile nello spazio dei parametri in base alla funzione obiettivo predefinita e ai dati disponibili. Inoltre, diversi algoritmi di ottimizzazione possono funzionare meglio per architetture ANN più semplici o più grandi. Ciò suggerisce l'importanza di un corretto accoppiamento di diversi algoritmi di ottimizzazione; architetture e metodi ANN per evitare un adattamento eccessivo dei dati del mondo reale. Alcuni metodi per evitare il problema dell'*overfitting* e dell'*underfitting* nell'apprendimento automatico supervisionato sono: [10]

- Regolazione della grandezza del *dataset*
- Convalida incrociata(*cross validation*)
- Regolarizzazione
- Selezione delle caratteristiche

2.6.1 Regolazione della grandezza del *dataset*

Generalmente, *dataset* di grandi dimensioni soffrono meno di *overfitting*, in quanto con una maggiore quantità di dati si riduce la sovrautilizzazione da parte del sistema che per progredire è costretto a generalizzare. Utilizzando un numero maggiore di dati si potrà quindi migliorare notevolmente la precisione del modello, riducendo l'eccesso di adattamento, questo aspetto è di rilevante importanza durante un'attività di analisi di dati. Vediamo ora in Figura 2.12 come si comporta il grafico in situazione di *overfitting* ed *underfitting*.

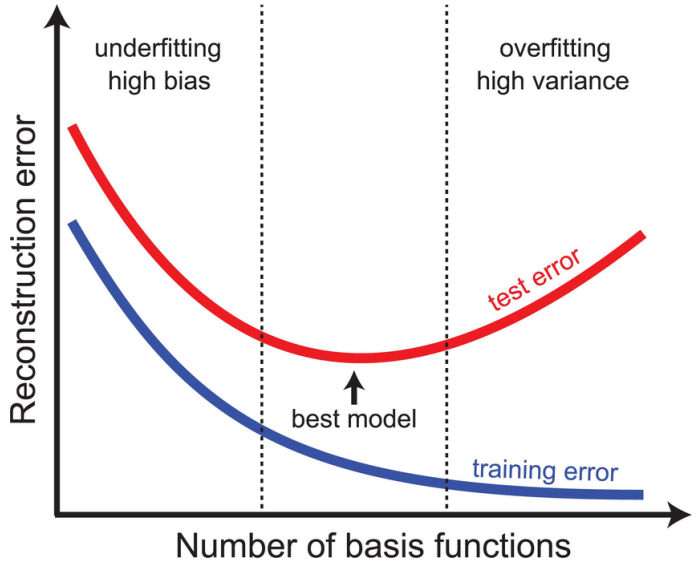

Figura 2.12: Situazioni di *overfitting* e *underfitting*

Dalla terza regione del grafico possiamo osservare che l'errore sul *train* continua a scendere, perché il modello impara sempre meglio com'è fatto quest'ultimo, ma aumenta sul *test*, perché non sa performare in modo corretto su un *dataset* mai visto, si delinea quindi una situazione di *overfitting* con alta varianza.

2.6.2 Convalida incrociata

La convalida incrociata è un metodo utilizzato per stimare le prestazioni dei modelli di *machine learning*, aiuta in modo efficiente a prevenire l'*overfitting*. Il metodo tradizionale di convalida incrociata sussiste nel dividere il dataset in 3 sottoinsiemi. Il *training set* formato dal 60% dei dati, il *validation set* formato dal 20% ed il restante 20% dei dati compongono il *test set*. Per far si che questa impostazione sia possibile è necessario avere un *dataset* di dimensioni cospicue ed il *training set* sia composto da abbastanza dati,

altrimenti sarà necessario utilizzare la convalida incrociata. La convalida incrociata non utilizza l'intero *training set* per l'addestramento del modello, viene rimossa una parte di dati prima dell'esecuzione del *training* in modo da poter essere utilizzata una volta che il modello è pronto. Lo scopo principale della convalida incrociata è dunque quello di trovare il modello più funzionale e prestante. Si possono delineare più tipi di convalida incrociata:

- *K-Fold standard*
- *Stratified K-Fold Cross-Validation*
- *Holdout Method*
- *Leave-One-Out-Cross-Validation (LOOCV)*
- *Leave-P-Out Cross-Validation (LpOCV)*

2.6.3 Regolarizzazione

Anche la regolarizzazione è utilizzata come metodo per prevenire l'*overfitting*, quando un modello è troppo complesso c'è la tendenza al sovradattamento. Con la regolarizzazione vengono aggiunti nuovi parametri alla *loss function*, che si sta cercando di minimizzare. E' una tecnica utilizzata per semplificare i modelli, per utilizzarla in modo adeguato deve essere di tipo moderato, non troppo piccola in quanto non avrebbe nessun impatto, ne troppo grande in quanto si tradurrà in *underfitting*, allontanando il modello dalla realtà.

2.6.4 Selezione delle caratteristiche

Con il processo di selezione delle caratteristiche(*features*) si selezionano un sottoinsieme di caratteristiche significative che successivamente vengono utilizzate per costruire il modello. Quindi, invece di utilizzare tutte le funzionalità, è meglio eliminare quelle ridondanti o non rilevanti ed utilizzare solo quelle importanti. In questo modo renderemo il processo di addestramento più veloce, prevedendo l'eccesso di adattamento, in quanto non vi è la necessità da parte del modello di apprendere funzionalità inutili.

3 Analisi del prezzo di S&P 500: un' esempio pratico

3.1 Preparazione del *dataset*

Per effettuare l'analisi del *dataset* ho utilizzato come linguaggio di programmazione *Python* in quanto flessibile, dalla sintassi concisa e fornito di librerie pronte all'uso per effettuare l'analisi dei dati. Di seguito elencherò le principali librerie utilizzate:

- P*Pandas* per la manipolazione e l'analisi dei dati
- *Matplotlib* per la creazione dei grafici
- *NumPy*, il quale aggiunge il supporto a matrici di grandi dimensioni, *array* multidimensionali oltre a proporre una vasta gamma di funzioni matematiche di alto livello per poter operare in modo efficiente su queste strutture dati
- *Scikit-learn e Keras* per l'apprendimento automatico e per poter operare con le reti neurali

Per recuperare le informazioni relative al prezzo di S&P 500 ho utilizzato il sito della Federal Bank of St.Louis (FRED), la quale possiede uno dei più grandi database di dati economici. Viene reputata come una fonte eccellente ed affidabile da utilizzare per l'analisi finanziaria. Tramite il pacchetto *Pandas Datareader* come vediamo in Figura 3.1 sono riuscito in poche linee di codice a recuperare i dati di mio interesse, su i quali successivamente ho svolto il training del modello.

```
# Specifico il range di date su cui verrà creato il dataset
start = "1993-02-01"
end = "2019-12-31"

data  = web.DataReader([f'{stock_index}'], 'fred', start, end)
```

Figura 3.1: Recupero dei dati relativi al prezzo di S&P 500

Successivamente come vediamo in Figura 3.2 ho proceduto ad eliminare i valori NaN dal dataset in quanto non utili e di possibile disturbo in relazione all'analisi.

```
#Vado a eliminare tutti i NaN
data.dropna(inplace = True)
data.info()
print(data.head())
```

Figura 3.2: Eliminazione valori NaN e display del dataset

Stampando le informazioni relative al dataset preso in analisi, possiamo vedere in Figura 3.3 che è composto da due colonne, una rappresentante i giorni e l'altra il prezzo relativo raggiunto dall'indice in quel determinato giorno. Tramite la funzione *head()* ci vengono mostrati i primi 5 prezzi dell'indice.

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1975 entries, 2012-02-27 to 2019-12-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sp500   1975 non-null   float64
dtypes: float64(1)
memory usage: 30.9 KB
             sp500
DATE
2012-02-27  1367.59
2012-02-28  1372.18
2012-02-29  1365.68
2012-03-01  1374.09
2012-03-02  1369.63
```

Figura 3.3: Struttura del *dataset* preso in analisi

Il *dataset* è composto da 1975 elementi ed i valori relativi al prezzo sono di tipo *float64*.

Ho effettuato le analisi modificando due variabili prediction_days e predictionLength, andrò quindi ad utilizzare 10, 20, 30 giorni precedenti(prediction_days) per predire 3, 7, 28 giorni successivi(predictionLength).

```
scaler = MinMaxScaler(feature_range=(0, 1))
train = scaler.fit_transform(data[stock_index].values.reshape(-1, 1))

train_output = train.copy()
train = train[:-predictionLength, :]
train_output = train_output[prediction_days:, :]

i=0
xTrain, yTrain = [], []

while i+prediction_days < len(train):
  xTrain.append(train[i: i+prediction_days, :])
  yTrain.append(train_output[i: i+predictionLength, :])
  i=i+predictionLength

xTrain = np.array(xTrain)
yTrain = np.array(yTrain)

# xTrain è un set formato da predictionDays valori ciascuno
# yTrain è un set formato da predictionLength valori ciascuno
print(xTrain.shape) # 648 righe, 30 giorni input
print(yTrain.shape) # 648 righe, 3 giorni output
```

Figura 3.4: Creazione della matrice xTrain di *input* e della matrice yTrain di *output*

Successivamente sono andato a ridimensionare i dati in modo che risultino schiacciati tra 0 e 1, in questo modo possono passare attraverso le funzioni di attivazione senza saturarle. Vado quindi a riempire tramite la funzione *append()* xTrain ed yTrain creando quindi due matrici contenenti rispettivamente 648 righe, predictionLength colonne per l'input e 648 righe e prediction_days colonne per l'*output*.
Terminata la procedura di preparazione dei dati, ho proceduto con la creazione ed il *train* del modello.

3.2 Creazione del modello e addestramento

Per la creazione e l'addestramento del mio modello ho usato tre tipologie di strutture di rete neurali, la prima è una struttura composta da *layers* LSTM, la seconda da *layers* GRU e la terza è un modello ibrido con il primo strato LSTM ed i successivi strati GRU. Tutti e tre i modelli sono di tipo sequenziale, ovvero sono modelli che gestiscono una semplice pila di strati, in cui per ogni livello vi è un tensore di input ed un tensore di output. Vedremo nelle seguenti sottosezioni il codice relativo ad ogni tipologia di struttura.

3.2.1 modello LSTM

Il modello in Figura 3.5 è formato da tre strati LSTM, composti da 50 neuroni ciascuno e separati l'un l'altro da uno strato di dropout, questa tecnica di regolarizzazione consente di diminuire il possibile overfitting del modello, agisce spegnendo in modo casuale dei neuroni dai corrispettivi layers ad ogni epoca di addestramento, lasciando gli altri neuroni apprendere e adattarsi, in questo modo il modello ne trae beneficio, risultando:

- generalizzato in modo più efficiente
- meno sovradattato
- formato da neuroni indipendenti, quindi meno sensibili ai pesi individuali degli altri neuroni
- nella maggior parte dei casi fautore di risultati migliori

Infine, ho inserito lo strato Dense che conterrà l'output finale del modello, una volta che i dati avranno attraversato tutti i *layers*. Lo strato Dense è composto da tante unità, quante specificate nei predictionLength nodi.

```
[ ]  model1 = Sequential()

     model1.add(LSTM(units=50, return_sequences=True, input_shape=(xTrain.shape[1], 1)))
     model1.add(Dropout(0.4))
     model1.add(LSTM(units=50, return_sequences=True))
     model1.add(Dropout(0.4))
     model1.add(LSTM(units=50))
     model1.add(Dropout(0.4))
     model1.add(Dense(units=predictionLength))

     # Più sarà basso l'errore quadratico medio più la predizione sarà accurata
     model1.compile(optimizer='adam', loss='mean_squared_error')
```

Figura 3.5: Creazione del modello composto da *layers* LSTM

3.2.2 modello GRU

Il modello in Figura 3.6 è composto da tre strati di neuroni GRU, separati ciascuno da uno strato di Dropout ed uno strato Dense finale.

```
model2 = Sequential()

model2.add(GRU(units=50, return_sequences=True, input_shape=(xTrain.shape[1], 1)))
model2.add(Dropout(0.4))
model2.add(GRU(units=50, return_sequences=True))
model2.add(Dropout(0.4))
model2.add(GRU(units=50))
model2.add(Dropout(0.4))
model2.add(Dense(units=predictionLength))

# Più sarà basso l'errore quadratico medio più la predizione sarà accurata
model2.compile(optimizer='adam', loss='mean_squared_error')
```

Figura 3.6: Creazione del modello composto da *layers* GRU

3.2.3 modello ibrido LSTM, GRU

Infine ho creato un modello ibrido come possiamo vedere in Figura 3.7 composto da uno strato LSTM e due strati GRU, come per le altre reti neurali anche questa contiene per ogni *layer* uno strato di Dropout e uno strato Dense per l'*output*.

```
model3 = Sequential()

model3.add(LSTM(units=50, return_sequences=True, input_shape=(xTrain.shape[1], 1)))
model3.add(Dropout(0.4))
model3.add(GRU(units=50, return_sequences=True))
model3.add(Dropout(0.4))
model3.add(GRU(units=50))
model3.add(Dropout(0.4))
model3.add(Dense(units=predictionLength))

# Più sarà basso l'errore quadratico medio più la predizione sarà accurata
model3.compile(optimizer='adam', loss='mean_squared_error')
```

Figura 3.7: Creazione del modello ibrido composto da *layers* LSTM, GRU

Ho utilizzato il mean_squared_error come loss function, più sarà basso questo valore e più la predizione dei valori risulterà accurata.

Dopo aver creato i tre modelli di reti neurali, sono andato ad addestrarli tramite il metodo 10-fold crossvalidation che possiamo vedere in Figura 3.8. Questa tecnica permette di generalizzare il modello riducendo il possibile *overfitting*, ho suddiviso il *dataset* in dieci parti uguali, successivamente ho definito un ciclo iterativo il quale prende come prima parte il *validation set* e le restanti parti come *train set*. Per ogni iterazione viene quindi definito come *validation set* la successiva porzione di dati, questo fa credere al modello di allenarsi sempre con dati differenti inducendo una riduzione del rischio di *overfitting*.

```
#Addestramento del modello con 10-fold crossvalidation

fold = 10
index = np.arange(start=0, stop=len(xTrain), step=len(xTrain)/fold, dtype=int)

for k in range(len(index)-1):
  i=index[k]
  j=index[k+1]
  xv=xTrain[i: j]
  yv=yTrain[i: j]

  if(k==0):
    xt=xTrain[j:]
    yt=yTrain[j:]
  else:
    xt=np.concatenate((xTrain[:i], xTrain[j:]), axis=0)
    yt=np.concatenate((yTrain[:i], yTrain[j:]), axis=0)

  model1.fit(xt, yt, validation_data=(xv, yv), epochs=2, batch_size=32)
  #model1.fit(xt, yt, validation_data=(xv, yv), epochs=10, batch_size=32)
  #model1.fit(xt, yt, validation_data=(xv, yv), epochs=20, batch_size=32)
```

Figura 3.8: Addestramento del modello con 10-fold crossvalidation

Per l'addestramento effettivo del modello ho utilizzato il metodo fit() che prende in ingresso i dati di input e i dati di output del train set e del *validation set*, inoltre ho specificato il numero di epoche, ovvero le iterazioni che concedo al modello per addestrarsi sul train set ed il batch_size, che definisce il numero di campioni che verranno propagati attraverso la rete di volta in volta. Ho infine effettuato tre sperimentazioni andando a modificare dinamicamente il parametri epochs(2, 10, 20) e batch_size(8, 16, 32). Da notare che le epoche vengono moltiplicate per il numero di fold(10), quindi il numero di epoche effettivo sarà il risultato di questa moltiplicazione. Commenterò i risultati ottenuti dalle sperimentazioni nella sezione relativa ai risultati raggiunti.

3.3 Creazione del test set

Completato il *training* del modello ho iniziato la creazione del *test set*, come possiamo vedere in Figura 3.9 ho estratto i dati relativi ai prezzi di chiusura di S&P 500 dal 01/01/2020 fino alla data attuale.

```
test_start = "2020-01-01"
test_end = str(date.today())

test_data  = web.DataReader([f'{stock_index}'], 'fred', test_start, test_end)
test_data.dropna(inplace = True)

test = scaler.transform(test_data[stock_index].values.reshape(-1, 1))
test = np.concatenate((train[-prediction_days:], test[: -predictionLength]), axis=0)
test

i=0
xTest=[]

while i+prediction_days < len(test):
  xTest.append(test[i: i+prediction_days])
  i=i+predictionLength

xTest = np.array(xTest)
print(xTest.shape)

predictions = model1.predict(xTest)
predictions = predictions.reshape(-1, 1)

predictions = scaler.inverse_transform(predictions)
true = test_data[stock_index].values.reshape(-1, 1)

predictions = predictions[-len(true):]
true = true[-len(predictions):]
```

Figura 3.9: Creazione del test set

Successivamente ho utilizzato questa finestra temporale per rappresentare graficamente le predizioni dei valori, prendendo gli ultimi giorni precedenti (prediction_days) e andando a predire il prezzo di chiusura dell'indice (tramite la funzione predict() di Keras) nei giorni successivi (predictionLength). Ho opportunamente scalato i valori per renderli meglio rappresentabili graficamente.

3.4 Plot delle predizioni e errori di predizione

In Figura 3.10 possiamo vedere il codice che ho utilizzato per la rappresentazione dei grafici, verranno dunque mostrati i valori reali dell'indice ed i valori predetti dal modello.

```
plt.figure(dpi=200)
plt.plot(true, label = "real values")
plt.plot(predictions, label = "predicted values")
plt.grid()
plt.title("Valori reali vs. valori predetti")
plt.xlabel("Giorno")
plt.ylabel("Prezzo")
plt.legend(loc='upper left')
plt.show()
```

Figura 3.10: Creazione dei grafico rappresentante valori reali e valori predetti

Ho utilizzato la libreria MatplotLib per la rappresentazione dei grafici, in quanto possiede funzioni in-built semplici che consentono di creare rappresentazioni grafiche in modo chiaro ed agevole. Per calcolare l'errore di predizione commesso dal modello ho sottratto in valore assoluto i valori reali dai valori predetti e successivamente ho diviso per i valori reali, come possiamo vedere in Figura 3.11, ho anche calcolato la mediana e l'IQR(range interquartile) per avere una stima di dove si trovi l'errore medio di predizione.

```
error = abs(true-predictions)/true
mediana = np.median(error)
IQR = iqr(error, rng=(5, 95))/2

print("Prediction error =", mediana, " ±", IQR)
plt.figure(dpi=200)
plt.plot(error, label="Errore")
plt.plot([0, len(error) - 1], [mediana, mediana], "r-", label = "mediana")
plt.plot([0, len(error) - 1], [mediana - IQR, mediana - IQR], "y--", label = "range IQR")
plt.plot([0, len(error) - 1], [mediana + IQR, mediana + IQR], "y--")
plt.grid()
plt.title("Errore di predizione")
plt.xlabel("Giorno")
plt.ylabel("Errore relativo")
plt.legend(loc="upper right")
plt.show()
```

Figura 3.11: Calcolo errore di predizione e creazione grafico

Al fine di trovare il miglior modello che possa approssimare i dati reali, ho effettuato vari test andando a variare i parametri di epochs(2, 10, 20) e batch_size(8, 16, 32), controllando la distanza tra l'errore rappresentato dalla loss e l'errore rappresentato dalla validation_loss(minimizzando questa distanza si avrà un modello più accurato), riporterò nella prossime tre sottosezioni, per ogni modello il grafico che risulta approssimare in modo più efficiente i valori reali, rappresentando anche il relativo errore di predizione.

3.4.1 Grafici e errori rete LSTM

Per quanto riguarda il modello LSTM dopo i vari test ho riscontrato che la combinazione che meglio approssima il modello e riduce l'errore di predizione è composta da 10 epoche e 16 come dimensione di batch_size, possiamo vederla in Figura 3.12.

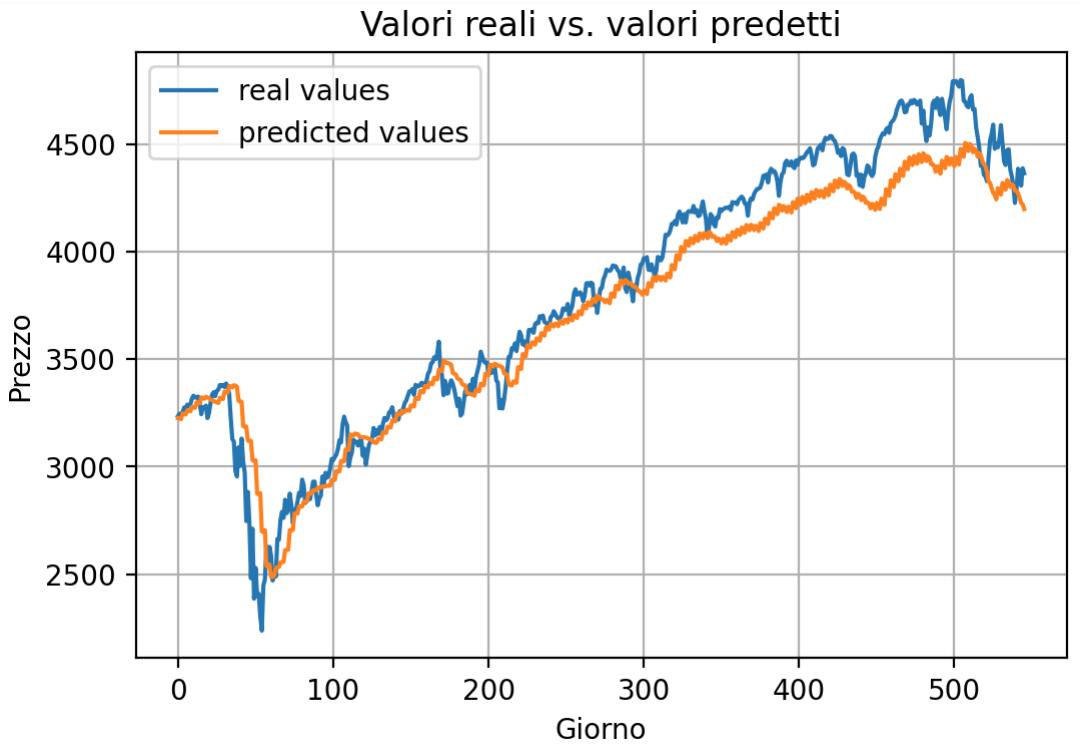

Figura 3.12: Grafico modello LSTM utilizzando epochs=10 e batch_size=16

Possiamo notare visivamente che i valori predetti non discostano di molto dai valori reali. Vediamo anche che i valori predetti sono leggermente spostati a destra rispetto ai valori reali, in quanto, nel caso di una brusca variazione il modello si accorge del cambio di tendenza e si aggiusta, ma solo a variazione avvenuta.

L'errore di predizione è pari a 0.0285 mentre il range IQR si attesta su $\pm$ 0.0343 come descritto dalla Figura 3.13.

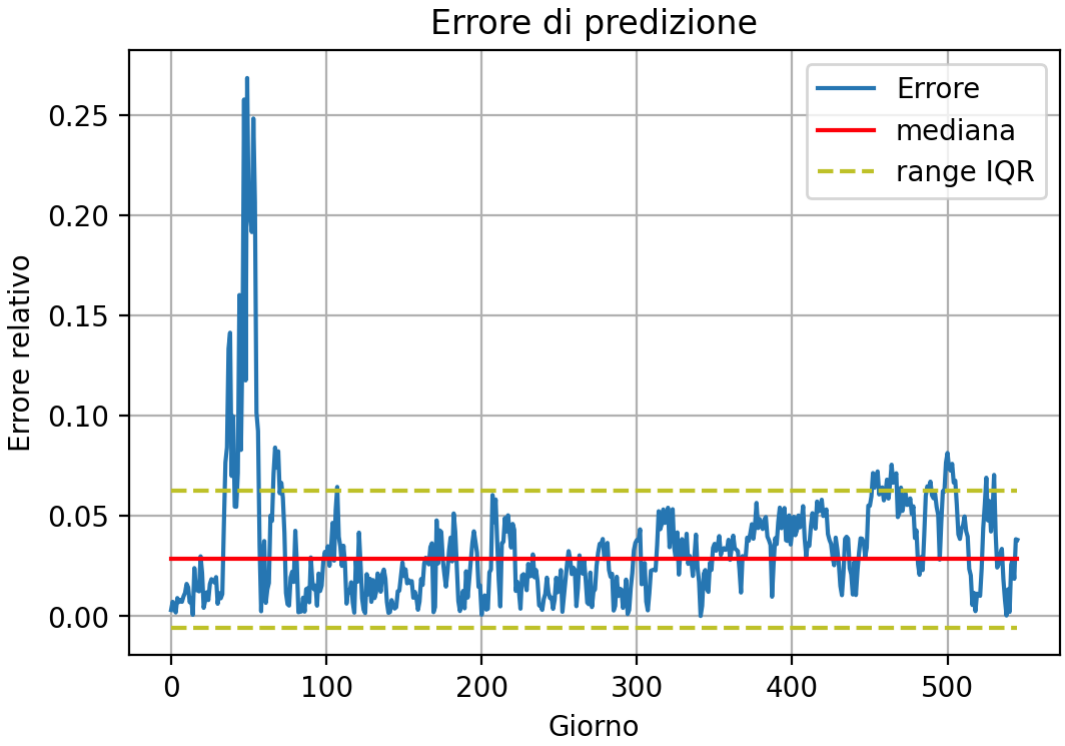

Figura 3.13: Errore modello LSTM utilizzando epochs=10 e batch_size=16

3.4.2 Grafici e errori rete GRU

Per la rete composta da *layers* GRU il modello con il minor errore di predizione è configurato con 2 epoche e 8 come dimensione di batch_size

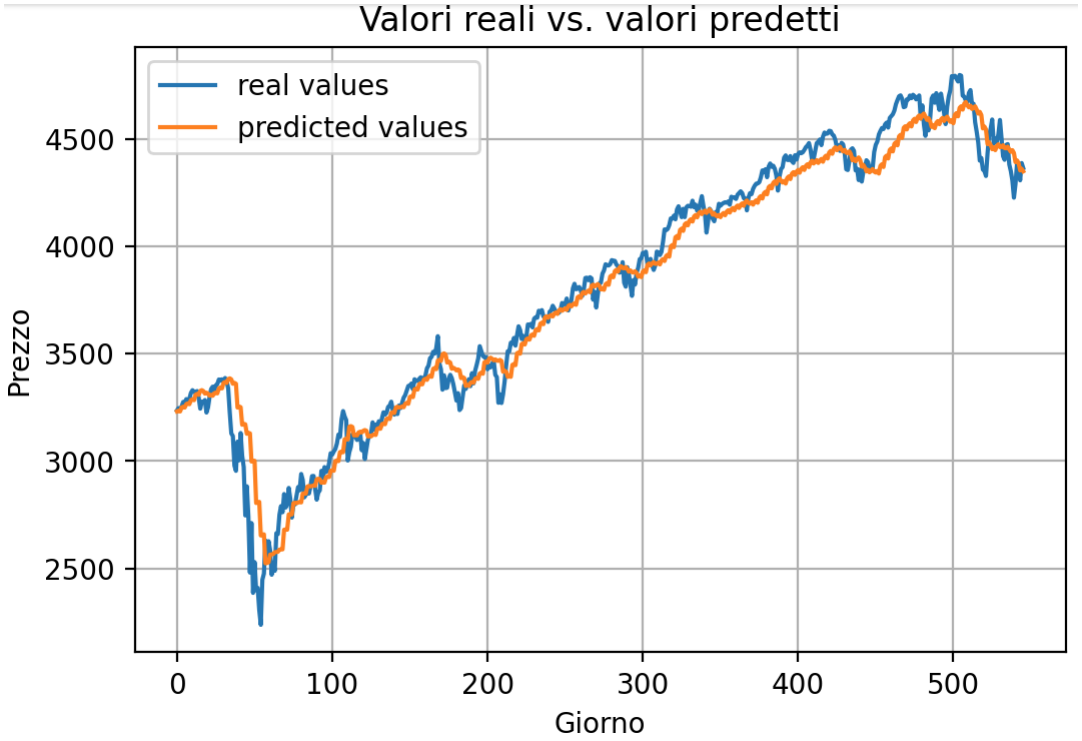

Figura 3.14: Grafico modello GRU utilizzando epochs=2 e batch_size=8

Dal grafico vediamo che bastano meno epoche per l'addestramento e una dimensione di batch_size inferiore per avere una precisione migliore rispetto a LSTM, questo è dato dal fatto che i neuroni GRU sono formati da una struttura più semplice rispetto ai neuroni LSTM e per addestrarsi impiegano un numero minore di iterazioni, epoche.

L'errore di predizione commesso dal modello è di 0.0130, mentre il range IQR è grande $\pm$ 0.0300.

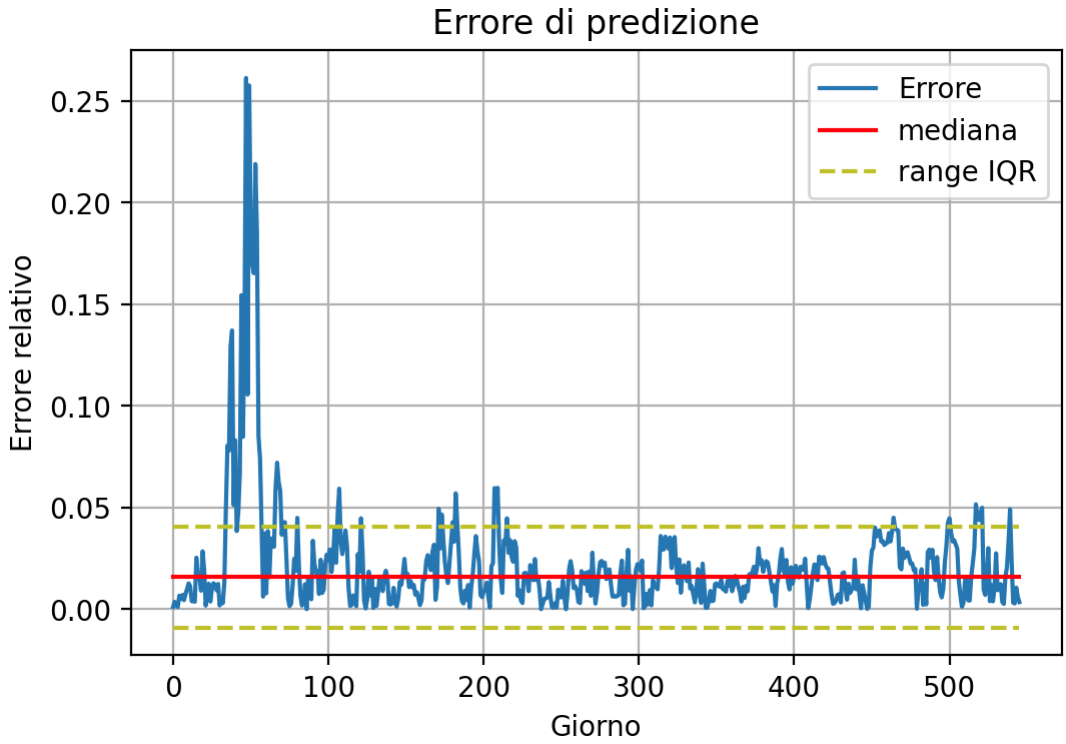

Figura 3.15: Errore modello GRU utilizzando epochs=2 e batch_size=8

3.4.3 Grafici e errori rete ibrida LSTM, GRU

Il modello ibrido, composto dal primo *layer* LSTM e gli altri due *layers* GRU ha dato come migliore approssimazione la combinazione di 2 epoche e 8 come dimensione di *batch_size*. Vediamo il risultato in Figura 3.16 Da notare che con la stessa configurazione

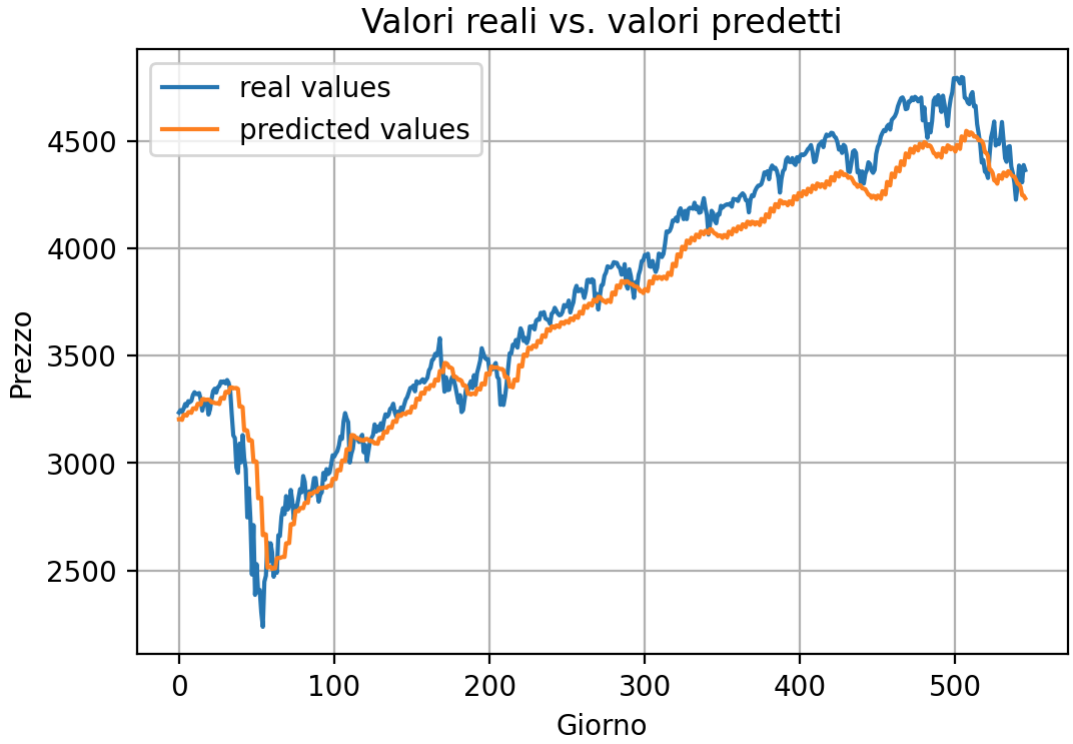

Figura 3.16: Grafico modello ibrido LSTM, GRU utilizzando epochs=2 e batch_size=8

si ha un errore più elevato rispetto al modello con solo *layers* GRU, questo perché il layer iniziale LSTM necessita di più epoche per addestrarsi a sufficienza.

Come si può vedere dalla Figura 3.17 l'errore di predizione in questo caso è di 0.0279 mentre il range IQR è grande ±0.0324

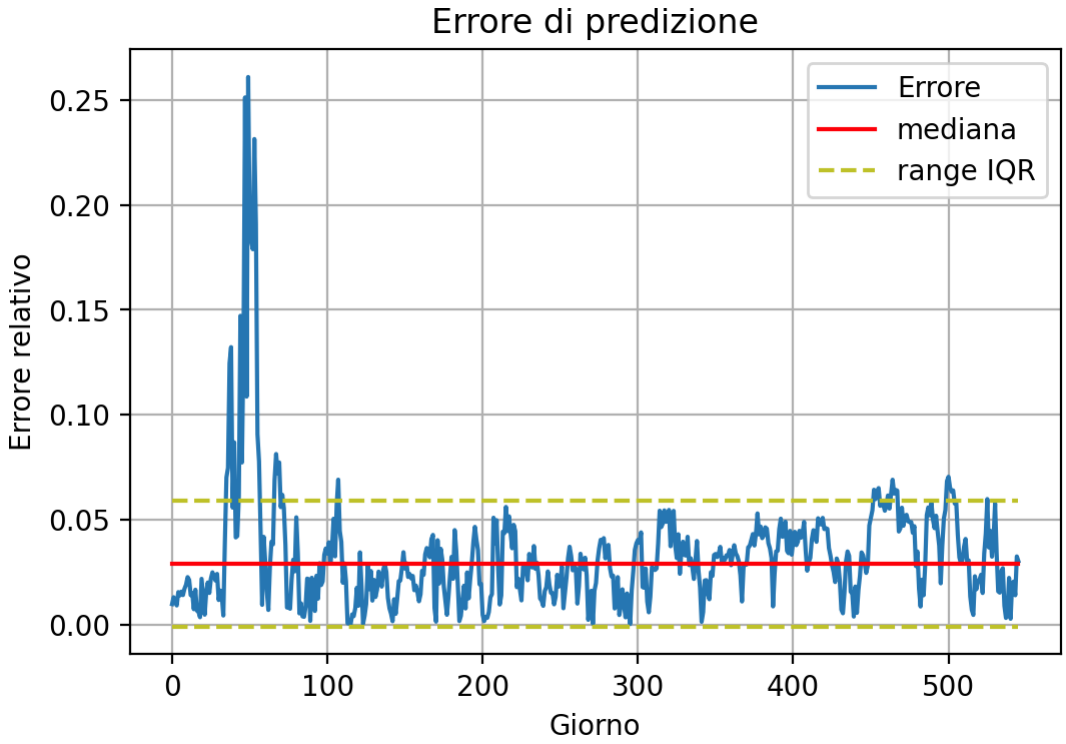

Figura 3.17: Errore modello LSTM, GRU utilizzando epochs=2, batch_size=8

3.5 Predizioni future

Infine, come possiamo vedere in Figura 3.18 sono andato a predire il valore di chiusura raggiunto dallo S&P 500 nei prossimi prediction_days, in questo caso ho utilizzato 3 giorni successivi futuri.

```
real_start = "2021-02-25"
real_end = str(date.today())

real_data  = web.DataReader([f'{stock_index}'], 'fred', real_start, real_end)

real_data.dropna(inplace = True)
real_prices = real_data[stock_index].values
real_prices = np.array(real_prices[-prediction_days:]).reshape((-1, 1))

real_prices = scaler.transform(real_prices)
real_prices = real_prices.reshape((1, -1))
future_predictions = model1.predict(real_prices)

#Predizioni per i prossimi tre giorni
future_predictions.reshape((-1, 1))
future_predictions = scaler.inverse_transform(future_predictions)
future_predictions
savePredictions("/content/gdrive/MyDrive/Colab Notebooks/Predizioni.txt", real_end + "LSTM", future_predictions)
```

Figura 3.18: Predizione di 4 giorni futuri

Dopo aver predetto i giorni li ho salvati in un file che ho chiamato "Predizioni.txt" nella cartella online in "/content/gdrive/MyDrive/Colab Notebooks/" di Google Drive tramite la funzione savePredictions definita in Figura 3.19

```
def savePredictions(path, date, values):
  '''Funzione chesalva le predizioni contenute nella variabile values
     all'interno del file indicato dalla stringa path
     path = (string) percorso al file
     date = (string) data delle previsioni
     values = (iterable float) predizioni future
  '''
  doc = open(path, "a+")
  doc.write(date + ":\t")
  for p in values[0]:
    doc.write(str(p) + "\t")

  doc.write("\n")
  doc.close()
```

Figura 3.19: Funzione per salvare le predizioni nel file Predizioni.txt

3.6 Risultati raggiunti

Effettuate le predizioni, ho salvato i dati nella tabella 3.20 che mostra le predizione dei prossimi 4 giorni futuri, mettendole in relazione ai valori reali assunti dall'indice in quei giorni.

	28-02-2022	01-03-2022	02-03-2022	03-03-2022
LSTM	4253.35	4163.8105	4167.5	4159.9116
GRU	4191.93	4251.8037	4226.6294	4214.6177
LSTM, GRU	4248.6123	4378.22	4391.566	4352.399
VALORI REALI	4373.74	4306.26	4386.54	4363.49

Figura 3.20: Tabella contenente predizioni e valori reali

Possiamo vedere visivamente che alcune predizioni non si discostano di molto dai valori reali. Successivamente ho proceduto con il calcolo dell'errore quadratico medio per ogni modello di rete, andando infine a calcolare la radice quadrata dei risultati ottenuti si ha mediamente l'errore in dollari della singola rete: 3.21.

```
LSTM      ((4253.35 - 4373.74)^2 + (4163.8105 - 4306.26)^2 + (4167.5 - 4386.54)^2 + (4159.9116 - 4363.49)) / 4 = 31052.0747
GRU       ((4191.93 - 4373.74)^2 + (4251.8037 - 4306.26)^2 + (4226.6294 - 4386.54)^2 + (4214.6177 - 4363.49)^2) / 4 = 20938.6816
LSTM, GRU ((4248.6123 - 4373.74)^2 + (4378.22 - 4306.26)^2 + (4391.566 - 4386.54)^2 + (4352.399 - 4363.49)^2) / 4 = 5245.8634
```

Figura 3.21: Calcoli effettuati per analizzare l'errore quadratico medio

Calcolando la radice dell'errore quadratico medio per ognuna delle reti, risulta che il modello ibrido LSTM, GRU commette il minor errore di predizione medio di 72.43 dollari, contro i 144.70 del modello GRU e i 176.22 del modello LSTM, che in questo caso commette un'errore medio più elevato.

3.7 Approcci alternativi

Un possibile approccio alternativo all'utilizzo di reti neurali LSTM e GRU è il modello ARIMA(Auto-Regressive Integrated Moving Average). Quest'ultimo è molto utilizzato in campo finanziario ed esprime un forte potenziale nelle previsioni del prezzo di mercato a breve termine, essendo dotato di robustezza ed efficienza. Sebbene alcune ricerche [5] affermano che ARIMA dia migliori risultati rispetto ai modelli LSTM e GRU, ci sono aspetti come parametri scelti e quantità di dati presi in analisi che possono influenzare i risultati. Inoltre ARIMA richiede una serie di parametri da calcolare (p, q, d) in base ai dati, mentre LSTM e GRU non richiedono l'impostazione di tali parametri. Una delle principali differenze che intercorre tra LSTM, GRU e ARIMA è che ARIMA tende a funzionare bene solo su serie temporali stazionarie (dove non c'è stagionalità, tendenza, ciclicità, irregolarità), quindi per poterlo utilizzare in modo corretto bisogna in primo luogo accertarsi che la serie storica presa in analisi sia stazionaria, ovvero quando possiede una media ed una varianza costante nel tempo, nel caso non lo fosse, bisogna assicurarsi di convertirla in modo appropriato prima di poter iniziare ad operare. Per capire quale tra questi modelli esprima un miglior risultato, in una determinata situazione, è possibile aggiungere funzionalità e metodiche al fine di migliorare l'accuratezza delle previsioni. La *"Sentiment Analysis"*, può essere d'aiuto a tal fine, consente di estrarre un "sentimento" positivo, negativo o neutrale attraverso l'analisi di documenti testuali generati in rete. Questa tecnica di *Machine Learning* fornisce l'apprendimento delle intenzioni degli investitori, chiarendo quale potrebbe essere il futuro andamento in relazione ad un determinato mercato azionario.

4 Conclusioni

Lo scopo di questo lavoro di tesi è stato lo studio e l'approfondimento di alcune delle principali tecnologie oggi utilizzate nel campo dell'intelligenza artificiale e *machine learning*. In particolare, ho preso come tema di riferimento l'analisi del mercato azionario per cercare di prevedere l'andamento del prezzo di un indice noto, lo "Standard & Poor's 500". Ho sviluppato tre tipologie di reti neurali LSTM, GRU, ibrida LSTM, GRU per cercare di capire quale fra le tre risultasse la più performante, nel ridurre l'errore di predizione, relativo ai valori reali del prezzo di chiusura giornaliero raggiunto dall'indice e nel predire il prezzo di valori futuri. Ho successivamente condotto vari test andando a modificare i parametri di allenamento del training set (epoche e batch_size), tenendo traccia dei risultati ottenuti. Dai risultati, emerge che la configurazione di tipo GRU necessita di un minor numero di epoche per essere allenata e dare un'errore di approssimazione molto piccolo. Infine, ho fatto prevedere al modello dei possibili valori futuri, sulla base di valori passati. Ho dunque calcolato di quanti dollari si discostavano mediamente le predizioni, ed è risultato che ha commesso un minor errore il modello ibrido LSTM, GRU, la combinazione dei due modelli potrebbe quindi portare a migliori predizioni future pur chiarendo che brusche variazioni del prezzo (es. guerre, covid, ecc.) diminuirebbero la precisione di predizione.

Bibliografia

[1] Artificial neural networks improve and simplify intensive care mortality prognostication: a national cohort study of 217,289 first-time intensive care unit admissions. https://jintensivecare.biomedcentral.com/articles/10.1186/s40560-019-0393-1.

[2] Forecasting daily and sessional returns of the ise-100 index with neural network models. http://journal.dogus.edu.tr/index.php/duj/article/viewFile/86/pdf_eavci.

[3] Illustrated guide to lstm's and gru's. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

[4] Supervised, unsupervised, reinforcement learning. https://www.phdata.io/blog/difference-between-supervised-unsupervised-reinforcement-learning.

[5] Un confronto tra arima, lstm e gru per la previsione delle serie temporali. In *ACAI 2019*.

[6] S Agatonovic-Kustrin and Rosemary Beresford. Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research. *Journal of pharmaceutical and biomedical analysis*, 22(5):717–727, 2000.

[7] Tesi di Laurea Triennale, Riccardo Martoglia, and Mattia Savoia. Università degli studi di modena e reggio emilia.

[8] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data mining and knowledge discovery*, 33(4):917–963, 2019.

[9] H Jabbar and Rafiqul Zaman Khan. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, 70, 2015.

[10] Will Koehrsen. Overfitting vs. underfitting: A complete example. *Towards Data Science*, 2018.

[11] Yiwei Liu, Zhiping Wang, and Baoyou Zheng. Application of regularized gru-lstm model in stock price prediction. In *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, pages 1886–1890. IEEE, 2019.

[12] Diego Mancuso. Le reti neurali come strumento di discriminazione: una rassegna della letteratura. *Rivista di Statistica Applicata*, 1998.

[13] Elena Loli Piccolomoni and Alessandro Fabbri. Reti neurali in ambito finanziario.

[14] Kamilya Smagulova and Alex Pappachen James. *Overview of Long Short-Term Memory Neural Networks*, pages 139–153. Springer International Publishing, Cham, 2020.

[15] Jinghua Zhao, Dalin Zeng, Shuang Liang, Huilin Kang, and Qinming Liu. Prediction model for stock price trend based on recurrent neural network. *Journal of Ambient Intelligence and Humanized Computing*, 12(1):745–753, 2021.