UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

FACOLTÀ DI SCIENZE MM. FF. NN.

DIPARTIMENTO DI INFORMATICA
CORSO DI LAUREA IN INFORMATICA

ESAME DI
INGEGNERIA DELLA CONOSCENZA

REALIZZAZIONE, APPRENDIMENTO E TESTING DI STOCK-PRICE-DICTOR, SOFTWARE PER LA PREVISIONE DEL PREZZO DI CHIUSURA DI UN TITOLO IN BORSA

SUPERVISIONE

CHIAR.MO PROF. NICOLA FANIZZI

STUDENTI

FRANCESCO FARINOLA
GAETANO DE GENNARO

INDICE

1. Introduzione

Prevedere le performance del mercato azionario è una delle cose più difficili da fare. Ci sono tanti fattori coinvolti nella previsione: fisici, psicologici, razionali e irrazionali. Tuttavia, è possibile applicare tecniche di machine learning e deep learning, che combinate con l'utilizzo di indicatori e news sul mercato, rendono più semplice ed accurata la previsione.

I metodi tradizionali sulle serie storiche come ARIMA e altri modelli stocastici sono efficienti solo quando le serie sono stazionarie, infatti questi metodi prevedono che i dati siano processati prendendo i loro logaritmi. Questo problema viene risolto con le reti neurali. In particolare, in questo progetto vengono utilizzate LSTM e MLP, nonché le migliori reti per serie storiche di dati.

A supporto di questi modelli è possibile utilizzare la simulazione Monte Carlo per comprendere l'impatto del rischio e dell'incertezza.

2. Obiettivi

Con la realizzazione di questo software, sono state messe in pratica alcune delle tecniche messe a disposizione dal deep learning. L'obiettivo di questo software è quello di prevedere il prezzo di chiusura del giorno successivo di un titolo qualsiasi in borsa, analizzando il prezzo di chiusura dei precedenti trenta giorni. Infine, attraverso l'implementazione di alcuni indicatori, viene consigliato l'acquisto o la vendita delle azioni.

3. Realizzazione del Software Stock-Price-Dictor

3.1 Metodi e strumenti utilizzati

Per la realizzazione di Stock-Price-Dictor è stato utilizzato un linguaggio piuttosto comune per il machine learning: Python (v 3.6.x).

3.1.1 PyCharm

Lo sviluppo è stato realizzato mediante pyCharm, un ambiente di sviluppo professionale del gruppo Intellij IDEA, ottenibile al seguente link:
https://www.jetbrains.com/pycharm/download/#section=windows

3.1.2 Qt Designer

Software presente nel pacchetto Qt (https://www.qt.io/download). Attraverso Qt Designer è stato possibile realizzare in maniera estremamente rapida ed efficiente la GUI del software oggetto di discussione. Il software consente l'esportazione dell'interfaccia creata in un file contenente codice XML. Grazie alla libreria PyQt5 (descritta nel paragrafo 3.1.2), è stato poi possibile convertire il file appena esportato in codice Python.

3.1.3 Yahoo Finance API

Per l'acquisizione dei dati finanziari in tempo reale sono state utilizzate le API di Yahoo Finance. Queste consentono in pochi istanti e in maniera gratuita il download dei dati richiesti.

3.1.4 Librerie Python

- **Pandas**: utilizzata per la lettura di file csv. e per la gestione dei dataframe.
- **Numpy**: libreria per il calcolo scientifico e per la gestione di array N-dimensionali.
- **Tensorflow**: libreria open-source per l'apprendimento automatico. Dispone di un insieme di strumenti, librerie e risorse che consente a chi ne fa uso di realizzare e distribuire algoritmi basati su machine learning
- **Keras**: libreria per le reti neurali compatibile con TensorFlow. Essa consente una prototipazione semplice e veloce, supporta reti convoluzionali e reti ricorrenti.
- **Yfinance**: libreria che mette a disposizione API per ottenere dati finanziari da Yahoo Finance.
- **Matplotlib**: libreria in grado di realizzare grafici 2D. Essa possiede una vasta gamma di grafici (a linee, a barre, a torta, istogrammi…). In questo progetto è stata utilizzata per la visualizzazione grafica di dati storici, e per il confronto tra i risultati ottenuti.
- **Sklearn**: libreria semplice ed efficiente per l'analisi dei dati e il data mining. Utilizzata in questo progetto per normalizzare i dati.
- **PyQt5**: insieme di librerie multipiattaforma che implementano API di alto livello per accedere a molti aspetti dei moderni sistemi desktop e mobili. Tra le innumerevoli funzionalità prevede moduli per la gestione di GUI.

4. Preparazione dei dati

4.1 Acquisizione

Nel file *DataAcquisition.py* è stata definita la funzione *get_data(ticker, start_date, end_date)*. La funzione salva nel dataframe alldata i valori per ciascun giorno di trading di "*Open*", "*Close*", "*High*", "*Low*", "*Adj Close*" e "*Volume*" tramite le librerie *yfinance* e *pandas_datareader*.

Es: *alldata = pdr.get_data_yahoo("AAPL", "2010-01-01", "2019-09-02")*.

Questa riga di codice salverà i dati di trading di AAPL dal "01-01-2010" al "02-09-2019" in *alldata*.

Inoltre nel file *stock_prices.csv* vengono esportati i valori di "*Adj Close*" che saranno utilizzati nella fase di training e testing.

Parametri di input

I parametri passati a tale funzione sono i seguenti:

- *ticker*: nome del titolo azionario dal quale si vuole acquisire dati (es. AAPL per Apple)

- *start_date*: data rappresentante l'inizio del periodo di acquisizione dati

- *end_date*: data rappresentante la fine del periodo di acquisizione dati

Per *start_date*, è stata scelta una data non troppo lontana dal periodo attuale. Così facendo vengono presi in considerazione dati non troppo distanti dai valori recenti, al fine di evitare gli esempi che non rispecchiano più il mondo reale.

Per *end_date* verrà utilizzata la funzione *.today()* della libreria *datetime* per assegnare il valore del giorno corrente.

Per tale ragione il valore di default è stato impostato nel file *LSTM.py* a "*2010-01-01*"

Paramtri di output

- *alldata*: dataframe contenenti i dati finanziari che vanno da *start_date* a *end_date* per *ticker*.

4.2 Preprocessing

Il file *Preprocessing.py* implementa la classe *DataProcessing* che permette la suddivisione dei dati acquisiti dal file *stock_prices.csv* in dati di training e dati di testing.

I parametri passati al costruttore sono:

- *filename*: contiene il nome del file .csv contenente i dati storici
- *perc*: percentuale di suddivisione dei dati tra dati di training e dati di testing. Di default 90%

Un esempio di creazione di un oggetto di classe DataProcessing è il seguente:

process = DataProcessing("stock_prices.csv", 0.9)

La classe inoltre contiene due funzioni: generate_train(window) e generate_test(window).

Funzione generate_train(window)

La funzione *generate_train* possiede come parametro di input *window*, che rappresenta il numero di giorni da utilizzare per predire il successivo. Quindi nel training, saranno utilizzati *window* giorni per predire il *(window+1)-esimo* giorno.

Pertanto questa funzione produce due vettori: *X_train*, contenente le features di input, e *Y_train* contenente le features di output.

Esempio con *window=5*:

Data	Close Adj	I	II	III
2019-01-01	156.05	X1	-	-
2019-01-02	140.51	X1	X2	-
2019-01-03	146.50	X1	X2	X3
2019-01-04	146.18	X1	X2	X3
2019-01-05	148.96	X1	X2	X3
2019-01-06	151.49	Y1	X2	X3
2019-01-07	151.98	…	Y2	X3
2019-01-08	150.49	…	…	Y3

Analogamente per *generate_test(window)*.

4.3 Normalizzazione

L'obiettivo principale della normalizzazione dei dati è quello di garantire la qualità di questi prima di darli in pasto a qualsiasi algoritmo di learning. Questo processo velocizza

e stabilizza il processo di training perché i dati vengono posti tutti all'interno di una stessa scala.

In questo progetto verrà utilizzata la normalizzazione min-max fornita dalla libreria sklearn:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

Per poter normalizzare i dati è stato definito uno scaler che trasforma i dati in un intervallo compreso tra 0 e 1.

sc = MinMaxScaler(feature_range=(0,1))

Per applicare lo scaler è stata utilizzata la seguente funzione:

sc.fit_transform(…)

Per eseguire la trasformazione inversa, e ottenere così i valori originari, si utilizza la funzione:

sc.inverse_transform(…)

5. Training e Testing

5.1 Training

In questo progetto sono stati utilizzati due modelli di deep learning: **MLP** (MultiLayer Perceptron) e **LSTM** (Long Short Term Memory)

5.1.1 MLP

MLP è un modello di rete neurale artificiale che mappa insiemi di dati in ingresso in un insieme di dati in uscita appropriati. Usa una tecnica di apprendimento supervisionato chiamata back-propagation.

E' fatta di strati multipli di nodi in un grafo diretto, con ogni strato completamente connesso al successivo. Eccetto che per i nodi in ingresso, ogni nodo è un neurone con una funzione di attivazione non lineare.

L'addestramento avviene presentando alla rete pattern di cui è noto il valore di uscita e propagando (forward-propagation) gli input verso gli output attraverso dei livelli nascosti (hidden layers).

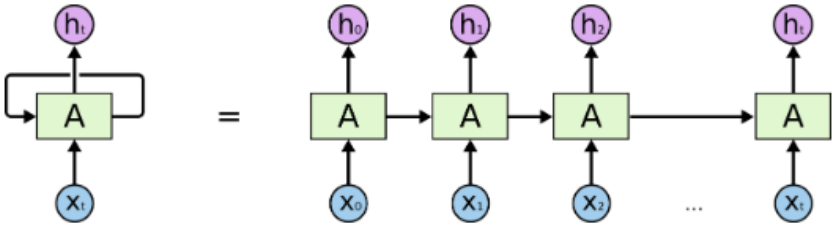
La differenza tra l'output prodotto dalla rete e quello desiderato è l'errore della rete, e l'obiettivo di questo algoritmo è di modificare i pesi della rete in modo da minimizzare l'errore medio sui pattern del training set (X_train). All'inizio dell'addestramento i pesi sono inizializzati con valori random, e successivamente vengono modificati in direzione opposta al gradiente dell'errore.

Per la creazione del modello è stata utilizzata la libreria *keras*. Essa ha concesso la creazione di una rete neurale composta da 2 hidden layers, con 100 neuroni ciascuno, e un output layer con un solo neurone. Questo perché in questo problema di regressione c'è la necessita di un solo valore in output (Adj Close).

5.1.2 LSTM

Le reti LSTM sono un tipo particolare di RNN (Recurrent Neural Network) e sono specializzate nell'evitare il problema delle dipendenze a lungo termine. LSTM contiene informazioni che possono essere scritte e lette fuori dal normale flusso delle reti ricorrenti in una cella.

Durante la fase di training la cella impara quali dati fare entrare, uscire o cancellare tramite il processo di back-propagation



Nella creazione di una LSTM, come per MLP, è molto importante il numero di layer e di neuroni da utilizzare negli hidden layer.

Nel nostro caso sono stati utilizzati 2 hidden layer, con il primo hidden layer con un numero di neuroni pari a:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

Dove:

- N_i è il numero di neuroni di input (nel nostro caso pari a *window*, cioè 30)
- N_o è il numero di neuroni di output, che come per MLP è uno solo

- N_s è il numero di esempi nel training set

- α è un numero arbitrario tra 2 e 10

Nel nostro caso, sono stati effettuati dei test con numero di neuroni compresi tra 54 e 11 e scelto la performance migliore, ossia 50.

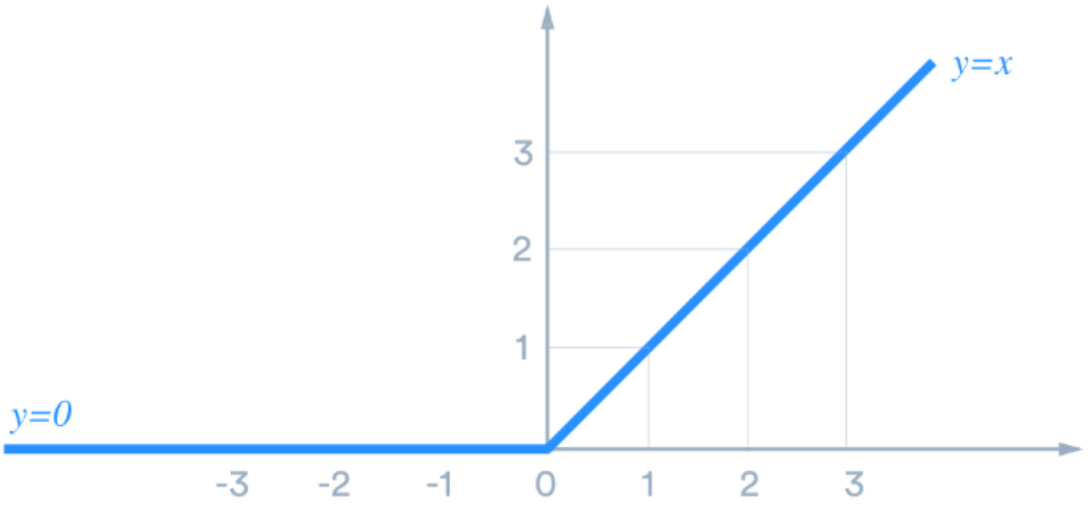
Per l'hidden layer successivo invece è stato scelto un numero di neuroni inferiore a 2/3 del layer precedente, quindi nel nostro caso è stato scelto 30.

5.1.3 Funzioni di attivazione

Nell'addestramento delle reti neurali tramite retro-propagazione dell'errore mediante discesa stocastica del gradiente, ogni parametro riceve ad ogni iterazione un aggiornamento proporzionale alla derivata parziale della funzione di perdita rispetto al parametro stesso.

Funzioni di attivazione non lineari classiche, come la **tangente iperbolica** o la **funzione logistica**, hanno gradiente a valori nell'intervallo [0,1] e poiché nell'algoritmo di retro-propagazione i gradienti ai vari livelli vengono moltiplicati tramite la regola della catena, il prodotto di n numeri in [0,1] decresce esponenzialmente rispetto alla profondità n della rete. Questo crea il problema di scomparsa di gradiente (**vanishing gradient**). Per risolvere questo problema, utilizzeremo la funzione di attivazione **ReLu (Rectified Linear)**:

$$f(net) = max(0,net)$$

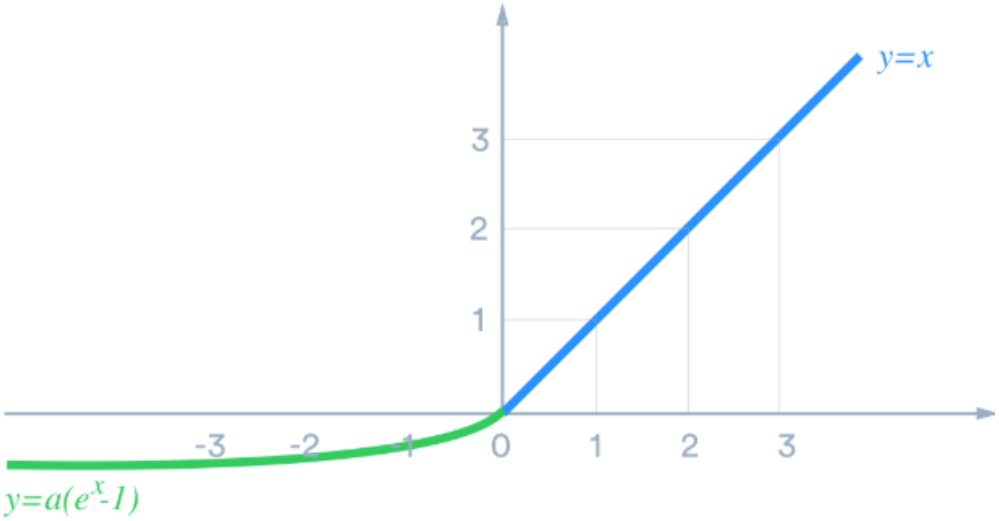


La derivata vale 0 per valori negativi o nulli di net e 1 per valori positivi quindi per valori positivi non abbiamo nessuna saturazione. Questo porta ad attivazioni sparse (parte dei neuroni sono spenti) che possono conferire maggiore robustezza.

Tuttavia, la funzione di attivazione ReLu potrebbe determinare la 'morte' di alcuni neuroni se i valori si bloccano nella parte negativa o nulla e, di conseguenza, avremo un output uguale a 0. Riducendo il learning rate si potrebbe risolvere il problema, ma è possibile applicare delle varianti della ReLu. Ad esempio, si potrebbe applicare la funzione **SELU (Scaled ELU)** che crea una pendenza logaritmica nella parte negativa della funzione ReLu:

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & per\ x < 0 \\ x & per\ x \geq 0 \end{cases}$$

La funzione SELU non solo risolve il problema di 'dying ReLu' ma velocizza il training avendo valori più vicini allo 0.



5.1.4 Loss Function e Ottimizzatori

La funzione model.compile() configura il processo di learning. Vengono passati 3 parametri:

- **Loss Function**: è la funzione di errore che il modello cercherà di minimizzare. Per problemi di regressione in questo dominio viene utilizzato l'errore quadratico medio (Mean Squared Error):

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

- **Metriche**: per problemi di regressione le metriche che si possono calcolare sono l'errore quadratico medio MSE e l'errore assoluto medio MAE. Il valore di queste metriche verrà visualizzato per ciascuna epoca in un grafico per visualizzare l'andamento della fase di training e individuare eventuali casi di sovradattamento.
- **Ottimizzatore**: permette di minimizzare la loss function e sono:
    1. *RMSprop*: è raccomandabile lasciare i parametri di default di questa funzione a eccezione del learning rate. Sono stati fatti dei test con differenti valori di learning rate e selezionato il migliore per il confronto (lr = 0.001)
    2. *Adam*: è una combinazione di RMSprop e lo Stochastic Gradient Descent con momentum. Questo usa una media mobile quadratica esponenziale v_t dei precedenti gradienti per scalare il valore di learning rate e una media mobile esponenziale m_t dei precedenti
    $$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
    $$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
    gradienti per una stima del momentum.
    3. *AdaMax*: è una variante di Adam che applica la norma infinita l_∞ alla media mobile quadratica esponenziale v_t per scalare il learning rate
    4. *Nadam*: è una combinazione di RMSprop e momentum. Nadam utilizza il valore v_t di Adam per il learning rate, mentre modifica il valore del momentum m_t applicando il gradiente accelerato di Nesterov. Per valori di learning rate piccoli, Nesterov diventa equivalente al normale momentum e, quindi, nel nostro caso non dovrebbero esserci grandi cambiamenti di performance rispetto ad Adam.

5.1.5 Regolarizzatori

Nella creazione dei modelli, viene utilizzato il **Dropout**, una tecnica di regolarizzazione per reti neurali. Dropout seleziona casualmente un numero di neuroni da ignorare durante il training. Questo significa che il contributo di questi neuroni per l'attivazione degli strati inferiori viene temporaneamente rimosso. Questi neuroni non prendono parte né nel forward pass né nel backword pass per l'aggiornamento dei pesi. L'effetto prodotto

è quello di una rete neurale meno sensibile a specifici pesi dei neuroni, ottenendo un migliore generalizzazione che previene sovradattamento dei dati di training.

Generalmente si sceglie un dropout compreso tra il 20 e il 50% e, ovviamente, sarà necessario usare una rete più grande poiché alcuni neuroni non verranno usati.

5.1.6 Struttura delle reti

Dopo vari test, utilizzando le varie funzioni di attivazione, ottimizzatori e dropout, e dopo aver migliorato i parametri di questi, sono stati ottenuti i seguenti modelli:

MLP:

```
model = Sequential()
model.add(Dense(100, activation=tf.nn.selu))
model.add(Dropout(0.2))
model.add(Dense(100, activation=tf.nn.selu))
model.add(Dropout(0.2))
model.add(Dense(1, activation=tf.nn.selu))
```

LSTM:

```
model = Sequential()
model.add(Dense(50, input_shape=(X_train.shape[1],1), return_sequences=True)
model.add(Dropout(0.2))
model.add(Dense(30)
model.add(Dropout(0.2))
model.add(Dense(1, activation=tf.nn.selu))
```

5.1.7 Compilazione e fase di training

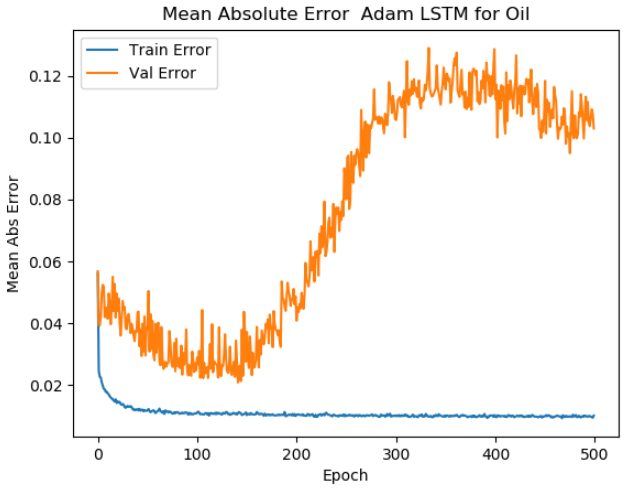
Il modello viene compilato per configurare il processo di training:

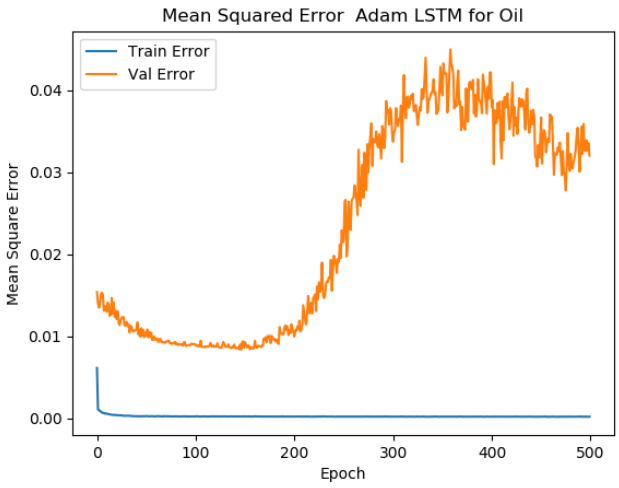
```
model.compile(optimizer=”Adam”,
metrics=[‘mean_absolute_error’,’mean_squared_error’],
loss=”mean_squared_error”)
```

Per poter lanciare il training si utilizza la funzione *model.fit()* di *keras* a cui vanno passati come parametri le feature di input (*X_train*), le features di output (*Y_train*), il numero di epoche, la percentuale di divisione del training set in train e validation set e la dimensione del batch. In questo progetto vengono utilizzate 500 epoche (i dati di training vengono processati dalla rete 500 volte), una *validation_split* di 0.2 (80% training set e 20% validation set) e una *batch_size* di 30 (30 esempi di training alla volta).

```
history = model.fit(X_train, Y_train, epochs=500,
validation_split=0.2, batch_size=30)
```

In *history* vengono salvati i valori di errore MSE e MAE per ciascuna epoca. Attraverso la funzione *plot_history(storia)* a cui va passato il modello allenato, è possibile analizzare tramite i valori d'errore l'andamento del training e se questi producono sovradattamento. La funzione stampa i grafici relativi all'errore assoluto e l'errore quadratico in relazione alle epoche dei training e del validation set. Nel caso di LSTM possiamo notare che l'errore, con l'aumentare delle epoche aumenta:

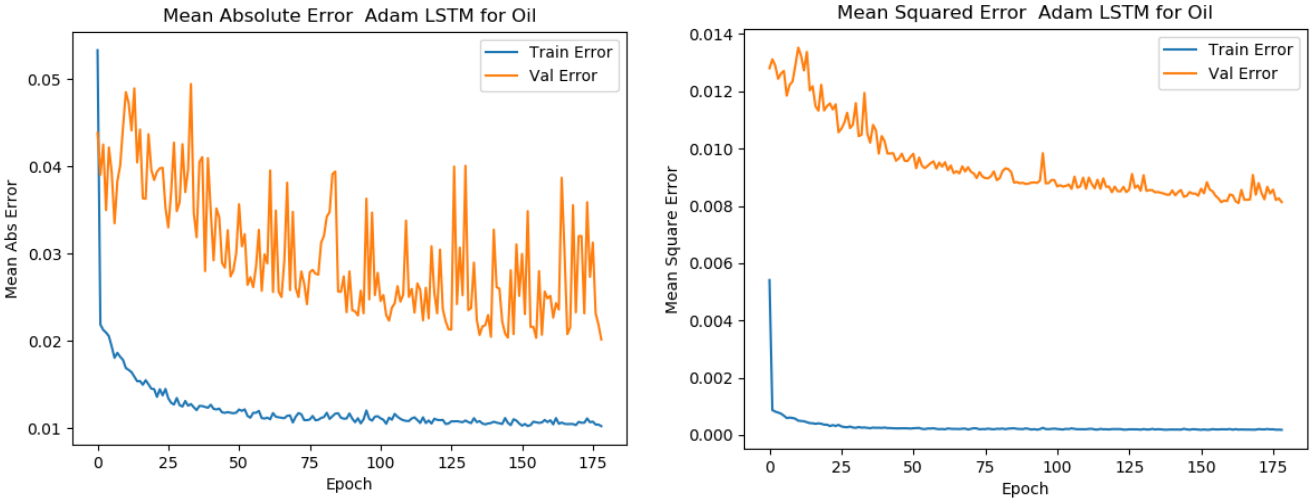




Per far fronte a questo problema verrà utilizzato l'EarlyStopping callback che permette di fermare l'apprendimento del modello quando il valore della loss function quando questo non effettua miglioramenti per determinato numero di epoche, evitando che il valore di loss aumenti in epoche future. Viene definito quindi l'early stop e passato come parametro nella funzione fit. Il parametro *patiance* determina il numero di epoche di cui controllare il valore di loss.

```
early_stop = EarlyStopping(monitor='val_loss',patiance=15)
model.fit(...,callbacks=[early_stop],...)
```

Dopo aver settato una patience di 30 epoche, il modello LSTM si è fermato a un numero ragionevole di epoche (175) ottenendo i seguenti grafici di errori:



5.2 Valutazione

Per valutare il modello sui dati di test invece viene utilizzata la funzione *model.evaluate(X_test,Y_test)*. Sono stati così ottenuti i valori di loss (MSE) e di MAE per il test set.

5.3 Testing

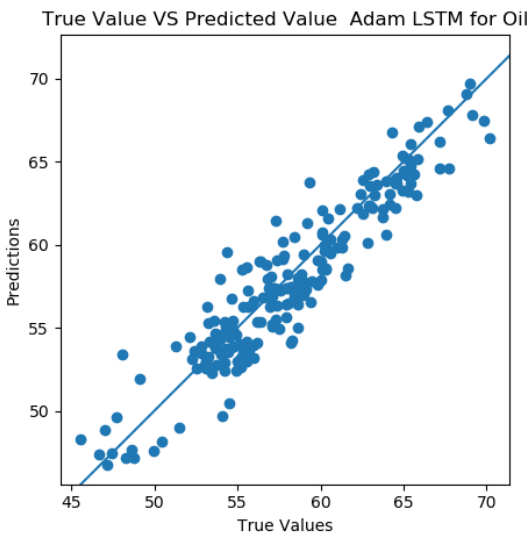
È possibile effettuare le predizioni sul test set attraverso la funzione *predict(X_test)*. I risultati vengono immagazzinati in una variabile *test_prediction*. Attraverso la funzione *flatten*() l'array risultante viene ridotto ad un'unica dimensione. Inoltre, i dati risultanti sono normalizzati. Per denormalizzarli basta utilizzare la funzione inversa di *sklearn: inverse_transform()*

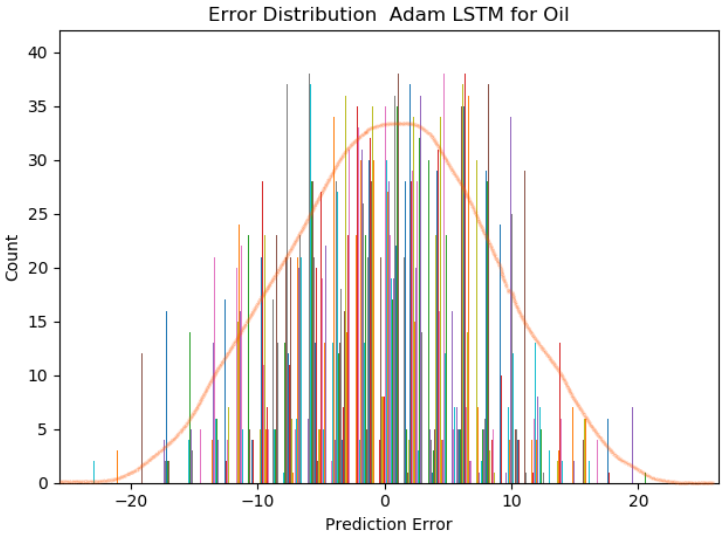
```
test_predictions=sc.inverse_transform(model.predict(X_test)).flatten()
```

Per controllare le performance delle predizioni sono stati creati due tipi di grafici:

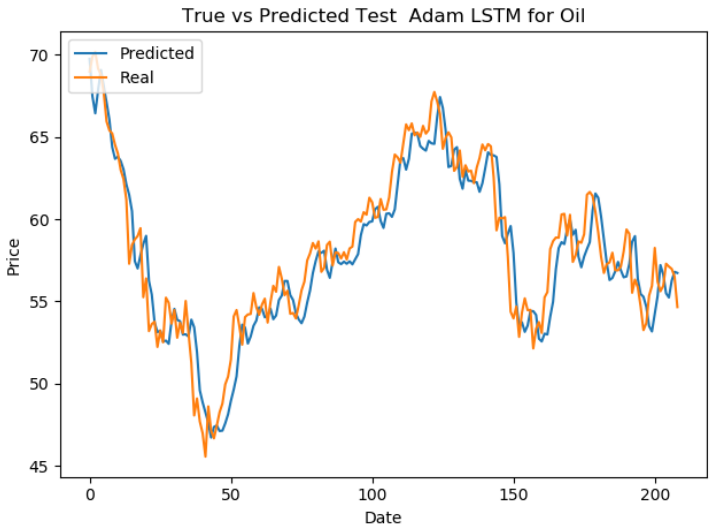
1. Il primo è un grafico di dispersione dove i valori reali e i corrispondenti valori predetti vengono riportati sullo spazio cartesiano. Viene disegnata una linea di identità x=y. È possibile osservare che più i dataset corrispondono, più i punti tendono a concentrarsi sulla linea d'identità
2. Il secondo invece è un istogramma che rappresenta la distribuzione dell'errore nelle predizioni. Per analizzare correttamente questo grafico è necessario

disegnare la distribuzione Gaussiana e vedere se l'istogramma prodotto ha una forma simile a quest'ultima





Inoltre, è stato stampato un altro grafico dove vengono disegnate le linee di prezzo reali (in arancio) e predetti sul test set per effettuare un confronto.



Tutti questi grafici sono stati utili per effettuare un confronto tra i due modelli realizzati (MLP e LSTM) con i vari parametri ottimizzati.

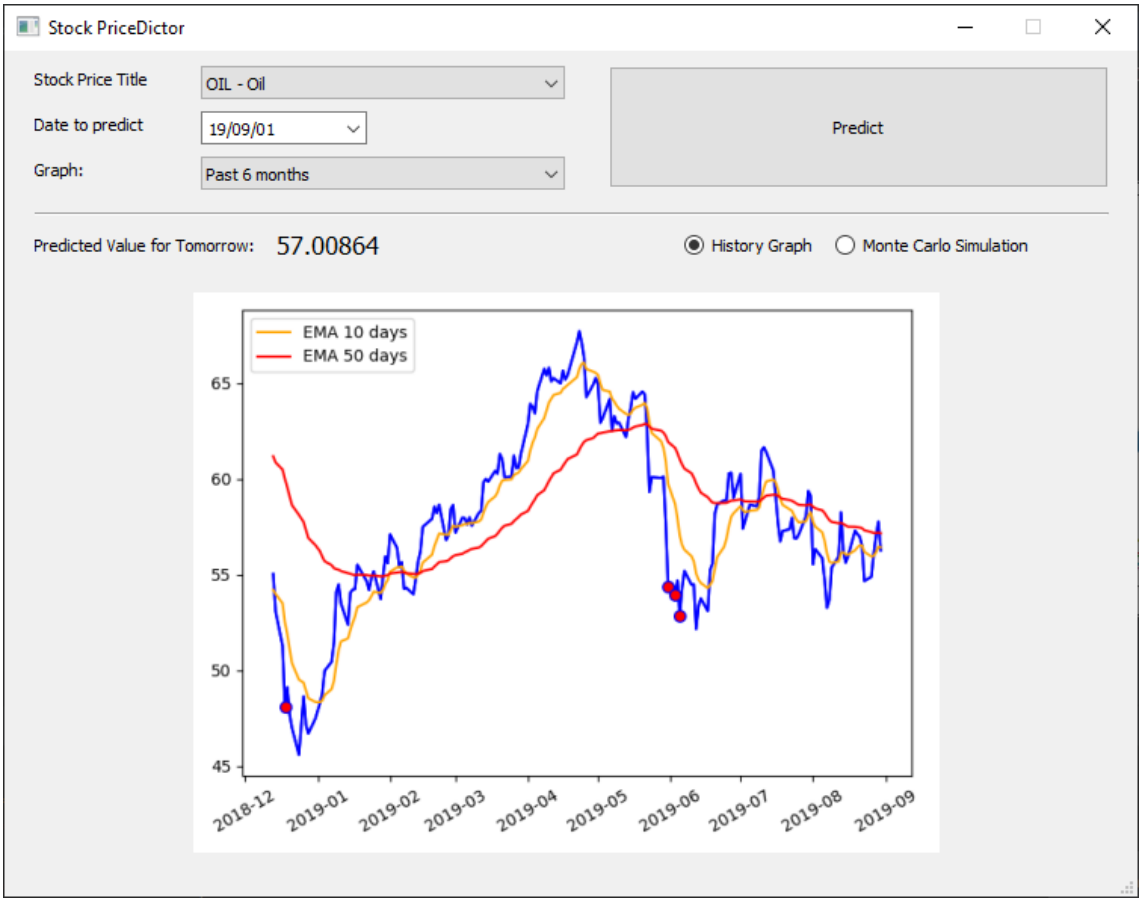
6. GUI

Come precedentemente accennato, la GUI di Stock-Price-Dictor è stata realizzata mediante il supporto di Qt Designer, che ha concesso la creazione dell'interfaccia mediante drag and drop dei vari componenti. Successivamente attraverso *PyQt5*, l'interfaccia è stata esportata in codice Python.

L’interfaccia utente è composta dai seguenti componenti:

- “Stock Price Title”: *QComboBox* che consente la selezione di un titolo tra quelli presenti nel menu a tendina.
- “Date to predict”: *QDateEdit* che consente la selezione di una data per la quale si desidera fare predizione. E’ possibile scegliere fino ad un massimo di 10 giorni antecedenti alla data attuale.
- “Graph”: *QComboBox* che consente di scegliere la scala del grafico “History Graph”. Si può scegliere da “Past 10 days” a “Past 10 years”
- “Predict”: *QPushButton* che avvia la computazione e predice il valore di chiusura del titolo selezionato.
- “Predicted value for tomorrow”: *QLabel* che al termine della computazione mostra il valore predetto per il giorno scelto.
- “History Graph”: *QRadioButton* che consente la visualizzazione del grafico che mostri la storia nel periodo storico selezionato. Oltre alla storia (linea blu) sono stati aggiunti due indicatori: la media mobile esponenziale per 10 giorni (linea arancione), e la media mobile esponenziale per 50 giorni (linea rossa). Quando queste due linee si intersecano, si prevede un cambio di direzione nella curva del grafico (linea blu) verso l’alto o verso il basso.

  Sono anche stati posizionati dei markers, rossi o neri, che rappresentano rispettivamente opportunità di acquisto o vendita del titolo corrispondente. Tali markers vengono posizionati quando il prezzo del titolo corrente, supera o viene superato dal valore massimo assoluto della borsa moltiplicato per 0.05.
- “Monte Carlo Simulation”: *QRadioButton* che consente la visualizzazione del grafico dei prezzi futuri utilizzando il campionamento casuale Monte Carlo



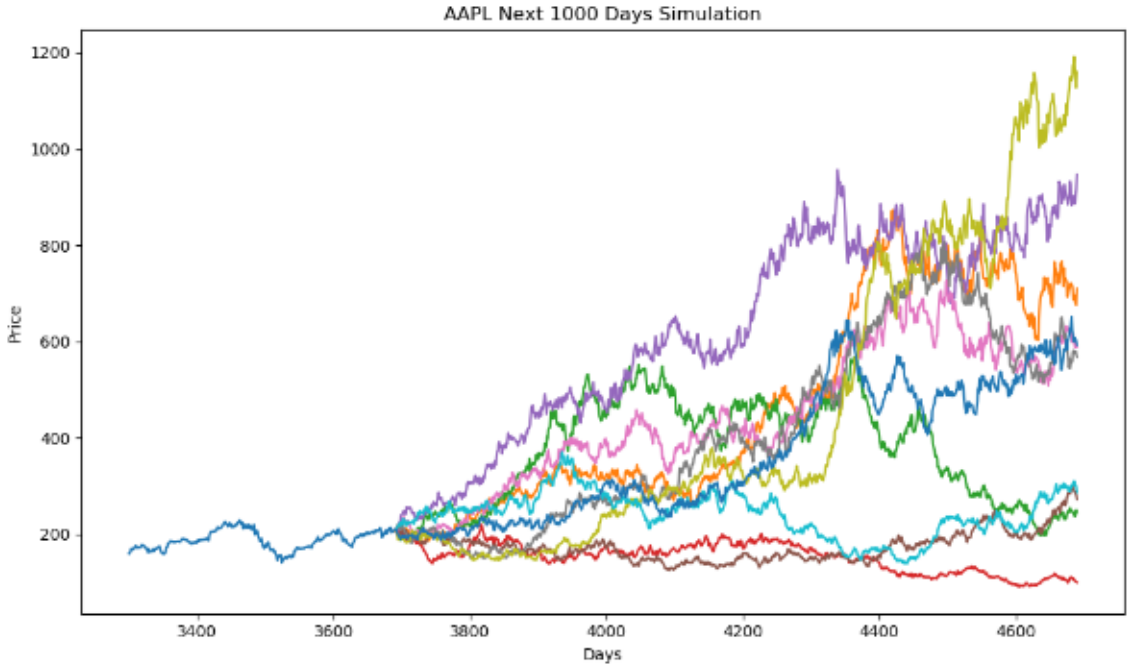
6.1 Campionamento Monte Carlo

Le simulazioni Monte Carlo vengono utilizzare per modellare la probabilità di risultati diversi in un processo che non può essere previsto facilmente a causa dell'intervento di variabili casuali. È utilizzato per comprendere l'impatto del rischio e dell'incertezza. Nel mercato finanziario ci sono due componenti che nei movimenti dei prezzi di un'attività: la deriva (drift) che è un movimento direzionale costante, un input casuale che rappresenta la volatilità del mercato. Osservando i dati storici è possibile calcolare la deriva, la deviazione standard, la varianza e lo spostamento medio dei prezzi di un titolo.

Generando un numero arbitrario di simulazioni (10), è possibile valutare la probabilità che il prezzo di un titolo segua una determinata traiettoria. La probabilità che il rendimento effettivo sia compreso in una deviazione standard del tasso più probabile (previsto) è del 68%; che sia compreso tra due deviazioni standard è del 95%; tra tre deviazioni standard è del 99,7%. Tuttavia, non vi è alcuna garanzia che si verifichi il risultato atteso in quanto le simulazioni ignorano tutto ciò che fa muovere i prezzi (macro-tendenze, fattori ciclici etc.).

La generazione di campioni è influenzata da valori generati casualmente a cui viene applicata la distribuzione inversa. Per la generazione dei prezzi verrà usata una catena di Markov, dove lo stato iniziale $S_0(=S_t)$ rappresenta l'ultimo prezzo di chiusura (giorno corrente) e lo stato S_{t+1} rappresenta il campione simulato calcolato tramite il moto Browniano geometrico: $S_{t+1}= S_t * r$

dove r è calcolato dalla formula r = deriva + (standard_deviation * $e^{\text{valore casuale}}$)



6. Risultati e Conclusioni

Sono stati effettuati molteplici test sui titoli azionari "AAPL", "MSFT", "AMZN", "CSCO" e "OIL". Sono stati scelti questi titoli perché nel tempo hanno mostrato comportamenti differenti e quindi è stato possibile analizzare meglio il comportamento degli algoritmi.

In particolare, sono stati effettuati test per trovare il numero ottimale di layer e neuroni nella struttura delle reti neurali; sono stati testati i diversi tipi di funzioni di attivazione, ottimizzatori, regolarizzatori, numero di epoche e tecniche di early stopping.

Per quanto riguarda la scelta dell'algoritmo, è stato notato che i modelli allenati con MLP hanno un errore quadratico di ordine maggiore rispetto ai modelli LSTM. Di

conseguenza il modello LSTM, come è stato verificato successivamente, fornisce predizioni più accurate.

Per quanto riguarda le funzioni di attivazione, inizialmente è stata utilizzata la funzione di attivazione ReLU, ma dopo alcuni test è stato notato il verificarsi del problema di 'dying ReLu' per cui è stata applicata una variante di questa (SELU) che gestisce meglio i valori negativi, non azzerandoli.

Per quanto riguarda il numero di neuroni e l'utilizzo di regolarizzatori, è stato notato che utilizzando come regolarizzatore il dropout (con 0.2) e ,aumentando il numero di neuroni a 60 per il primo hidden layer e 40 per il secondo, si ottengono performance migliori.

Sono stati poi testati i diversi ottimizzatori per minimizzare l'errore. Con gli ottimizzatori RMSprop e Adamax sono state riscontrate performance pessime mentre sono state ottenute performance simili con gli ottimizzatori Adam e Nadam. Tuttavia, è stato notato che Adam sembri più stabile con l'aumentare delle epoche a differenza di Nadam, anche se quest'ultimo decresce molto più velocemente raggiungendo il minimo globale il tempi più brevi rispetto ad Adam.

Inoltre, se si utilizzano troppe epoche nel modello LSTM si verifica l'aumento dell'errore quadratico medio, circa dopo circa 170 epoche. Si è tentato così di applicare l'early stopping. Nonostante i diversi tentativi con valori di pazienza differenti non è stato possibile trovare una pazienza ottimale per poter ottenere un numero di epoche ragionevole. I vari titoli si comportavano in modo differente così si è deciso di stabilire un numero di epoche per il training uguale a 100.

7. Sviluppi futuri

Siccome sono numerosi i fattori che influenzano il mercato finanziario, è possibile estendere il modello creato aggiungendo conoscenza al modello tramite, ad esempio, analisi di news positive e negative per un determinato titolo, implementazione di indicatori per l'analisi tecnica, riconoscimento di pattern ciclici e effetti del tasso d'interesse sui titoli.