LIBERA UNIVERSITÀ INTERNAZIONALE DEGLI STUDI SOCIALI
"LUISS - GUIDO CARLI"

DIPARTIMENTO DI ECONOMIA E FINANZA

Corso di Laurea in Economia e Finanza

Reinforcement Learning for Algorithmic Trading: DDQN

RELATORE:

PROF. FEDERICO CARLO EUGENIO CARLINI

CORRELATORE:

PROF. NICOLA BORRI

CANDIDATO:

ANDREA CHIAVERINI

Anno Accademico 2022-23

Al Me di dieci anni fa,
determinazione, impegno, costanza, sacrifici e testa dura
sono tutti gli ingredienti che servono per portarti da un punto A ad un punto B,
non è mai troppo tardi per cambiare strada.
Ce l'hai fatta!

Ai miei Affetti,
a quelli di sempre, ai nuovi, a quelli Veri,
a quelli che conoscono anche l'altro lato di Me, ma restano comunque...
Che senso ha raggiungere un traguardo, se non hai nessuno con cui gioire?!
A voi!

Alla mia Famiglia,
che ha vissuto tutti i miei sbalzi di umore durante questo lungo percorso,
rimanendo sempre Famiglia.
Grazie!

Al mio Relatore,
molto di più di un semplice insegnante.
Insegnare bene è importante, ma ispirare è molto di più…
La ringrazio, veramente!

Al mio Datore di Lavoro,
non solo un "capo", ma una guida,
un mentore...

Al domani,
che abbia in serbo altre sfide e altri stimoli, non solo lavorativi...
Perché un domani "piatto" sarebbe la cosa più noiosa di tutte,
ad un domani pieno di sorprese e pieno di "guai"

“Life is one big training set.”

David Silver

INDICE

CAPITOLO 2

DEFINIZIONE DEL PROBLEMA & ARCHITETTURA DEL MODELLO

CAPITOLO 3

RISULTATI

CAPITOLO 4

CONCLUSIONI & SVILUPPI FUTURI

CODICE SORGENTE

Introduzione

La Società odierna: una Società tecnologica, sempre connessa, in costante e perpetua comunicazione dove, grazie ad Internet e al progresso informatico, sono state abbattute le barriere della distanza. È proprio in funzione di questa nuova conformazione del mondo in cui viviamo che, nelle ultime due decadi, stiamo assistendo ad una massiccia ed esponenziale produzione di dati. Con un valore del mercato mondiale stimato pari a 274 miliardi di dollari e un volume di memoria di quasi 100 Zettabytes[1], il *mainstream* li ha soprannominati il nuovo "*oro nero"*[2]. Viviamo nell'era dei dati!

È quindi evidente come i mercati finanziari, pionieri dell'approccio *data-driven*, siano stati *in primis* beneficiari di questa metamorfosi della società, indubbiamente guidati dal mondo *FinTech* che si pone sempre all'avanguardia nell'utilizzo delle più recenti tecnologie e nello sfruttamento dei dati da queste generati.

Ma perché vi è questa spasmodica attenzione per i dati? Da dove provengono e chi li genera? Come possono essere utilizzati per ricavare informazioni sulle quali prendere decisioni? La risposta, a mio avviso, ritengo sia semplice: i dati siamo noi. Ognuno di noi, in ogni momento della sua giornata, produce dati. Quando ci spostiamo il GPS del nostro telefono e/o dei nostri *wearables* traccia questi movimenti, quando comunichiamo con messaggi di testo, vocali e chiamate vengono generati dati. Ancora, le nostre attività sui *social networks*, le ricerche che svolgiamo tramite motori di ricerca, le spese effettuate con pagamenti elettronici, gli investimenti finanziari effettuati tramite le nostre banche, su delle piattaforme di trading o tramite un intermediario: altri dati. Qualsiasi cosa noi facciamo generiamo dati. Si potrebbe quasi affermare, esasperando un po' il concetto, che i dati che produciamo descrivono chi siamo. Questo è vero anche per i dati finanziari. Si pensi, banalmente, ai prezzi di un titolo finanziario, un *futures* su di un tasso di cambio per esempio. Quel valore rispecchia quelle che sono le aspettative di tutti gli individui che hanno la possibilità di comprare e vendere quel titolo, poiché non appena una nuova informazione (un comunicato stampa di una Banca Centrale, un dato economico-finanziario di un Paese, …) viene resa pubblica, ogni *traders* si riverserà sul mercato comprando o vendendo quel titolo, in base alle aspettative che ha sul prezzo dello stesso. Effettivamente, la dinamica appena citata altro non è che scelte individuali (dati) che producono una *price-action* (dati) nei mercati

[1] 1 Zettabyte corrisponde a 1 miliardo di Terabytes, il quale è equivalente a 1,000 Gigabytes
[2] In riferimento al petrolio, considerato uno dei mercati più fiorenti e ricchi dell'ultimo secolo

mondiali, alla luce di nuove informazioni (dati) delle quali gli attori del mercato sono venuti in possesso…

Arrivati a questo punto, il lettore avrà sicuramente già colto l'importanza che i Dati hanno ed avranno nel prossimo futuro per i mercati finanziari. Pertanto, praticamente, qual è la figura che ha ed avrà la responsabilità di gestire, (ri)elaborare ed interpretare questa mastodontica quantità di informazioni? Una prima risposta potrebbe essere: il *Data Scientist*, ma sarebbe una visione incompleta, poiché anche gli ingegneri, gli accademici e molte altre figure vengono coinvolte, direttamente e non, all'interno di questo complesso processo.

Il processo della gestione dei dati risulta evidentemente complesso e frastagliato. Inizia dalla generazione di questi, passando per la raccolta, la pulizia, l'elaborazione, l'interpretazione ed infine l'utilizzo finale delle informazioni.

La presente tesi si concentrerà su una della fasi centrali di questo complesso e articolato processo: l'elaborazione. Assumendo di disporre di un gran quantitativo di dati puliti[3], diviene cruciale disporre di modelli che siano in grado di elaborare questi dati al fine di fornire informazioni e/o di prendere decisioni sulla base di questi. In questo preciso contesto si colloca un altro fenomeno in estrema espansione: il *Machine Learning* (ML) e l'Artificial Intelligence (AI). All'interno di questo ambito di studi si colloca il *Reinforcement Learning* (RL), o Apprendimento per Rinforzo, ovvero degli algoritmi che a differenza degli altri modelli, hanno la peculiarità di tenere in considerazione non solo la massimizzazione di una singola scelta ma anche la sequenzialità delle scelte che risultano non indipendenti tra loro. La logica "madre" sulla quale gli algoritmi di RL vengono costruiti è quella del *trial and error*, che in fondo altro non è che il normale metodo di apprendimento che anche noi essere umani utilizziamo (spesso inconsciamente) nella vita di tutti i giorni. Semplificando al massimo il concetto, durante l'allenamento[4] l'algoritmo di RL riceverà un "premio"

[3] Per "puliti" si fa riferimento a quei dati che hanno subito un processo di *data cleaning*. Tale processo, consiste nel controllare (ed eventualmente correggere) la validità, la consistenza e la coerenza dei dati che si vanno ad utilizzare. Per quanto esso sia un argomento poco dibattuto nel mondo accademico, risulta essere uno dei punti chiave nel mondo dei *practitioners*. Il motivo risulta evidente con un esempio: si immagini di costruire una complessa architettura informatica, che impiega avanguardistici algoritmi di analisi per prendere scelte di investimento che coinvolgono ingenti somme di denaro. Qual è la probabilità di ottenere un risultato soddisfacente e profittevole con dei dati (effettivamente il carburante di questa struttura) "corrotti", "sporchi" e inaffidabili?!

[4] Si utilizza il termine "allenamento" o "*train*" per indicare quella fase in cui vengono forniti dei dati all'algoritmico, sulla base dei quali stimerà i propri parametri (o la propria intelligenza, per usare un approccio più "filosofico").

(*reward*) quando i risultati sono buoni e sosterrà un "costo" (*cost*) (ossia una ricompensa negativa) quando questi sono cattivi[5]. Esso sarà quindi incentivato a definire un modello di comportamento (*policy*) in grado di massimizzare le ricompense (presenti e future) e minimizzare i costi, sfruttando (*exploitation*) ed esplorando (*exploration*) i dati con i quali viene nutrito (*feed*).

Nonostante gli ambiti in cui il RL vede delle applicazioni sono molteplici, risulta evidente come il mondo del Trading si presti particolarmente bene all'utilizzo dei sopracitati algoritmi. Un mondo che gode di un'abbondanza di dati accurati, in cui bisogna sempre e costantemente prendere delle decisioni (buy/sell), con la necessità di elaborare (quanto più velocemente possibile!) quantitativi di informazioni (dati) enormi che ci forniscono una descrizione dei mercati finanziari. Estremizzando, potremmo vedere i *traders* come degli algoritmi di RL che elaborano informazioni e prendono decisioni di investimento che vengono ritenute corrette, ricevendo una *reward*, qualora risultino profittevoli oppure sbagliate, ricevendo un *reward* negativa, qualora generino delle perdite. Come apprende un *trader*? Allenandosi nel lavoro che svolge, imparando dai propri errori (*trial and error*), sfruttando (*exploitation*) le tecniche e le conoscenze di cui dispone ma anche sperimentandone (*exploration*) di nuove. D'altronde, questo altro non è che l'approccio seguito dagli algoritmi di RL, quindi perché non usare proprio questi modelli per fare direttamente trading?

Ovviamente, le implicazioni algoritmiche, nonché pratiche, risultano essere enormemente più complesse e problematiche dell'esempio appena illustrato. Saranno proprio queste dinamiche che verranno affrontate dalla presente tesi, la quale è strutturata come segue:

1. Dapprima esplode i concetti chiave che plasmano e caratterizzano il mondo, più generale, del ML e dell'AI, per poi scendere nei dettagli degli algoritmi di RL: quali sono, come funzionano, pregi e difetti. Questa parte verrà ampliamente tratta nel Capitolo 1 "*Framework Teorico*".
2. Verrà poi presentato un ambito applicativo reale. Si esploreranno i dati utilizzati, l'architettura che genererà il nutrimento dell'algoritmo, i modelli di RL che verranno utilizzati, nonché le peculiarità e le criticità che si andranno ad indagare. Le suddette tematiche saranno presentate all'interno del Capitolo 2 "*Definizione del problema & Architettura del Modello*"

[5] John C. Hull, *Machine Learning in Business "Un'introduzione alla Scienza dei Dati"*, Seconda Edizione, 2019.

3. Si analizzeranno i risultati, uscendo da uno squisito ambito teorico per poter apprezzare ciò che effettivamente gli algoritmi di RL possono fare nel mondo del Trading Algoritmico. Tali risultati verranno presentati nel Capitolo 3 "*Risultati*".

CAPITOLO 1

FRAMEWORK TEORICO

1.1 Machine Learning for Algorithmic Trading

Chiunque negli ultimi anni si sia addentrato nel mondo delle analisi quantitative, siano esse finanziarie o no, avrà sicuramente sentito parlare di Machine Learning e di Intelligenza Artificiale. Sebbene si assista spesso ad un abuso di questi termini per finalità di Marketing, è indiscutibile la risonanza e la popolarità che queste aree hanno guadagnato (e continueranno a guadagnare) nel corso degli anni. Il motivo di tale successo, seppur tecnicamente più complesso da spiegare, risulta molto semplice in termini concettuali: sta rendendo possibile l'elaborazione e l'utilizzo di grandi masse di dati, al fine di prendere decisioni più efficienti ed efficaci durante un processo di analisi. Il *Thinking Approach* che sottende questa area di studi, non consiste unicamente nel "costruire un modello per stimare dei parametri", ma bensì nell'implementare vere e proprie architetture di modelli che interagiscono tra di loro al fine di raffinare dati "grezzi", magari provenienti da diverse *sources* (quantitativi puri, dati di testo, trascrizioni di file audio o di immagini, …), condividerli all'interno di un *framework* di modelli e generare degli *outputs* finali (o addirittura eseguire direttamente delle azioni).

Il **Machine Learning** (ML) si occupa dell'utilizzo di grandi quantità di dati per comprendere le relazioni tra variabili, fare previsioni e prendere decisioni in un ambiente che si evolve continuamente. Possiamo inquadrarla come una branca dell'Artificial Intelligence (AI), la quale si occupa dello sviluppo di metodi che consentano alle macchine di imitare (e magari superare) l'intelligenza umana[6].

Il **trading algoritmico** invece, si affida a dei *computer programs* che eseguono algoritmi[7] al fine di automatizzare alcuni o tutti gli elementi di una strategia di trading, quali la

[6] John C. Hull, *Machine Learning in Business "Un'introduzione alla Scienza dei Dati"*, Seconda Edizione, 2019.

[7] Intesi come "sequenze di *steps* o regole disegnate allo scopo di raggiungere o adempiere ad un preciso obiettivo"

generazione di idee di investimento, l'ottimizzazione delle allocazioni di portafoglio, gestione dei rischi, ecc.[8]

Pertanto, inserendo il ML e l'AI all'interno dell'ambito applicato del trading algoritmico, si possono individuare diverse aree di attività:

- Analizzare dati di mercato (e non) al fine di definire delle previsioni (*forecast*) sul futuro andamento di titoli finanziari, sia per uno scopo puramente previsionale e di stima, sia per poter prendere decisioni di investimento (*buy/sell*)
- Gestire le decisioni del punto precedente in un'ottica di portafoglio, massimizzando la gestione del rischio

In gergo, quando si parla di "***generare alpha***", si fa riferimento alla calibrazione di modelli di ML con lo scopo di sfruttare le informazioni contenute nei dati per costruire strategie profittevoli e, soprattutto, sostenibili. Generare alpha risulta essere lo scopo comune di tutti gli algoritmi di trading algoritmico.

1.1.1 ML4T (Machine Learning for Trading)

Il più ampio progetto di un algoritmico di trading, che sfrutta le potenzialità del ML e dell'AI, viene spesso definito **ML4T** (*Machine Learning for Trading*) e si riferisce alla costruzione di un'architettura[9] che mira ad automatizzare l'intero processo di investimento, partendo dalla raccolta dei dati, quindi alla gestione delle API[10] dei *data providers*[11], passando per il *data-cleaning* e per l'utilizzo di modelli di ML e AI, al fine di generare delle decisioni di investimento ed operare in modo algoritmico nei mercati finanziari (in questo caso si parlerà di *execution*). Nonostante la costruzione del ML4T risulti essere il fine ultimo nell'utilizzo degli algoritmi di trading, si presenta come un argomento che va ben al di là dello scopo di questa Tesi, pertanto, d'ora in avanti si andranno ad approfondire

[8] Stefan Jensen, "*Machine Learning for Algorithmic Trading*", Second Edition, 2020

[9] Con la parola "architettura" si fa riferimento alla struttura generale del progetto e a come i suoi vari componenti (hardware, software, reti, database, …) interagiscono tra di loro.

[10] *Application Programming Interface* (API), in questo contesto ci si riferisce ad un *software*, spesso fornito da un *data provider* sotto forma di librerie specifiche per determinati linguaggi di programmazione, che ha lo scopo di permettere la comunicazione (i.e. la trasmissione dei dati) tra due o più *computers*.

[11] Soggetti (i.e. società), tipicamente con modelli B2B, che si fanno carico di raccogliere (e in alcuni casi fare una prima pulizia) dati, in questo caso finanziari, e fornirli in *real-time* o non ai sottoscrittori (i.e. clienti).

fondamentalmente tematiche relative all'utilizzo dei modelli, piuttosto che al loro inserimento all'interno di un *framework end to end*[12]. E' in ogni caso importante tenere a mente che, anche un eccellente algoritmo di investimento, se non supportato da un'adeguata architettura, potrebbe dare risultati non soddisfacenti. La seguente figura mostra, seppur in modo esemplificativo, un *workflow* di ML4T.

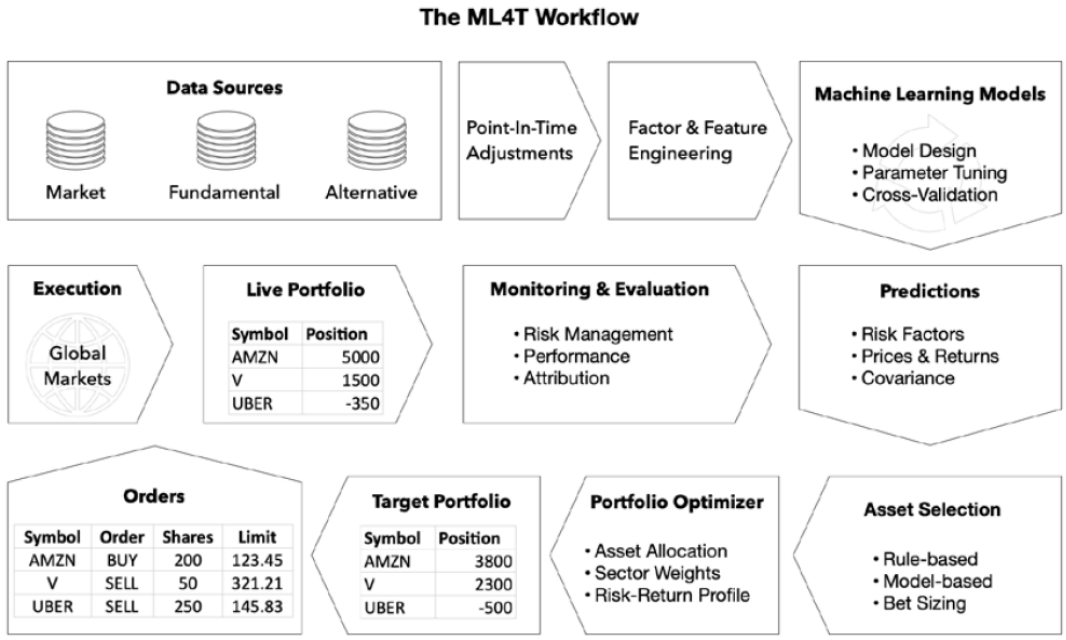


***Figure 1**: ML4T Workflow Example*
Source: Stefan Jensen, "Machine Learning for Algorithmic Trading", Second Edition, 2020

Al fine di fornire al lettore, un'*overview* sui concetti chiavi che sono coinvolti nel più ampio argomento del ML applicato al trading algoritmico, si presenteranno in modo molto sintetico una serie di concetti e terminologie fondamentali che verranno a mano a mano riprese all'interno della Tesi.

1.1.2 Train, Validation & Test datasets

I dati che vengono utilizzati per "calibrare" un modello vengono solitamente divisi in tre *sub-samples*, che in gergo sono appunto definiti *train*, *validation* e *test*. Il *Train-set* rappresenta la porzione maggiore dei dati (solitamente intorno al 70%) e viene utilizzare per

[12] Si utilizza il termine *end to end*, in ambito di ML4T, facendo riferimento ad un'architettura che gestisce l'intero processo di investimento in modo automatizzato, dalla raccolta dei dati fino all'esecuzione degli ordini nei mercati finanziari.

"allenare" il modello. Sono quei dati che il modello "conoscerà" e sulla base dei quali stimerà i propri parametri e definirà la propria conoscenza. Il *Validation-set* è una seconda porzione del dataset originale (solitamente 15%) che ha lo scopo di validare la qualità dei parametri stimati dal modello nel *train-set*, analizzando i risultati che questo produce su dati differenti (i.e. usando i dati del validation, valideremo il modello usando degli *unbiased results*). Ad esempio, si potrebbero confrontare le distribuzioni degli output prodotti dal modello nel *train-set* con quelli prodotti dal *validation-set* e, qualora queste distribuzioni risultassero simili (i.e. il modello è stabile anche al di fuori del *training*), testarlo/usarlo nel *test-set*. A volte nella pratica, la parte di *validation* non viene effettuata, poiché i modelli vengono periodicamente ricalibrati al fine di avere un maggior *fit* con il presente (data la continua mutabilità dei regimi "nascosti" nei mercati finanziari). Pertanto, usare dei parametri stimati nel *train-set* solo dopo averli validati nel *validation-set*, rischierebbe di usare *live*[13] dei modelli con parametrizzazioni obsolete rispetto al regime corrente. Infine, vi è il *Test-set*. Questo si presenta come quella porzione del dataset originale (solitamente 20-30%) sul quale viene effettivamente testata la qualità e la stabilità dei parametri di un modello. Per richiamare una terminologia econometrica, il *train-set* corrisponderebbe all'*in-sample* mentre il *test-set* all'*out-of-sample*.

1.1.3 Overfitting & Underfitting

Si avrà *Overfitting* quando si otterranno ottimi risultati sul *training-set* ma scarsi risultati sul *test-set*, mentre si avrà *Underfitting* quando si otterranno scarsi risultati sia sul *training-set* che sul *test-set*. Mentre per l'*underfitting* le ragioni sono più ovvie, banalmente il modello è inadatto allo scopo, l'*overfitting* risulta essere una problematica in cui si può più facilmente incorrere durante la calibrazione di un modello. La ragione principale per cui si può avere un *overfitting* del modello è quella di utilizzare un modello troppo complesso, che andrà ad approssimare eccessivamente bene sui dati del *training-set*, perdendo però di generalizzazione, ovvero quella capacità per cui i parametri stimati da un modello sono in grado di generare buoni risultati anche al variare del dataset, avendo "imparato" adeguatamente "le regole" sottostanti ai dati. La seguente figura illustra un esempio nel

[13] In gergo, con il termine "*live*" si fa riferimento a quei modelli (o componenti del *software*) che vengono effettivamente utilizzati da una società che fa trading algoritmico, ovvero quelle componenti che effettivamente veicolano risorse finanziarie.

quale si costruiscono tre modelli di stima di una funzione "ignota", partendo da un solo grado di libertà e, a mano a mano, "complicando" il modello.

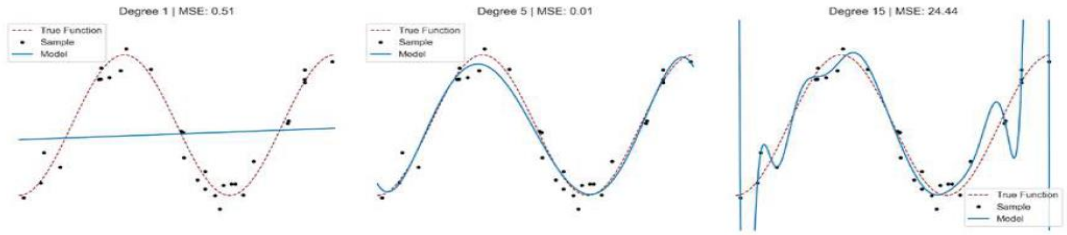


Figure 2: A visual example of *overfitting* with polynomials
Source: Stefan Jensen, "Machine Learning for Algorithmic Trading", Second Edition, 2020

Risulta evidente come utilizzando un solo grado di libertà, il modello stimato sia inadeguato (i.e. *underfitting*) ad approssimare la funzione che genera i dati, così come un modello che utilizza 15 gradi di libertà generi dei risultati pressoché perfetti per i dati sul quale viene allenato (*training-set*) andando però a perdere di generalizzazione, ovvero travisando quella che è la funzione dalla quale vengono generati i dati. In conclusione, nell'esempio della figura, un modello con 5 gradi di libertà si presta ad essere la soluzione migliore a gestire il *trade-off* tra precisione della stima e generalizzazione della *true function*.

1.1.4 Look-Ahead bias

Un altro errore in cui si può incorrere nella costruzione di un modello di ML per il trading algoritmico è quello del Look-Ahead bias. Questo fenomeno si manifesta quando il modello viene allenato fornendogli dei dati in un ordine temporale differente da quello che effettivamente si sarebbe verificato in un'applicazione *live*. Così facendo, si andrà ad allenare un modello "illudendolo" di poter disporre di informazioni prima che queste siano effettivamente disponibili nella realtà. Per fornire un esempio pratico, si pensi ai bilanci trimestrali nell'*equity* (i.e. *stock market*). Volendo allenare un modello di ML, nutrendolo con i dati trimestrali delle società, si potrebbe erroneamente pensare di renderli disponibili al termine di ogni trimestre. Così facendo si cadrebbe in un gigantesco *Look-Ahead bias* perchè, nonostante sia vero che le trimestrali fanno riferimento al trimestre effettivamente chiuso, non è altrettanto vero che in un contesto *live* si disporrebbe di questi dati esattamente alla chiusura del trimestre. Nella realtà infatti, questi dati vengono resi pubblici settimane dopo la chiusura del trimestre, generando *price-move* proprio in quell'istante. Pertanto, allenare un modello di ML fornendogli le trimestrali al termine esatto del trimestre,

creerebbe un'assunzione tale per cui l'algoritmo conosce in anticipo dei dati non ancora pubblici, rendendo i parametri stimati inutilizzabili in un contesto *live*.

1.1.5 Features Engineering

Quando si parla di Features Engineering (FE), si fa riferimento alla costruzione e/o all'estrazione di "caratteristiche" intrinseche nei dati, che hanno lo scopo di esaltare le peculiarità del dominio. Nel caso specifico del trading algoritmico, esempi di FE potrebbero essere:

- *Indicatori di analisi tecnica*: Bande di Bollingers, Relative Strength Index (RSI), Punti Pivots, …
- *Indicatori statistici stimati con finestre rolling*: Medie e Volatilità (SMA o EWMA), GARCH, …
- *Dummy variables*: giorni della settimana, orari, sessioni di trading (US, EU, …), sigma events, …
- …

Il mondo delle FE è di certo un ambito del trading algoritmico in cui vi è la maggiore possibilità di spaziare con la creatività, poiché, lo scopo ultimo di queste è quello di esaltare una caratteristica dei dati, al fine di rendere più facile per l'algoritmo l'individuazione di determinate regole intrinseche nei dati. È importante però fare un appunto sulle FE, soprattutto in merito alla domanda "qual è il numero di Features sufficienti per istruire un modello?". Non vi è una risposta esatta alla domanda, ma la sola esistenza di diversi modelli di c.d. *Features Selection*, ovvero di selezione di un *sub-samples* di features, dovrebbe suggerire al lettore che anche nell'ambito delle FE vale il detto "*il troppo storpia*". Si può tranquillamente affermare che l'obiettivo è: fornire all'algoritmo delle *features* che evidenzino caratteristiche dei dati che non sono correlate tra loro (banalmente, è inutile fornire una volatilità con finestra 5 giorni e un'altra con finestra 6 giorni, in quanto daranno sostanzialmente la stessa informazione). Apprezzando un concetto che nell'Econometria è ormai noto da decenni, si potrebbe affermare che la *ratio* sottostante la creazione di un *pool* di features adeguato, dovrebbe essere il medesimo utilizzato nella ricerca di un *pool* di variabili indipendenti che non presentano multi-collinearità tra di loro.

1.1.6 Hyper-parameters Tuning

Un ultimo concetto utile è sicuramente quello dell'*Hyper-parameters Tuning*. Lo si potrebbe definire come "la fase finale" della calibrazione di un modello. Questo avviene quando bisogna scegliere alcuni parametri del modello che, almeno teoricamente, non si potrebbero stimare direttamente dai dati. Si pensi, ad esempio, al parametro di regolarizzazione all'interno di una Lasso Regression, oppure alla "soglia" di accettazione per l'output di una Logit. Per ovviare a queste scelte, nel mondo del ML viene utilizzata una tecnica chiamata *Cross-Validation*. Sostanzialmente, si vanno a valutare delle metriche che suggeriscono la qualità di un modello in fase di test, variando questi parametri. Banalmente, la parametrizzazione che permetterà al modello di performare al meglio, verrà selezionata. Per fare un esempio, nel caso di un modello di *Supervised-ML*, una metrica che viene spesso impiegata è quella dell'*Accuracy*[14]. Esistono molte tecniche per fare cross-validation, la cui spiegazione va al di là dello scopo della presente Tesi. Si vuole però sottolineare che, ogni tecnica di cross-validation che impiega il c.d. *shuffle*, ovvero il ri-ordino casuale dei dati per la costruzione del train e del test set, viene spesso sconsigliata, in quanto l'ordinalità temporale dei dati è una caratteristica molto importante per i dati dei mercati finanziari, a causa di fenomeni di auto-correlazione che spesso questi dati presentano.

1.1.7 Tecniche di Machine Learning

Volendo infine presentare una breve classificazione delle varie tecniche di ML, fatta eccezione per il RL che sarà argomento del prossimo paragrafo, queste sono raggruppabili nei seguenti gruppi:

- **Supervised Machine Learning**

  È una tecnica di apprendimento automatico in cui i dati utilizzati per l'addestramento dell'algoritmo sono già etichettati (*labels*) e l'obiettivo è quello di prevedere i *labels* usando nuovi dati non visti in precedenza (*test-set*). In altre parole, l'algoritmo viene "guidato" dai *labels* forniti nel *train-set* per fare previsioni su dati sconosciuti (*test-set*).

[14] Questa è la capacità di un modello di fare previsioni corrette. Viene calcolata come numero di previsioni corrette fatte dal modello diviso il numero totale delle previsioni che sono state fatte. Un modello perfetto avrà un'accuracy pari ad 1 (i.e. 100%).

Ci sono due tipi principali di problemi di Machine Learning supervisionato: il problema della *regressione* e il problema della *classificazione*. Nel problema della regressione, l'obiettivo è quello di prevedere una variabile continua (ad esempio il rendimento di un titolo azionario). Nel problema della classificazione, invece, l'obiettivo è quello di prevedere una variabile categoriale (ad esempio, prevedere se il successivo prezzo di un titolo sarà in rialzo o in ribasso).
Esempi tipici dell'utilizzo del Supervised ML in ambito di trading algoritmico, sono:

- *Probabilità di successo di un trade*, allenando ad esempio un modello Logit con delle FE costruite su caratteristiche del mercato e del titolo nel momento in cui un *trade* viene aperto.

- *Parametrizzazione delle FE e/o dei modelli*. Per esempio, allenando un modello regressivo OLS o Lasso, sia con serie storiche delle FE di mercato e del titolo, utilizzando come variabile endogena (*labels*) i P&L[15] che si sarebbero ottenuti in base a diverse parametrizzazioni delle FE e/o dei modelli. Si otterrebbero così dei coefficienti beta tali per cui, in fase di calibrazioni dei modelli o delle FE, si potrebbero inserire le variabili esogene date (i.e. FE di mercato e del titolo) e far variare le esogene relative alle parametrizzazioni che possiamo controllare, andando ovviamente a ricercare quella parametrizzazione che massimizza l'*outcome* (i.e. il *label*). In gergo, queste tipologie di modelli si annoverano sotto il nome di "*Hidden Regime Models*"

- …

- **Unsupervised Machine Learning**

  In questo caso, i dati non sono etichettati e l'obiettivo è quello di trovare relazioni o pattern nascosti nei dati. Ad esempio, si potrebbe utilizzare il Machine Learning non supervisionato per raggruppare persone in base a determinate caratteristiche e vedere se esistono differenze significative tra i gruppi. In altre parole, l'algoritmo deve scoprire da solo quali sono le caratteristiche importanti nei dati e come utilizzarle per fare previsioni o raggruppare gli elementi in base alle loro somiglianze.

[15] Profit & Loss

Il Machine Learning non supervisionato è utile quando non si dispone di etichette per i dati o quando si vuole scoprire pattern nascosti nei dati che potrebbero non essere evidenti a prima vista. Tuttavia, ha anche alcune limitazioni, poiché l'algoritmo non ha una "guida" esterna per fare le previsioni e può essere difficile interpretare i risultati ottenuti.

Alcuni esempi di Unsupervised ML applicato al trading algoritmico sono:

- *Clustering*: è una tecnica di Machine Learning non supervisionato che mira a raggruppare gli elementi dei dati in base alle loro somiglianze. In un contesto di trading algoritmico, il clustering potrebbe essere utilizzato per raggruppare i titoli in base alle loro caratteristiche (ad esempio, settore, capitalizzazione di mercato, volatilità, etc.) e fare previsioni sulla performance futura dei titoli all'interno di ogni cluster.
- *Principal Component Analysis* (PCA): la PCA è una tecnica di *dimensionality reduction* che mira a ridurre il numero di caratteristiche presenti nei dati, mantenendo le informazioni più importanti. In un contesto di trading algoritmico, la PCA potrebbe essere utilizzata per analizzare i dati storici dei prezzi dei titoli e individuare le componenti principali che influiscono sulla performance del mercato. Ad esempio, se si applica un modello di PCA su un basket di tassi di cambio valutari (a.k.a. *Forex Market*), si noterà che la prima componente, solitamente, è rappresentata dall'influenza del dollaro. Si potrebbe quindi voler implementare una strategia di trading che è non correlata con la componente principale del mercato (i.e. fattore di rischio comune a tutti, il mercato), magari facendo un'ottimizzazione di portafoglio che ne minimizza l'esposizione.
- …

- **Semi-Supervised Machine Learning**

E' una tecnica di ML che si trova a metà strada tra il *Supervised ML* e l'*Unsupervised ML*. In questo tipo di apprendimento, solo alcuni dei dati sono etichettati e l'obiettivo è quello di utilizzare queste etichette per etichettare anche gli altri dati.

Il Machine Learning semi-supervisionato è utile in situazioni in cui si dispone di molti dati non etichettati, ma si ha anche a disposizione un piccolo numero di dati etichettati che possono essere utilizzati come riferimento.

Alcuni esempi applicativi per l'Algorithmic Trading potrebbero essere:

- *Apprendimento con etichette imprecise*: in un contesto di trading algoritmico, l'apprendimento con etichette imprecise potrebbe essere utilizzato per etichettare automaticamente i dati storici dei prezzi dei titoli come "tendenza al rialzo" o "tendenza al ribasso". Ad esempio, si potrebbe utilizzare un piccolo numero di dati etichettati manualmente come riferimento per addestrare un modello che poi può essere utilizzato per etichettare automaticamente il resto dei dati. Queste a sua volta potrebbero essere delle FE (1: potenziale rialzo, 0: potenziale ribasso) da poter fornire ad un altro algoritmo, come ad esempio un algoritmico di RL.
- …

- **Deep Learning**

Il Deep Learning è una tecnica di apprendimento automatico basata sulla creazione di modelli di rete neurali profonde, cioè modelli di rete neurali composti da molti strati di neuroni artificiali. Le reti neurali sono modelli ispirati al funzionamento del cervello umano, che sono in grado di "imparare" dai dati utilizzati per addestrarle.

Le reti neurali profonde sono in grado di "imparare" rappresentazioni complesse dei dati, che possono poi essere utilizzate per fare previsioni o classificare nuovi dati.

Il Deep Learning richiede solitamente una quantità significativa di dati per addestrare i modelli, poiché i modelli di rete neurale profonda sono in grado di "imparare" rappresentazioni complesse dei dati. Inoltre, il Deep Learning richiede spesso l'utilizzo di potenti computer per addestrare i modelli, poiché i modelli sono spesso molto complessi e richiedono tempo per essere addestrati.

Un esempio di utilizzo al trading algoritmico, che tra l'altro, si adatterebbe molto bene per determinati modelli di RL, è il GAN:

- Generare dati di addestramento artificiali che possono essere utilizzati per addestrare modelli di ML per il trading algoritmico, come il RL.
- Migliorare la qualità dei dati per addestrare i modelli. Ad esempio, il GAN potrebbe essere utilizzato per eliminare il rumore (*noise*) o gli errori

- **Artificial Intelligence (AI)**

Alcuni esempi di AI applicate al Trading algoritmico possono essere:

- Reti Neurali

- Intelligenza artificiale evolutiva (EAI). È una tecnica di ML che utilizza gli algoritmi di ottimizzazione evolutiva per addestrare i modelli. In un contesto di trading algoritmico, l'EAI potrebbe essere utilizzata per ottimizzare i parametri di un modello di trading algoritmico, ad esempio per individuare le combinazioni di parametri che portano a prestazioni ottimali. Ad esempio, gli algoritmi genetici si prestano perfettamente a svolgere il compito sopra-descritto, beneficiando di un minor rischio di "incastrarsi" in un ottimo locale durante una ricerca dello stesso, nonché fornendo spesso prestazioni più efficienti di altri algoritmi più semplici.

1.2 Reinforcement Learning: un'introduzione

Il Reinforcement Learning (Apprendimento per Rinforzo) è una tecnica di apprendimento automatico (ML) che mira a far imparare ad un agente (*agent)* come interagire con un ambiente (environment) in modo da massimizzare una ricompensa (*reward*). In altre parole, l'obiettivo del RL è far sì che l'agente impari a compiere azioni che portino a risultati positivi a lungo termine. L'agente riceve una *reward* ogni volta che compie un'azione che porta a un risultato positivo, e una punizione (*reward* negativa) ogni volta che compie un'azione che porta a un risultato negativo. Nel mentre, utilizza queste ricompense e punizioni per modificare il proprio comportamento (*policy*) e imparare a compiere le azioni più efficaci per ottenere la più alta *reward*.

Volendo dare una definizione più concettuale di cosa è e di come lavora il *Reinforcment Learning* (RL), piuttosto che una definizione squisitamente tecnica, allora si può pensare al RL come ad un soggetto, come ad esempio un essere umano, che apprende dall'interazione costante e ripetuta con un ambiente, sia capitalizzando "conoscenza" dalle scelte "giuste" che "imparando dai propri errori" (meccanismo *trial and error*).

Effettivamente, astraendo il concetto si potrebbe pensare ad un infante che impara a camminare. Questo, altro non fa che fare dei tentativi ripetuti (*trial*), inizialmente cadendo a terra nel più dei casi (*error*), fintanto che la sua abilità motoria non apprende a tal punto (*Reinforcement*) da smettere di cadere e riuscire a camminare. Ad ogni tentativo, che si potrebbe definire come "un'interazione con l'ambiente", sta semplicemente cercando di apprendere delle distribuzioni di probabilità che legano ad una sua scelta (*action*), presa in un contesto specifico (*state*) un successo (cammina) o in un insuccesso (cade), ovvero una

reward. Cercherà quindi di calibrare le variabili che sono sotto il suo controllo (equilibrio, bilanciamento del corpo, …) al fine di massimizzare l'output di una funzione che viene definita in base al contesto specifico in cui si trova, in sostanza, una massimizzazione vincolata. Ovviamente, il contesto (le variabili che descrivono l'ambiente) andrà a caratterizzare quale scelta massimizzerà tale funzione, definirà quindi uno stato. Basti pensare a quanto il camminare sia diverso se ci si trova in discesa o in salita, cambia dove deve essere indirizzato il peso del corpo (salita a monte, discesa a valle); ancora se la superfice è scivolosa o aderente (scivolosa passi corti, aderente passi più lunghi); e così via.

Un altro esempio potrebbe essere una qualsiasi persona che, ritrovandosi in ritardo per andare al lavoro, cercherà di ottimizzare il tempo che impiega per prepararsi e fare colazione.

> *"Ognuno di noi, se ci si pensa, dispone di un proprio train-set costruito durante l'arco della propria vita. Un set di dati che inconsciamente abbiamo memorizzato e su cui abbiamo allenato la nostra mente per stimare i "nostri parametri". La vita che abbiamo vissuto fino ad oggi è il nostro train-set, il futuro è il nostro test-set e il nostro comportamento nel presente altro non è che l'utilizzo dei parametri che abbiamo stimato sino ad oggi. Ognuno di noi, in ogni momento, fa un costante online-learning, aggiornando continuamente i propri parametri."*

Sulla base di quanto detto, riprendendo l'esempio dell'ottimizzazione del tempo di preparazione per andare al lavoro, quella mattina in cui un individuo è in ritardo, come affronterebbe il problema? (un problema? Quasi come se fosse un "gioco" da risolvere dove si gioca contro il tempo…) Beh, se quanto detto poc'anzi è vero, l'individuo saprebbe esattamente quali sono le azioni che, attuate nello specifico stato in cui si trova, gli permetterebbero di ridurre al minimo il tempo impiegato nel prepararsi; quindi, tenderà a sfruttare (*exploitation*) le informazioni di cui dispone, ovvero sfrutterà la propria esperienza. Effettivamente, egli ha inconsciamente costruito delle distribuzioni di probabilità… Data la vita vissuta sino a quell'istante, che rappresenta il suo *train-set*, sa per esempio che, fare prima colazione e dopo farsi la doccia e vestirsi, gli farà guadagnare tempo (massimizzerà il proprio *output*). Ancora, sa che in base al giorno della settimana, c'è una strada che è più veloce di un'altra. Però, di tanto in tanto, proverà a fare delle scelte differenti, mai fatte: dei tentativi, delle esplorazioni (*exploration*). Ad esempio, proverà ad invertire l'ordine doccia/colazione, oppure percorrerà una strada nuova (nei limiti delle scelte di cui dispone, ovvero dello spazio definito dalle variabili che sono sotto il suo controllo), etc. Al termine di questa esplorazione, a prescindere dal risultato che otterrà, egli avrà guadagnato

informazioni sulla *reward* ottenibile attuando una determinata scelta in un dato stato. Quello appena descritto, nel RL viene definito come "il dilemma *Exploration* vs *Exploitation"*: è meglio sfruttare ciò che si conosce o esplorare nuove possibilità? Sebbene l'argomento sarà discusso in modo più approfondito in seguito, si può anticipare sin da subito che è stato empiricamente dimostrato che inserire una parte di "esplorazione" nel RL, implementa notevolmente la velocità di apprendimento dell'algoritmo ed evita che questo si "incastri" in una conoscenza parziale, non potendo mai esplorare *in todo* lo spazio di azioni di cui dispone.

In definitiva, il RL "impara" a mappare delle situazioni in base a delle azioni, cercando di massimizzare una funzione quantitativa di *reward*. L'algoritmo, quindi, non saprà mai qual è l'azione perfetta, bensì suggerirà quale azione/i permette di ottenere la *reward* più alta in un dato contesto, semplicemente provandola durante il suo "allenamento". In alcuni casi, un'azione intrapresa nel presente influenzerà anche le *rewards* future e gli stati in cui l'algoritmo si troverà a prendere altre decisioni. Si creerà quindi una sequenzialità decisionale che dovrà essere tenuta in considerazione. Pertanto, apprendimento tramite *trial-and-error* e ricompense dilazionate nel tempo, sono due tematiche cruciali nel RL.[16]

Ovviamente, il RL è notevolmente distante dagli approcci utilizzati nel *Supervised ML*. Questi ultimi, utilizzano un *training-set* di *labeled examples* costruiti esternamente, dove ognuno di questi descrive un output dato uno specifico contesto, i.e. uno stato. Basti pensare ad una regressione lineare: la variabile dipendente è la *reward* ottenibile, dato uno stato (i.e. le variabili indipendenti). Nonostante tali modelli siano ampiamente utilizzati, sia per la praticità di questi che per la loro comprensibilità, non si presentano adeguati per apprendere da un'interazione con un'ambiente, essendo spesso impossibile ottenere dei *labels* da questo. Si potrebbe allora pensare che il RL possa appartenere all'universo degli algoritmi di *Unsupervised ML*, poiché questi non si affidano ad un output esterno (i *labels*) che gli "suggerisce" qual è il corretto comportamento da adottare in un dato stato, ma bensì cercano di individuare una sorta di "funzione nascosta" che sia adeguata a descrivere il problema. Sicuramente modelli di Unsupervised ML possono essere utili per "potenziare" (*boosting*) un algoritmo di RL, si può pensare ad esempio alle reti neurali utilizzate nel Q-learning, ma *stand-alone* non risultano comunque in grado di risolvere dei problemi di RL.

[16] S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd, 2020

Si deve quindi pensare al RL come ad un terzo paradigma all'interno del ML. Questo, si costruisce considerando esplicitamente l'intero problema come "un unico blocco da risolvere", diventando quindi un *goal-directed agent* che interagisce con un ambiente caratterizzato da incertezza.[17] Bisogna però tenere a mente che, come suggerito da S.Sutton, il RL non deve necessariamente essere visto come "un'entità completa" in grado di svolgere tutte le attività, bensì, si può pensare ad un algoritmo di RL anche come una componente di un sistema più ampio, un braccio di un robot ad esempio. Pensando ad una situazione di trading algoritmico, non è detto che ci sia soltanto un algoritmo di RL che "guida" tutte le decisioni di investimento. Potrebbero tranquillamente esserci molti algoritmi di RL che interagiscono tra di loro per definire la migliore scelta di investimento, come fossero un *team* di *traders*. Per fare qualche esempio, un algoritmo potrebbe occuparsi unicamente dell'allocazione di portafoglio ed ottimizzare determinate metriche (Sharpe Ratio, Gain Loss, Volatilità, …); un altro algoritmo potrebbe occuparsi di minimizzare la perdita inserendo dei *trailing stop-loss* "intelligenti"; un altro ancora potrebbe indicare al primo quali sono i titoli "caldi" che vale la pena inserire in portafoglio; e così via. Un altro esempio in cui il RL potrebbe interagire in modo più "esterno", potrebbe essere quello di definire un problema di Multi-Armed Bandits (uno dei problemi più basilari nel mondo del RL) dove ogni leva è una strategia di trading e l'agente deve periodicamente selezionare la strategia che, data la sua esperienza, profitterà di più in un dato regime economico-finanziario.

Alcuni esempi molto popolari, seppure non in ambito economico-finanziario, dell'utilizzo del Reinforcement Learning vedono algoritmi applicati al gioco degli Scacchi o del Go, che sono stati in grado di battere i campioni mondiali delle relative discipline. Sono molto spesso utilizzati in ambito robotico, pensiamo banalmente all'aspirapolvere che pulisce in autonomia, ricaricandosi quando è scarica. L'intelligenza relativa alla ricarica è, nel più dei casi, un algoritmo di RL che impara a capire quando deve ricaricarsi e quando può continuare a pulire, ottenendo una reward positiva (+1) quando pulisce senza spegnersi e una reward negativa (-1) quando si spegne prima di essere riuscito a tornare alla base per ricaricarsi. Cercando di massimizzare questa reward, l'algoritmo imparerà come comportarsi in base agli stati (distanza dalla base di ricarica, batteria residua, etc.).

[17] S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd, 2020

Alla luce delle considerazioni, perlopiù astratte e concettuali, nonché degli esempi presentati sinora, il lettore avrà già sicuramente apprezzato la potenzialità di questa famiglia di algoritmi, sia in ambito economico-finanziario sia in ambiti completamente diversi.

Si dedicheranno quindi i successivi sotto capitoli alla formalizzazione tecnica e matematica di un problema di RL, nonché si esploreranno le caratteristiche fondamentali di questi, presentando alcune tipologie di algoritmi che, nello specifico, si prestano eccellentemente alla risoluzione di problemi contestualizzati in un ambito di trading algoritmico.

D'ora in avanti, la notazione che verrà utilizzata all'interno della presente tesi corrisponderà a quella utilizzata nella seconda edizione di "*Reinforcement Learning: an Introduction*" (Sutton e Barto, 2020).

1.3 Concetti chiave nel Reinforcment Learning

1.3.1 Struttura del Problema

Come già anticipato nei paragrafi precedenti, all'interno di un problema di RL vi è un agente (*agent*) che interagisce con un ambiente (*environment*). L'agente agisce in base alla propria *policy*, ovvero una strategia che sceglie le azioni da intraprendere in ogni stato (*state*). L'*environment*, a sua volta, fornisce all'agente una ricompensa(*reward*) in base all'azione (*action*) scelta e ai cambiamenti di stato che ne conseguono. In sostanza, un agente di Reinforcement Learning impara a prendere decisioni, all'interno di un *environment* ad esso sconosciuto, compiendo una serie di azioni e ottenendo delle *rewards* (quantitative) associate alla "qualità" dei risultati a cui queste azioni hanno portato. Accumulando esperienza tramite un processo di *trial-and-error*, l'agente imparerà quali sono le azioni migliori da compiere a seconda dello stato in cui si trova, il quale verrà definito dalle caratteristiche dell'ambiente in un dato stato e dalle azioni che l'agente ha preso in precedenza. Prendendo come esempio il Tic-Tac-Toe (a.k.a. *tris, filetto, …*), uno stato potrebbe essere la condizione attuale di gioco (il turno in cui ci si trova, condizione della partita, precedente mossa dell'avversario, …), le possibili azioni di cui l'agente dispone sarebbero le celle in cui può mettere il cerchio o la croce, le *rewards* sarebbero +1 per ogni partita vinta e -1 per ogni sconfitta e, ovviamente, le mosse compiute dall'agente nei turni precedenti (escludendo la situazione in cui si è nel primo turno) influenzeranno le scelte che questo potrà/dovrà prendere nel turno attuale. In questo problema di RL, l'agente giocherà

un numero N di partite (*trial-and-error*), potenzialmente molto grande, cercando di massimizzare la *reward* che ottiene ad ogni partita. Questo sarà possibile mediante il continuo "aggiornamento" della propria *policy*, ovvero della strategia che l'agente segue durante una partita di Tic-Tac-Toe.

Pertanto, un modello di *Reinforcement Learning* è formato dai seguenti elementi:

- Un **set di stati** $(s_0, s_1, \dots, s_n) \in \mathcal{S}$, definito dall'interazione tra l'*environment* e l'agente
- Un **set di possibili azioni** $(a_0, a_1, \dots, a_n) \in \mathcal{A}$, che verranno selezionate dall'agente in base allo stato in cui si trova e alla *reward* che si aspetta di ottenere selezionando un'azione in un dato stato (si parlerà di *action-value function*)
- Una ***reward*** $R \in \mathbb{R}$, che l'ambiente "darà" all'agente ogni qual volta che questi due avranno un'interazione. Ovvero, scegliere l'azione a_t in s_t darà all'agente una ricompensa R_{t+1} in $t+1$.
- Una ***policy***, che verrà iterativamente sviluppata dall'algoritmo di RL, con lo scopo di mappare ogni stato s_t in input a un'azione a_t in output; ovvero una strategia da seguire, tale per cui, dato uno stato l'algoritmo selezionerà un'azione che massimizza la *reward* ottenibile (presente e futura). Più formalmente, potremmo definire la *policy* come un *mapping* degli stati con le probabilità di selezionare ogni possibile azione. Pertanto, se l'agente segue una *policy* π al tempo t, si avrà che $\pi(a|s)$ è la probabilità che l'algoritmo selezionerà l'azione $a \in A_t$ nello stato $s \in S_t$.[18]
- Una serie di funzioni chiamate ***state-value-function*** (denotata con $v(s)$) e ***action-value-function*** (denotata con $q(s,a)$) che determinano, rispettivamente:
    - Il valore atteso della ricompensa totale ottenuta dall'agente se si trova in un dato stato
    - Il valore atteso della ricompensa totale ottenibile compiendo un'azione, dato che l'agente si trova in un particolare stato. Ovvero, l'*Expected Reward* associata ad un'azione in uno particolare stato

Un agente di RL interagisce con l'*environment* in un certo istante di tempo t. Ad ogni t l'agente riceve in input uno stato $s_t \in \mathcal{S}$ e un *reward* $R_t \in \mathbb{R}$ (la *reward* R_t sarà la ricompensa, riconosciuta dall'ambiente all'agente, data dall'interazione che vi è stata in $t-$

[18] S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd, 2020

1). In base a questi input, l'agente determina l'azione $a_t \in \mathcal{A}(s_t)$ da compiere, dove $\mathcal{A}(s_t)$ rappresenta l'insieme di tutte le possibili azioni in un dato stato s_t. Quest'ultima viene ricevuta dall'*environment* che la "processa" ed elabora un nuovo stato s_{t+1} e un nuovo segnale di reward r_{t+1}, corrispondente al successivo input dell'agente nel periodo di tempo $t + 1$.

Questo processo, considerato in modo ricorsivo, genera l'algoritmo di apprendimento dell'agente di Reinforcement Learning. L'obiettivo dell'agente è guadagnare il più possibile in termini di reward cumulativo finale. Lo scopo può essere raggiunto utilizzando diverse metodologie. L'agente, durante l'allenamento, è in grado di imparare opportune strategie che gli permettono di ottenere un maggior guadagno immediato oppure avere un guadagno maggiore a lungo termine a discapito dei reward immediati.

1.3.2 Markov Decision Process (MDP)[19]

MDP è una classica formalizzazione di un processo decisionale sequenziale, dove le azioni non influenzano soltanto le *rewards* immediate ma anche le successive situazioni (i.e. stati) attraverso le *rewards* future. Pertanto, un MDP utilizza un approccio di *rewards* dilazionate nel tempo e risulta quindi essere un'ottima formalizzazione matematica per descrivere problemi di RL, permettendo di dare delle contestualizzazioni formali e teoriche ben precise.

In un contesto che segue le regole di un MDP, si avrà un *agent* e un *environment* (tutto ciò che è esterno all'agente). Questi interagiranno costantemente: l'agente selezionerà delle azioni e l'ambiente risponderà a tali azioni generando nuove condizioni (i.e. nuovi stati) e remunerando l'agente con delle rewards numeriche che l'agente cercherà di massimizzare nel corso del tempo, selezionando le azioni che ritiene "migliori". La seguente figura mostra uno schema del funzionamento di un MDP:

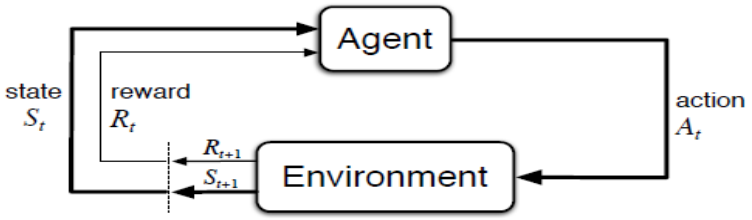


Figure 3: Agent-Environment interaction in a Markov Decision Process
Source: Sutton and Burton, "*Reinforcement Learning: an Introduction*", Second Edition, 2020

[19] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

In particolare, l'agente e l'ambiente interagiranno ad ogni time step t di una sequenza di tempi discreti $t = 0, 1, 2, ...$ Ad ognuna di queste interazioni, l'agente riceverà una rappresentazione dello stato $s_t \in \mathcal{S}$ in cui l'ambiente si trova e selezionerà un'azione $a_t \in \mathcal{A}(s_t)$. Il *time-step* successivo, in parte come conseguenza dell'azione selezionata dall'agente, l'agente riceverà una ricompensa numerica $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, trovandosi in un nuovo stato s_{t+1}. Si andrà pertanto a generare una sequenza di traiettorie del tipo:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

In particolare, in un *finite-MDP* l'insieme degli stati, delle azioni e delle rewards hanno tutte un numero finito di elementi. In questo caso, le variabili aleatore R_t e S_t avranno una distribuzione di probabilità discreta e definita che dipende dall'azione selezionata nello stato precedente. Semplificando, se lo spazio degli stati, delle azioni e della ricompensa è finito, allora selezionare se in *t*, dato uno stato si sceglie un'azione, vi è una certa probabilità (i.e. una distribuzione di probabilità) di ritrovarsi in uno stato $s' \in \mathcal{S}$ in $t + 1$. Formalizzando:

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \qquad (1.1)$$

p è una funzione deterministica ordinaria di 4 argomenti, definita come segue:

$$p: \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

Pertanto la funzione p caratterizzerà completamente la dinamica di un processo del tipo MDP . Essendo p una funzione che definisce una distribuzione di probabilità per ogni scelta di stati e azioni in un dato tempo t, allora vale la proprietà per cui:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \qquad for\ all\ s' \in \mathcal{S}, a \in \mathcal{A}(s)$$

In conclusione, la probabilità di ogni stato e azione, dipenderanno dallo stato e dall'azione immediatamente precedente, e così via in modo ricorsivo. Un dato stato S_t sarà quindi il risultato dell'intera interazione passata tra agente e ambiente. Se questa proprietà viene rispettata, si parlerà di **Markov Property.**

Partendo dalla *four-arguments function p* nella (1.1), si possono derivare diverse funzioni che descrivono l'ambiente, come ad esempio la "probabilità di transizione di stato" (***state transition probabilities***):

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \textstyle\sum_{r \in \mathcal{R}} p(s', r \mid s, a) \qquad (1.2)$$

Essa rappresenterà la probabilità di passare in un dato stato s data un'azione a intrapresa nello stato precedente, infatti "probabilità di transizione di stato". In questo caso, p sarà una *three-arguments function*, definita come segue:

$$p: \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

Si può anche definire il Valore Atteso delle reward in uno stato successivo s' date tutte le possibili coppie di stato, azione precedenti.

$$r(s, a) \doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \,|s, a) \qquad (1.3)$$

Ovvero una media della *rewards* ottenibili in t, dove le ricompense sono ponderate per la probabilità di ottenere una data ricompensa (i.e. ritrovarsi in uno stato s') data l'azione intrapresa nello stato precedente. E' un valore atteso di variabile aleatoria, dato che la seconda sommatoria (quella su s) somma a 1.

Passando da un ambito di definizioni formale e rigoroso, per tornare ad un approccio più pragmatico, è importante ricordare che la definizione dell'ambiente (tutto ciò che l'agente è in grado di modellizzare per trovare la sua *policy* ottimale), delle azioni e della funzione di *reward* sono tutte variabili che verranno fornite dall'utente all'algoritmo. Pertanto, al lettore sarà evidente sin da subito quanto possa essere cruciale questa parte in fase di definizione di un problema di RL.

Nonostante il *framework* dei *finite-MDP* sia ottimale per la definizione di un problema di RL, è importante ricordare che non sempre è possibile definire una probabilità di transizione di stato e il mondo dei mercati finanziari è una di queste situazioni. Data l'estrema imprevedibilità dei mercati, non esiste una mappatura "perfetta" delle probabilità di transizioni. Nonostante ciò, si è soliti lasciare valida questa ipotesi di prevedibilità, al fine di studiare più agevolmente i vari problemi di RL.

1.3.3 Goals & Rewards[20]

Nei problemi di RL, l'obiettivo (*goal*) dell'agente viene formalizzato all'interno di un segnale (che esso riceve dall'ambiente), chiamato *reward*, che consiste in un numero $R_t \in \mathbb{R}$. L'obiettivo dell'agente sarà quindi quello di massimizzare la quantità totale di

[20] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

ricompense ricevute, pertanto, non sarà orientato a massimizzare unicamente la ricompensa immediatamente successiva allo stato in cui si trova (come gli algoritmi di Supervised ML), ma bensì vorrà massimizzare le ricompense cumulate nel tempo. Ovviamente, cercherà di massimizzare il valore atteso di questa serie di ricompense cumulate, utilizzando un fattore di sconto che determina l'importanza delle rewards future rispetto al presente.

Alla luce di quanto appena detto, è evidente come la definizione della rewards function sia fondamentale per istruire l'agente a sviluppare un comportamento mirato per uno specifico obiettivo. In ambito di trading algoritmico si potrebbe pensare di utilizzare il rendimento cumulato come funzione di rewards da massimizzare, questa però non terrebbe conto di caratteristiche legate al rischio. Potrebbe ad esempio portarci ad una *policy* che genera elevati profitti se applicata per periodi lunghi ma che all'interno del suo *path* presenta enormi *drawdowns*[21] o eccessiva volatilità. Per fornire un esempio molto banale, si può pensare ad una *policy* che permetterebbe di ottenere un +150% sull'investimento in 3 anni ma che presenta un profilo di rischio pericoloso al suo interno, magari con *drawdown* fino al 50% e volatilità enorme, si ipotizzi 80% su base annua. In un contesto reale, un investitore non potrebbe sapere con certezza che tra 3 anni otterrà il 150% sul proprio investimento, però, durante questi 3 anni potrà osservare gli enormi *drawdowns* e la grande volatilità della strategia di investimento. Pertanto, la probabilità che l'investitore "spenga" l'algoritmo di investimento prima del termine dei 3 anni risulta molto elevata. Come ovviare a questo problema? Beh, si potrebbe pensare ad una funzione di *Reward* che tenga conto di queste caratteristiche di rischio, utilizzando per esempio lo Sharpe Ratio, il Gain-Loss ratio sui trades, etc. Anche in questa situazione, avere una certa esperienza del dominio sul quale si applicherà l'algoritmo, potrebbe fare la differenza.

In generale, l'agente cercherà di massimizzare il Valore Atteso di una sequenza di *rewards*, definita $\boldsymbol{G}$. E' importante sottolineare, soprattutto alla luce delle considerazioni fatte nel paragrafo precedente, che G è definibile come "*una qualsiasi funzione specifica ottenibile da una sequenza di rewards*". Pertanto, anche se nel corso della tesi verrà formalizzata come sommatoria delle *rewards* (e.g. somma dei rendimenti ottenuti da una scelta di investimento), nulla vieta di plasmare G al fine di indurre l'agente a sviluppare comportamenti differenti. Riprendendo l'esempio di sopra, G potrebbe essere lo Sharpe

[21] Il *drawdown* è una misura della perdita di valore di un investimento o di un portafoglio di investimenti rispetto al suo valore massimo precedente. In altre parole, il *drawdown* è la quantità di perdita che si verifica dal momento in cui l'investimento o il portafoglio raggiunge il suo valore massimo fino a quando non viene riportato al valore massimo precedente.

Ratio, o una qualsiasi funzione finanziaria, in un dato intervallo temporale. Prendendo un esempio assurdo, se G fosse il negativo della standard deviation, allora l'agente non farebbe mai trading. Questo perché, essendo la volatilità sempre maggiore di 0, cambiandola di segno il valore massimo che l'algoritmo potrebbe ottenere in G sarebbe 0 ed una volatilità nulla la si potrebbe ottenere facilmente non facendo mai trading. Le precedenti considerazioni, come d'altronde l'intera tesi, vogliono rimarcare l'importanza che la creatività e l'esperienza del dominio possono fare la differenza nella definizione di un problema di RL e, di conseguenza, nella "bravura" dell'algoritmo che si va a costruire per risolvere tale problema.

Nel più semplice dei casi, si potrebbe definire G come sommatoria delle *rewards*:

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T, \qquad (1.4)$$

Dove T rappresenta lo *step* finale. Ovviamente, questo approccio avrebbe senso soltanto in un'applicazione dove l'interazione agente-ambiente si interrompesse naturalmente ad un determinato *time-step*. Questo renderebbe possibile dividere il problema in sotto-sequenze, chiamati *Episodi*. Degli esempi potrebbero essere una serie di partite a scacchi, dove ogni partita rappresenta un episodio. Un esempio finanziario potrebbe essere un trimestre, intervallo temporale sul quale solitamente vengono valutate le *performance* di un *trader*. Ogni episodio potrebbe terminare in un modo differente, questo viene chiamato *terminal-state*, ovvero quello stato dopo il quale il *setting* dell'ambiente ritorna ad uno "stato iniziale". In questa tipologia di problemi si parlerà di *episodic-tasks.*

Nel caso in cui l'interazione agente-ambiente non è divisibile all'interno di episodi, ma continua illimitatamente nel tempo, si parlerà di *continuing tasks*. In questa situazione, si necessiterà di una formulazione diversa da quella definita nella (1.4) e bisognerà inserire un fattore di sconto (*discounting factor*). In tal caso l'agente, in ogni stato, selezionerà quell'azione che gli permetterà di massimizzare il valore attuale delle *rewards* future

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \cdots, \qquad (1.5)$$

dove $\gamma \in [0, 1]$ e viene chiamato *discount rate*. Dato che $\gamma^0 = 1$, G sarà dato dalla somma della reward immediatamente successiva più tutte le rewards future attualizzate. Se $\gamma = 0$ allora l'algoritmo sarà "miope", ovvero valuterà unicamente la reward immediatamente successiva; qualora $\gamma = 1$ allora l'algoritmo terrà molto in considerazione le future rewards; infine, qualora $\gamma < 1 \; and \; \gamma \neq 0$, quando $k \to \infty$ si avrà che $\gamma \to 0$.

1.3.4 Policy & Value Function[22]

Gli algoritmi di RL dovranno stimare la c.d. *Value-function*, una funzione degli stati (o delle coppie stati-azioni) che stima "quanto è buono" per l'agente trovarsi in un determinato stato (o "quanto è buono" selezionare un'azione dato che si trova in un determinato stato). Ovviamente, la quantificazione di "quanto è buono" è individuabile nelle future ricompense attese. Dato che queste ricompense saranno definite dal comportamento che l'agente avrà nel futuro, una *Value-function* sarà definita in accordo con un comportamento, una strategia, una *policy* π. Una policy è un *mapping* che parte dagli stati e definisce la probabilità di selezionare ogni possibile azione. Se un agente segue una policy π al tempo t, allora $\pi(a|s)$ sarà la probabilità che $A_t = a$ se $S_t = s$, ovvero la probabilità di selezionare una determinata azione trovandosi in un preciso stato. Semplificando, potremmo vederle come tante distribuzioni di probabilità, ognuna per ogni stato (essendo l'azione condizionata a questo), dove ogni distribuzione va a definire quale azione genera il valore atteso maggiore.

Pertanto, una *Value-function* di uno stato s seguendo la policy π, $v_\pi(s)$, sarà il valore atteso delle *rewards* assumendo di iniziare nello stato s e guidando le scelte seguendo π a partire da quel momento.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s\right], for\ all\ s \in \mathcal{S} \qquad (1.6)$$

Dove $\mathbb{E}_\pi$ rappresenta il valore atteso di una variabile aleatoria, dato che l'agente segue la policy π. Pertanto, data una policy π, ogni stato s avrà la sua *Value-function* che indicherà all'agente qual è il valore atteso (i.e. quanto può guadagnare) in un determinato stato se si comporta in un modo (i.e. segue una *policy*).

E' immediatamente derivabile un'altra relazione, che esprime il valore atteso di scegliere un'azione a in un determinato stato s, seguendo una policy π da quel momento in poi. Questa viene definita *Action-Value function*:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s, A_t = a\right] \qquad (1.7)$$

$v_\pi(s)$ e $q_\pi(s,a)$ possono essere stimate dall'esperienza. In un MDP in cui il numero di stati e azioni sono finiti e in numero limitato, è possibile utilizzare come funzione di valore una

[22] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

look-up table, che consiste semplicemente in una matrice in cui gli stati e le azioni corrispondono alle righe o alle colonne della stessa. In particolare, mentre l'agente prende decisioni sulla base della policy, si vanno ad aggiornare i valori della matrice mediando i valori delle esperienze passate in base alle rewards ottenuti, agli stati esplorati e alle azioni compiute. Questo porterà a una futura convergenza alla vera e propria funzione di valore descritta dalla matrice di dimensioni finite. L'utilizzo della lookup table non è attuabile nel caso in cui gli spazi degli stati o delle azioni siano troppo grandi o addirittura spazino in un insieme reale. In questo caso le funzioni di valore vengono approssimate, utilizzando delle funzioni parametrizzate i cui parametri vengono aggiornati in modo da avvicinarsi al valore corrispondente di ritorno di un particolare stato o di una coppia stato-azione.

1.3.5 Bellman Equation[23]

Una proprietà fondamentale della *Value-Function*, usata in tutto il RL e nella programmazione dinamica (*dynamic programming*), è quella di soddisfare una relazione di ricorsività.

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma v_\pi(s')], \qquad for\ all\ s \in \mathcal{S} \qquad (1.8)
\end{aligned}
$$

La (1.8) è l'**Equazione di Bellman** per la value function v_π**.** Esprime la relazione tra il valore di uno determinato stato e il valore dello stato successivo. La seguente figura, ne fornisce una rappresentazione:

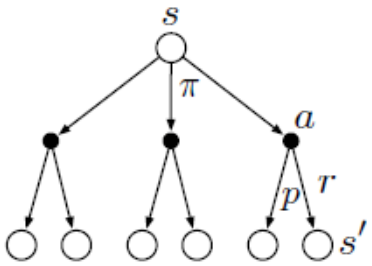


Figure 4: Backup diagram for v_π
Source: Sutton and Burton, "*Reinforcement Learning: an Introduction*", Second Edition, 2020

[23] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

Partendo da uno stato s (nodo in alto), l'agente potrebbe scegliere una qualsiasi azione (tre pallini neri), in base alla policy π (frecce al primo livello) che segue. In base alla scelta presa, l'ambiente risponderà con uno dei possibili stati futuri s' (due pallini vuoti in basso), che hanno una probabilità p di presentarsi (che seguono una *state transition probability function*) e, inoltre, remunererà l'agente con una *reward* r, in base alla dinamica della funzione p. La *Value-Function* v_π è l'unica soluzione dell'equazione di Bellman.

Intuitivamente l'equazione di Bellman permette di arrivare a una stima del valore prendendo in considerazione separatamente il reward e la stima dei reward futuri. Questo è possibile poiché si fa una media di tutte le possibilità future pesate sulla base della probabilità che occorrano.

1.3.6 Optimal Policies & Optimal Value Function[24]

Risolvere un problema di RL, semplificando, significa trovare una *policy* che ottiene elevate rewards durante il tempo. Una policy π è considerata migliore di una policy π' se e soltanto se il valore atteso delle rewards che ottiene è maggiore o uguale in ogni stato. Ovvero:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}, \qquad for\ all\ s \in \mathcal{S}$$

ed esiste sempre una policy che è migliore o uguale di un'altra. Questa viene definita *optimal policy*, e si scrive come π^*, da cui deriva l'*optimal value-function* t.c.

$$v_*(s) \doteq \max_\pi v_\pi(s), \qquad for\ all\ s \in \mathcal{S},$$

dove π^* è la policy ottimale che, se seguita, permette di ottenere il massimo valore atteso delle ricompense in un dato stato, i.e. la max value-function, per tutti gli stati. Trovare questa policy è l'obiettivo ultimo dei problemi di RL in quanto, ovviamente, porterebbe ai migliori risultati.

Quanto detto per la *Value-function* è applicabile anche alla *Action-Value function*:

$$q(s,a) \doteq \max_\pi q_\pi(s,a), \qquad for\ all\ s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Qualora un agente imparasse l'*optimal policy*, otterrebbe eccellenti risultati. Realisticamente parlando, questo accade molto raramente poiché, per i problemi in cui vale la pena utilizzare algoritmi di RL, la ricerca dell'ottimalità richiederebbe un costo in termini

[24] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

di calcolo computazione insostenibile. Sarebbe quindi ideale che l'agente riuscisse, quantomeno, ad approssima quella che è un *optimal policy*.

1.3.7 Exploration, Exploitation & Epsilon-Greedy policy[25]

Un agente, durante la formazione della propria *policy*, si troverà a dover scegliere quale azione prendere in base allo stato in cui si trova. Assumendo che esso disponga di una distribuzione di probabilità per ogni stato, i.e. una *value-function*, sembrerebbe logico selezionare (ogni volta) la scelta che, dato il valore atteso della distribuzione, garantisce la *reward* più elevata con la maggiore probabilità. Seguendo questo approccio nella selezione dell'azione, si parlerà di *greedy-actions* (azioni avide). Ogni qual volta si prenderà una scelta seguendo questa logica, si starà sfruttando (*exploitation*) la conoscenza di cui si dispone sul valore di un'azione in un dato stato. A volte però, potrebbe essere interessante provare delle azioni che non sono mai state scelte, per esplorare (*exploration*) la *reward* ottenibile ed aumentare la conoscenza dello stato da parte dell'agente. Queste vengono definire *non-greedy actions* (azioni non avide). Sebbene possa sembrar logico che sfruttare senza esplorare sia la soluzione migliore per massimizzare G_t nell'immediato, è bene tenere a mente che esplorare porterà a risultati di gran lunga migliori nel lungo termine. Questo perché non è possibile dire che "una scelta è la migliore" senza aver esplorato le conseguenze (i.e. le rewards) derivabili dal prendere scelte (i.e. azioni) differenti. E' proprio in questo contesto che si sviluppa il dilemma del *trade-off* tra *exploration vs exploitation*, di cui si è già parlato nei sotto capitoli precedenti. Ovviamente, durante la formazione di una *policy*, prendere delle azioni *greedy* non permette di prendere delle azioni *non-greedy*; pertanto, vi sarà una conflittualità tra le due scelte. Una possibile soluzione a questo *trade-off*, che poi è la soluzione più utilizzata nella pratica, sarà quella di selezionare delle azioni *greedy* nella maggior parte dei casi; quindi, sfruttare la conoscenza di cui si dispone, e di selezionare le azioni *non-greedy*, con una piccola probabilità ε, ovvero selezionare randomicamente un azione tra quelle *non-greedy* con uguale probabilità, a prescindere dalle stime azione-stato (*action-value function*) di cui si dispone. Questa tipologia di approccio si chiama ***Epsilon-greedy method***.

[25] Parzialmente tratto da S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd Edition, 2020

Per fornire un esempio, di seguito si riporta un grafico tratto da Sutton e Barton "*Reinforcement Learning: an Introduction*", dove gli autori confrontano le *rewards medie ottenute* da algoritmi *greedy* e *non-greedy*.

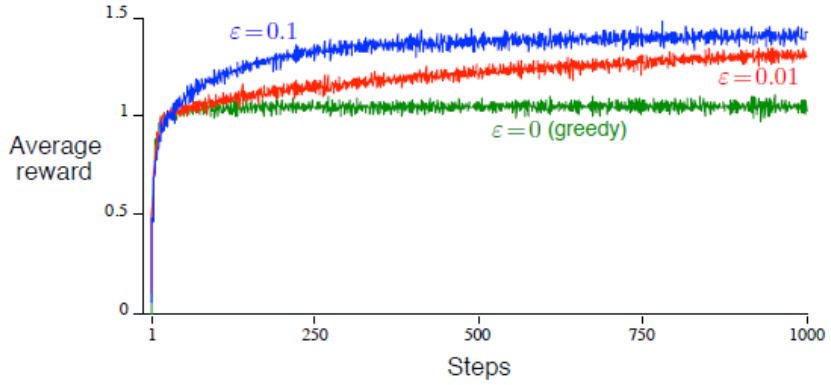


Figure 5: Average performances of greedy and not-greedy methods
Source: Sutton and Burton, "*Reinforcement Learning: an Introduction*", Second Edition, 2020

Gli algoritmi *greedy* inizialmente apprendono più velocemente, infatti nei primissimi *steps* si può notare come questi migliorino più rapidamente l'Average-reward. Però, al crescere dei *time-steps* gli *epsilon-greedy methods* performano nettamente meglio, questo perché l'esplorazione garantisce loro una più completa conoscenza delle azioni e degli stati, mentre i *greedy-methods* si "incastrano" molto più spesso nelle selezioni di azioni sub-ottimali.

1.3.8 Model-free e Model-based RL

Possiamo distinguere gli algoritmi di Reinforcement learning in due categorie differenti, algoritmi *Model-based* e algoritmi *Model-free*. In particolare, la prima tipologia utilizza un concetto molto importante chiamato Modello. Quest'ultimo consiste in una rappresentazione dell'*environment* creata all'interno dell'agente attraverso l'esperienza e l'interazione con esso. Si può pensare al modello come una componente concettualmente divisa in due parti. Una parte è dedicata alle transizioni, essa indica in particolare quali sono le condizioni necessarie affinchè vi sia un passaggio da uno stato s al tempo t a uno stato s' a un tempo $t + 1$. La seconda parte è dedicata al *reward*, in particolare il modello definisce quale reward è associato a una qualsiasi transizione di stato possibile nel sistema. Quando il modello è definito e completo, assumendo che quest'ultimo abbia una rappresentazione di tutto ciò che realmente rappresenta l'*environment*, l'agente è in grado di pianificare le proprie mosse sulla base di simulazioni dell'*environment* reale basate sul modello in modo da ottenere le migliori performance in termini di *reward* finale. Per quanto riguarda il *model-*

free reinforcement learning, si ha un algoritmo che non tenta di costruirsi una rappresentazione interna dell'*environment*. In questo caso l'agente intende imparare le funzioni di valore direttamente dall'esperienza derivata dall'interazione con l'ambiente a ogni singolo step. Sarà proprio questa la tipologia di modelli applicata nell'ambito del Trading Algoritmico

1.4 Modelli di RL per l'Algorithmic Trading

Sebbene esistano numerosi algoritmi di Reinforcement Learning, nel presente sotto capitolo verranno trattati solamente una piccola porzione di questi. Le ragioni di tale sale scelta sono sostanzialmente due: *in primis* questa Tesi non ha lo scopo di fare un'*overview* completa su tutti i modelli di RL e su come questi siano applicabili al mondo dell'Algorithmic Trading; *in secundis* non tutti gli algoritmi di RL sono utilizzabili per fare direttamente trading. In merito a quest'ultimo punto, si può pensare al Multi-Armed Bandits, il quale sostanzialmente ha l'obiettivo di selezionare la "leva" più profittevole scegliendo tra *k* leve, dove ognuna di queste leve è collegata a una Slot Machine che, ovviamente, potrà dare o non dare un premio dopo il relativo utilizzo. In questo caso, potremmo vedere le Slot Machine come delle strategie di trading differenti, pertanto l'algoritmo potrebbe andare a selezionare la leva (i.e. la strategia) che è più profittevole in un dato momento. Così facendo, l'algoritmo non farà direttamente trading, sebbene indirettamente si giunga allo stesso risultato (selezionando una strategia si avrà PNL, così come facendo direttamente trading). Dato che la presente Tesi ha come scopo quello di istruire un Agente a fare direttamente trading, è logico allora escludere il Multi-Armed Bandits dall'elenco dei potenziali algoritmi di RL da utilizzare, quindi da descrivere. Il medesimo ragionamento è applicabile per altri modelli di Reinforcement Learning.

The last but not the least, questa tesi non ha come unico scopo centrale quello di "dimostrare che il RL, applicato al trading algoritmico, è profittevole", bensì quello che vuole sperimentare e testare è se "un aspirante professionista, con un *background* prettamente finanziario, è in grado di apportare valore aggiunto (i.e. miglioramento nella profittabilità) nella costruzione di un sistema di trading che si affida al RL?". Ha senso andare a constatare quest'ultima affermazione poiché, negli ultimi decenni, l'influenza di Matematici, Statistici, Econometrici, ecc. sta evidentemente plasmando il mondo degli investimenti finanziari. Pertanto, coloro che si affacciano al mondo del lavoro possedendo unicamente conoscenze (squisitamente) finanziarie, potrebbero sentirsi scoraggiati dal dislivello nozionistico presente rispetto a figure con *backgrounds* molto più quantitativi. Questo è

lo scopo di questa Tesi! Mostrare come la conoscenza del dominio in cui si opera, accompagnata da un'adeguata conoscenza dei modelli che si devono utilizzare, può migliorare le *performance* di un sistema di trading, affermando quindi che la sostituzione delle figure "finanziarie" con figure prettamente "quantitative" è sbagliata! Bensì, la soluzione migliore è quella di permettere la cooperazione tra chi è "esperto della tecnologia" (i.e. *backgrounds quantitativi*) e chi è "esperto del dominio" (i.e. *background finanziari*).

1.4.1 Deep Q-Learning

Il *Deep Q-Learning*, è un algoritmo di Reinforcement Learning che viene utilizzato per risolvere problemi di ottimizzazione e di decisione in ambienti in cui lo stato dell'ambiente è noto. Lo stato dell'ambiente è noto quando si dispone di informazioni (i.e. dati) che descrivono l'ambiente in modo accurato. Nel caso dell'Algorithmic Trading, i dati che descriveranno gli stati, quindi l'ambiente, saranno prezzi e *features*. E' una variante dell'algoritmo di Q-Learning, che utilizza una rete neurale per approssimare la funzione di valore Q, che rappresenta il valore atteso di prendere un'azione in un dato stato, ovvero $Q(s, a)$.

Alcune delle ragioni per cui è necessario utilizzare una rete neurale che approssima il Q-value, piuttosto che affidarsi direttamente ad un algoritmo di Q-Learning, sono:

- *Elevata dimensione degli stati*. Prendendo il mondo del trading come esempio, il lettore potrà agevolmente immaginare quanto possa risultare complesso mappare tutti gli stati possibili con tutte le azioni accettabili. Si pensi ad una descrizione degli stati composta da un vettore con dieci valori (prezzo del titolo più nove *features*) e undici azioni (cinque quantitativi diversi che si possono comprare o vendere, più l'azione di non fare nulla). Risulta subito evidente la gigantesca quantità di combinazioni che si potrebbero generare, poiché per ogni combinazione di descrizione dello stato e di singola azione dovrebbe essere calcolato il valore $Q(s, a)$. E' impensabile risolvere questo problema utilizzando un approccio matriciale.
- *Funzione Q-value troppo complessa*, che rende difficile la sua definizioni in modo esplicito. Ad esempio, questa potrebbe essere non lineare.

- *Computational time*. Utilizzare una rete neurale per approssimare il valore Q, può ridurre il tempo di calcolo rispetto ad una Q-value function esplicita, poiché non è più necessario esplorare l'intero ambiente per poterlo mappare.

Alla luce di quanto appena detto, risulta necessario affrontare due *framework* teorici che sono alla base del Deep Q-Learning:

1. Q-Learning
2. ANN (*Artificial Neural Networks*), ovvero le Reti Neurali

1.4.2 Q-Learning

Il Q-Learning è un algoritmo di Reinforcement Learning (*control-model*[26], *model-free* e *off-policy*), annoverabile nella categoria degli algoritmi di TD (*Temporal Difference*). Lo scopo è quello di approssimare il valore $Q(s,a)$[27], $\forall s \in \mathcal{S}$ *and* $\forall a \in \mathcal{A}$, aggiornando le stime di Q, sia in base alle nuove informazioni (R_{t+1}) che in base allo scarto che si ha tra il valore di Q nello stato successivo e il valore Q nello stato attuale. Questa peculiarità permette di istruire l'Agente, insegnandogli che non è importante solo la *reward* che è conseguenza diretta dell'azione che si prende in s, bensì è rilevante anche la situazione (s') (in cui ci si ritrova dopo aver preso un'azione (a)) e l'azione (a') che si andrà a selezionare. Lo si potrebbe definire come un approccio di c.d. *forward-looking*.

In particolare, tra gli algoritmi di RL-TD si annoverano: *one-step* TD, Sarsa, Expected-Sarsa e Q-Learning. Ciò che differenzia questi algoritmi è individuabile in due caratteristiche fondamentali:

1. *La funzione che si va ad approssimare*. Nel caso del *one-step* TD, l'algoritmo va ad approssimare la Value-Function[28], mentre negli altri modelli si lavora sull'Action-Value function, ovvero la (1.7)
2. *In quale modo si tiene in considerazione "il futuro"*. Per una migliore spiegazione, risulta necessario fare una differenziazione tra due tipologie di modelli, i quali

[26] Un modello di controllo è un sistema di apprendimento che viene utilizzare per prendere decisioni (i.e. azioni) basate sullo stato attuale di un sistema.

[27] Equazione (1.7)

[28] Equazione (1.6)

definiscono il modo in cui vengono utilizzati i dati "di esperienza" per migliorare la prestazione dell'Agente, ovvero:

a. *On-Policy*: qualora l'Agente utilizzi la *policy* corrente per decidere la prossima azione da intraprendere ($a'|s'$)
b. *Off-Policy*: qualora l'Agente utilizzi una *policy* diversa da quella corrente per decidere la prossima azione ($a'|s'$)

Formalmente, quando l'Agente si trova in uno stato $s \in \mathcal{S}$ prenderà una decisione $a \in \mathcal{A}$ in base alla *policy* π corrente (e.g. *epsilon-greedy*), osserverà la *reward* immediata $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ e si ritroverà in uno stato successivo $s' \in \mathcal{S}^+$, dove dovrà scegliere nuovamente un'azione $a' \in \mathcal{A}^+$. Dopo aver osservato la reward e lo stato successivo, si andrà ad aggiornare il Q-value relativo a quel determinato stato e alla specifica azione intrapresa. Il metodo di aggiornamento del Q-value, utilizzato dal Q-Learning, è il seguente:

$$Q(s,a) \longleftarrow Q(s,a) + \alpha\left[R_{t+1} + \gamma \max_a Q(s',a') - Q(s,a)\right]. \quad (1.9)$$

Le variabili coinvolte sono:

- $Q(s,a)$: Il Q-value associato al prendere una determinazione azione in un determinato stato
- α: è il cosiddetto *learning-rate* ed è compreso tra 0 e 1. Definisce quanto velocemente i valori dell'action-value function devono essere aggiornati. Un valore prossimo a 0 farà sì che l'aggiornamento risulti estremamente graduale, viceversa un valore prossimo a 1 renderà ogni aggiornamento molto incisivo. Pertanto, la scelta di questo parametro apre un *trade-off* tra velocità di apprendimento e stabilità dei Q-values. Un valore spesso utilizzato è 0.5
- R_{t+1}: la *reward* immediata, conseguenza di aver scelto a in s
- γ: è un fattore di sconto "del futuro", compreso tra 0 e 1. Questo parametro controlla quanto è importante la sequenzialità nelle scelte. Utilizzando un valore prossimo allo 0, si darà molta più importanza al presente (i.e. R_{t+1}) rispetto al futuro (i.e. $\max_a Q(s',a')$), accadrà l'opposto inserendo un valore prossimo ad 1. Pertanto, in problemi dove la sequenzialità risulta importante (gli Scacchi per esempio) si preferirà un valore di γ elevato. Un valore spesso utilizzato è 0.9
- $Q(s',a')$: Action-Value del prendere l'azione a' nello stato successivo a quello corrente

- $\max_a Q(s', a')$: rappresenta il più alto Q-value ottenibile nello stato successivo. Pertanto, rappresenta l'azione che applicata in s' restituisce la reward maggiore.

Il Q-Learning viene definito un modello *Off-policy* per la presenza di $\max_a Q(s', a')$ nella (1.9). Seguendo le definizioni di cui sopra, un modello viene definito *Off-policy* qualora l'agente utilizzi una *policy* diversa da quella corrente per definire $a'|s'$. Effettivamente, selezionare l'azione a' che massimizza il Q-value in s' non è un approccio che si lega ad una *policy* specifica, bensì seleziona a' indipendentemente dalla *policy* che caratterizza l'*agent*.

Per fare un esempio di modello *Off-policy*, si potrebbe considerare il modello di TD chiamato Sarsa. Tale caratteristica è immediatamente osservabile dall'equazione che si utilizza per aggiornare i Q-values:

$$Q(s,a) \longleftarrow Q(s,a) + \alpha[R_{t+1} + \boldsymbol{\gamma Q}(\boldsymbol{s'}, \boldsymbol{a'}) - Q(s,a)].$$

In questo caso, l'azione da intraprendere in s' non è più quella che massimizza il Q-value, come nel caso del Q-Learning, bensì è l'azione che l'Agente intraprenderebbe se si ritrovasse a prendere una decisione in s'. È evidente come si stia utilizzando la *policy* corrente per determinare $a'|s'$.

Di seguito si riporta lo Pseudo-code per un algoritmo di Q-Learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 6: Q-Learning pseudo-code
Source: Sutton and Burton, "*Reinforcement Learning: an Introduction*", Second Edition, 2020

Pertanto, l'aggiornamento iterativo dei Q-values permette la convergenza della *policy* verso la sua ottimalità.

1.4.3 ANN (Artificial Neural Networks)

Un ANN, o *Rete Neurale Artificiale*, è un modello di ML ispirato al funzionamento del cervello umano. Esso è costituito da una serie di "*neuroni*" artificiali, che sono in grado di ricevere input, elaborarli e produrre output.

Il funzionamento di un ANN è abbastanza semplice: gli input vengono forniti alla rete e vengono elaborati attraverso una serie di strati di neuroni (*layers*). Ogni neurone riceve gli input dai neuroni dello strato precedente, li elabora attraverso una funzione di attivazione e li invia agli strati successivi. Gli output dell'ultimo strato di neuroni vengono quindi utilizzati come risultato finale dell'elaborazione.

Gli ANN possono essere addestrati utilizzando un processo di "*Backpropagation*", che consiste nel confrontare gli output della rete con i valori desiderati (sarà quindi un processo supervisionato in questo caso) e nel regolare i pesi dei collegamenti tra i neuroni per minimizzare l'errore (solitamente si utilizza MSE, o *Mean Squared Error*). Questo processo viene ripetuto più volte, utilizzando diverse coppie di input e output desiderati, fino a quando l'errore non raggiunge un livello accettabile. Una volta che l'ANN è stato addestrato (i.e. sono stati individuati i migliori pesi che collegano i neuroni), si possono passare dati input sconosciuti per ottenere degli output stimati.

L'elemento chiave che rende le reti neurali estremamente interessanti è il modo in cui esse processano le informazioni. Il *Perceptron* è la componente che rappresenta il neurone. Essa non è altro che una tipologia di classificatore binario che riceve un numero n di input I, opportunamente pesati tramite i pesi w_j, e mappa in uscita il corrispondente valore di y. Si può esprimere questa funzione nel modo seguente:

$$y(x_i) = f\left(\sum_{j=1}^{n} I_{i,j} * w_j\right),$$

dove:

- x_i è l'input *i*-esimo
- $I_{i,j} \in I_i$ è l'insieme delle *features* del corrispondente X_i
- $w_j \in W$, con W che corrisponde all'insieme dei pesi e $w_0 = \theta$ è chiamato *bias* e corrisponde ad una variabile di correzione

Graficamente il processo è il seguente:

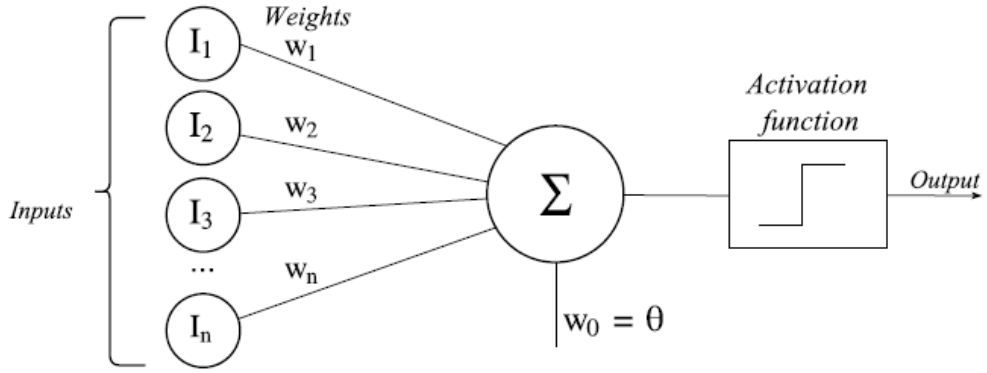


Figure 7: Funzionamento di un *Perceptron*

Pertanto, la funzione f, detta funzione di attivazione, modifica il risultato della sommatoria ottenendo l'output del *perceptron*. Diverse sono le funzioni di attivazione, tra cui:

- *"degli step"*: restituisce 1 se l'input è maggiore ad una soglia predeterminata, altrimenti restituisce 0
- *Sigmoid*: applica la seguente trasformazione $\frac{1}{(1+e^{-x})}$ e restituisce un valore compreso tra 0 e 1, a seconda dei dati input. E' un ottima funzione che tiene conto di relazioni non lineari tra le variabili input e serve a "ingabbiare" i valori output di una relazione lineare tra 0 e 1, motivo per cui viene utilizzata anche nel modello Logit.
- *ReLu* (*Rectified Linear Unit*): definita da $f(x) = \max(0, x)$, restituirà il valore dell'input se questo è maggiore di 0, altrimenti 0. Se la *threshold* della funzione di attivazione degli step fosse 0, il funzionamento sarebbe uguale a quello di una ReLu. La differenza risiede nel fatto che la funzione degli step ha cambiamenti "bruschi" tra 0 e 1, mentre la ReLu è continua e a mano a mano che l'input è maggiore di 0 restituisce un output, pari ad x. Solitamente, una ReLu è più facile da addestrare poiché avendo dei cambiamenti meno "bruschi", impatterà in modo graduale sull'aggiustamento dei pesi w_j

Si può quindi scegliere la funzione di attivazione desiderata, lineare o non, a seconda della tipologia di inputs di cui si dispone e dalla natura del problema da risolvere.

L'insieme di più *perceptron* (i.e. neuroni) forma un *Layer*. Esso corrisponde ad una funzione $f(x)$ avente in ingresso n input I e in uscita un numero di output pari al numero di neuroni che compongono il *layer* finale. Questi *layers* dovranno poi essere combinati per ottenere una *Rete Neurale Completa*, definendo:

- L'*input-layer* che prenderà n input I
- k *Hidden-Layers* formati da un numero variabile di neuroni. Solitamente il $layer_{k+1}$ avrà un numero di neuroni pari al doppio di quelli del $layer_k$. Inoltre, maggiore sarà k (i.e. più saranno i *layers* nascosti) e maggiore sarà la complessità della Rete Neurale che si andrà a definire. Difatti, una rete si definirà *Deep* (profonda) nel momento in cui si aumenta il numero di *layers* intermedi.
- L'*Ouput-Layer*, costituito da un numero di neuroni pari al numero di output che si desidera

La maggior parte delle Reti Neurali è di tipo *Fully-Connected*, ovvero ogni Neurone di un *layer* è connesso a tutti i Neuroni del *layer* precedente e di quello successivo.

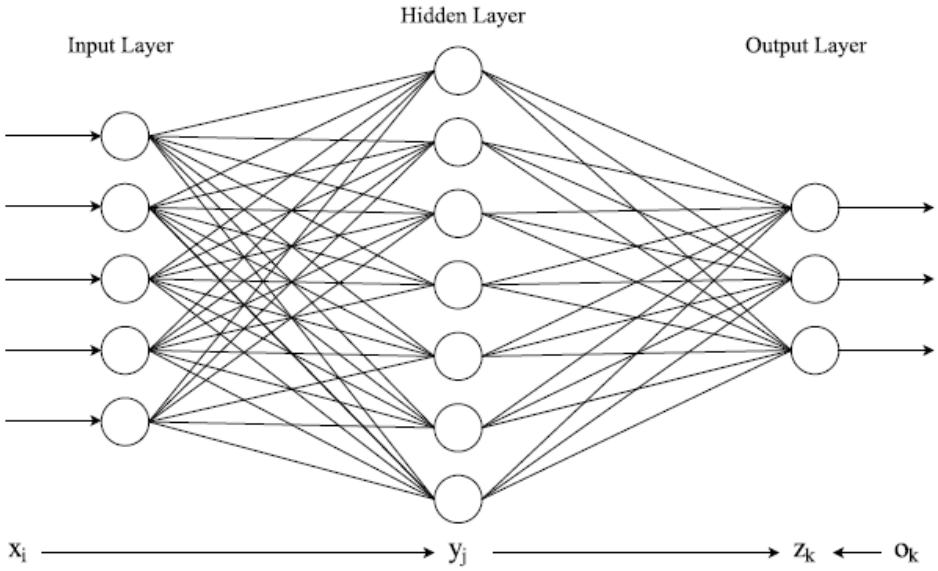


Figure 8: Esempio di Rete Neurale *fully-connected* con 5 input, 1 *hidden-layer* e 3 output

Lo scopo finale di una ANN è quello di approssimare una qualche funzione f, in modo che l'output sia il più vicino possibile al risultato migliore. Per esempio, nel caso di un problema di classificazione, una ANN ha lo scopo di trovare una funzione parametrizzata $f(x;\theta) = y$, dove θ rappresenta i pesi della rete, che mappi gli input x in un'opportuna classe y in output.

Si ricorda infine che in una Deep ANN, così come in molti altri modelli, vi è il rischio di *overfitting*. Pertanto, se inizialmente si potrebbe pensare che aumentare il numero di *hidden-layers* possa migliorare la qualità della rete neurale, bisognerebbe anche tenere in

considerazione l'elevato rischio di ottenere ottimi risultati in fase di *training* ma, a causa di una pessima capacità di generalizzazione, dei pessimi risultati in fase di *testing*.

1.4.4 Function Approximator – Deep Q Network

Considerando un numero discreto e limitato di coppie stato-azione, la action-value function Q(s, a) può essere rappresentata semplicemente con una matrice in cui ogni riga rappresenta i possibili stati e ogni colonna rappresenta le possibili azioni. I valori della matrice in questo caso corrispondono esattamente alla funzione Q (s, a). Pertanto, in fase di aggiornamento dei Q-values, altro non si farà che andare a sostituire il vecchio q-value, associato ad una coppia stato-azione, con il valore risultante dalla nuova esperienza che l'agente"vive". Tale approccio però, può risultare inapplicabile in presenza di situazioni come:

- *Elevato numero di coppie stato-azione*, che rende ingestibile la tabella da un punto di vista matriciale
- *Spazio degli stati continuo*, ovvero definito in tutto $\mathbb{R}$. In questo caso, non solo la matrice dei q-values non è manutenibile, ma risulta effettivamente impossibile memorizzare ogni singolo valore senza applicare una discretizzazione dello spazio degli stati
- *Spazio degli stati che non viene interamente "coperto" dai dati di training*. In questa situazione in cui i dati di training non contengono tutte le possibili combinazioni che lo spazio degli stati potrebbe avere, risulterebbe impossibile mappare tutte le coppie stato-azione, lasciando quindi "l'intelligenza" dell'agente incompleta, poiché vi è effettivamente un *lack* in termini di esperienza, causato dall'assenza dei dati che descrivono tali condizioni.

È evidente come le possibili condizioni appena descritte siano presenti nel mondo del trading algoritmico. Infatti:

- I dati finanziari, nel più dei casi, risultano definiti in uno spazio continuo
- Costruendo uno stato con delle *features*, perlopiù definite in uno spazio continuo, risulta enorme il numero di stati possibili, e conseguentemente di coppie stato-azione, che bisognerebbe mappare
- E' improbabile che i dati di training, con il quale si allena un'agente di RL, possano contenere tutti gli stati possibili (i.e. tutte le combinazioni di *features*). Si pensi ad

esempio ad uno scenario Covid, impensabile fino a che non è accaduto, pertanto non apprendibile da parte di un'agente. In parte, questa problematica può essere risolta con la generazione di dati sintetici, ad esempio usando un modello GAN, che possono aumentare significativamente la numerosità dei dati di trading di cui si dispone

Date le problematiche sopra descritte, si è reso necessario l'utilizzo di una nuova rappresentazione della *value-function*, che prende il nome di *Value-Function-Approximator*.

Una funzione di approssimazione è una qualsiasi funzione parametrizzata, i cui parametri sono indicati da θ, per cui vale la seguente formulazione:

$$\hat{v}(s;\theta) \approx v_\pi(s)$$

$$\hat{q}(s;a;\theta) \approx q_\pi(s;a)$$

Sostanzialmente, una funzione di approssimazione permetterà, tramite i parametri θ, di mappare i vari stati senza la necessità di disporre di una matrice che ne descrive tutte le possibili combinazioni.

Questa tipologia di funzione darà un vantaggio sia dal punto di vista della gestione della memoria in fase di calcolo, sia dal punto di vista della generalizzazione dell'algoritmo. Quest'ultimo beneficio aiuterà a risolvere il bullet point n.3 di cui sopra, ovvero la problematica tale per cui si potrebbero verificare situazioni in cui le combinazioni di *features* (i.e. lo stato) non è mai stato incontrato da parte dell'agente. Con una funzione di approssimazione, si sfrutterebbero i parametri θ per stimare i q-values di uno stato che non si è mai verificato.

Può essere utilizzato un qualsiasi tipo di funzione di approssimazione comune anche a vari algoritmi di Supervised Learning (e.g. Decision trees, Nearest neighbour etc.), ma le funzioni utilizzate più comunemente negli ultimi anni sono le Reti Neurali.

Esse hanno la caratteristica di essere funzioni differenziabili. La necessità di funzioni differenziabili permette di adattare i parametri della funzione di approssimazione utilizzando una metodologia molto comune all'interno del Deep Learning, la *Backpropagation*.

In particolare, un modo di utilizzare l'ANN come *value-function-approximator*, nonché la metodologia che verrà utilizzata nella parte sperimentale di questa tesi, è quella di definire N features in input per uno stato e tramite una rete neurale ottenere K q-values in output, dove k rappresenta il numero di azioni possibili in uno stato.

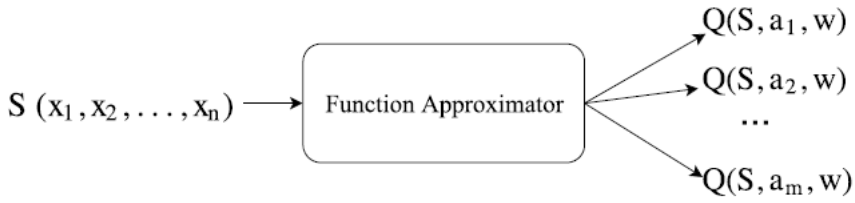


Figure 9: Funzione di approssimazione

1.4.5 Experience Replay & Target Model

L'**Experience Replay** è una tecnica utilizzata in molti algoritmi di apprendimento per agenti basati su reti neurali, in particolare nel Deep Q-Network (DQN). Durante l'interazione tra l'agente e l'environment tutte le esperienze < s, a, r, s' > (stato, azione, reward e stato successivo) vengono salvate in una replay memory, memoria di grandezza fissa e di tipo FIFO (i.e. *First In First Out*). Durante l'allenamento della rete, invece di utilizzare le recenti esperienze una di seguito all'altra, vengono utilizzati dei mini-batch di esperienze prese in modo casuale all'interno della replay memory.

Il vantaggio di utilizzare l'Experience Replay è che consente all'agente di rivisitare esperienze passate e di apprendere da esse, anziché solo dalla singola esperienza corrente. Ciò aumenta la varietà di dati che la Rete Neurale utilizza per l'addestramento e migliora la stabilità del processo di apprendimento. Inoltre, l'Experience Replay aiuta a prevenire il problema della dipendenza dalla sequenza, in cui l'agente potrebbe imparare un comportamento sbagliato se le esperienze sono state presentate in un ordine particolare.

Questa tecnica ha dimostrato di essere molto efficace nel migliorare la capacità dell'agente di imparare da esperienze precedenti e di raggiungere prestazioni ottimali nel tempo.

Il secondo metodo viene detto **Target Model**. Durante la computazione del gradiente e quindi durante l'aggiornamento dei pesi della rete effettuato con la tecnica di *backpropagation*, si utilizzano contemporaneamente due funzioni di approssimazione, i.e. due reti strutturalmente identiche ma con pesi differenti. Dopo un certo numero di passi, stabilito in base alle esigenze, la Target Q-Network sarà aggiornata con i pesi dell'altra rete, c.d. Online Q-Network.

1.4.6 Pseudo-Code

Di seguito si riporta lo pseudo-code relativo ad un algoritmo di Deep-Q-Learning (DQN). Si possono apprezzare due differenze evidenti, rispetto allo pseudo-code sul Q-Learning precedentemente presentato:

- L'utilizzo dell'Experience Replay alle righe 1, 10 e 11. Infatti, alla riga 1 viene creato un *buffer* in memoria per memorizzare le transizioni <s, a, r, s'>. Queste vengono memorizzate nel loop alla riga 10 e ne viene estratto un campione (*minibatch*) alla riga 11 per poi allenare l'Online-ANN .
- L'utilizzo del Target Model alle righe 2, 3, 13 e 14. Si può notare come alla riga 3 viene inizializzata la Target-ANN con gli stessi pesi e con la stessa architettura dell'Online-ANN della riga 2. Durante l'apprendimento però, solamente l'Online-ANN viene aggiornata (riga 13) e ogni C steps viene aggiornata la Target (riga 14).

Deep Q-Learning Algorithm

Algorithm 1 DQN Algorithm adapted from [10]

1: Replay memory D initialization
2: Action-value function Q initialization with random weights θ
3: Target action-value function $\hat{Q}$ initialization with weights $\theta^- = \theta$
4: **for** episode=1 to M **do**
5: Initialize sequences $s_1 = \{x_1\}$ and preprocessed sequence $\phi = \phi(s_1)$
6: **for** t=1 to T **do**
7: ϵ-greedy policy, select $a_t = \begin{cases} a \text{ random action} & \text{with prob. } \epsilon \\ argmax_a Q(\phi(s_t), a; \theta) & \text{otherwise} \end{cases}$
8: Execute action a_t in emulator and observe reward r_t and image x_{t+1}
9: Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
11: Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
12: Set $y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ is terminal} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
13: Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t. the network parameter θ
14: Every C steps reset $\hat{Q} = Q$, i.e. set $\theta^- = \theta$
15: **end for**
16: **end for**

Figure 10: Deep Q-Learning pseudo-code

1.4.7 Double DQN (DDQN)

Per comprendere il funzionamento del DDQN, si rende necessario introdurre brevemente il concetto di Double Q-Learning. Si immagini di dividere un problema di RL in due *sets*, ovvero creare due differenti tabelle per i Q-values ed effettuare stime differenti, chiamate $Q_1(a)$ e $Q_2(a)$, dove ognuna di queste vuole essere una stima del vero valore di

$q(a), for\ all\ a \in \mathcal{A}$. Infine, quando si andranno ad aggiornare i valori Q, in modo randomico si aggiornerà una stima relativa ad uno o dall'altro set. Ipotizzando il lancio di una moneta, se esce testa l'aggiornamento sarà il seguente:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha\left[R_{t+1} + \gamma Q_2\left(s', \underset{a'}{arg\max}\, Q_1(s',a')\right) - Q_1(s,a)\right], \quad (1.10)$$

altrimenti si andrà ad aggiornare $Q_2(s,a)$ invertendo Q_1 con Q_2 all'interno della formula.

Sostanzialmente, il Double Q-Learning utilizza un set di Q-values (Q_1 nella (1.10)) per selezionare l'azione a' che, data una *policy*, verrà presa in s', per poi andare ad utilizzare tale azione per estrarre il Q-value derivante dall'altro set di Q-values (Q_2 nella (1.10)).

Lo scopo del Double Q-Learning è quello di eliminare il cosiddetto *maximization bias.* Tale bias è dovuto dall'operatore di massimo all'interno della formula $Q(s,a)$ del Q-Learning. Accade poiché la stima $\mathbb{E}\left\{\max_a Q(s',a)\right\}$ non è reale, ma è *biased*, ovvero si discosta dal valore reale. Risulta essere quindi un'approssimazione "rumorosa" della soluzione ottimale Q_*. Considerando, per esempio, un set di azioni con lo stesso valore reale di action-value function in un certo stato. La stima che viene fatta dal nostro algoritmo, essendo rumorosa, conterrà delle variazioni in positivo o in negativo del valore reale. Ciò che viene implicitamente fatto dall'operatore di massimo è la scelta dell'azione che ha l'errore di stima più alto in positivo, ripercuotendo la scelta di questa azione anche agli stati e alle selezioni successive. Questo porta ad un'accumulazione dell'errore che crea seri problemi alla stabilità dell'algoritmo.

Il cambiamento effettuato all'interno dell'algoritmo di DQN non è così sostanziale come si potrebbe pensare. Infatti, avendo a disposizione già due reti, la Target Q-Network e la Online Q-Network, è possibile solamente cambiare la scelta alla riga 12 di 3.2.4 come segue:

$$y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ is terminal} \\ r_j + \gamma \hat{Q}(\phi_{j+1}, argmax_{a_j} Q(\phi_{j+1}, a_j; \theta); \theta^-) & \text{otherwise} \end{cases}$$

Quello che cambia dunque è l'utilizzo dell'Online-ANN per scegliere l'azione (Q_1 nella (1.10)) e la Target-ANN per generare il target Q-value per quell'azione (Q_2 nella (1.10)). In questo modo si riduce la sovrastima stabilizzando l'algoritmo.

CAPITOLO 2

DEFINIZIONE DEL PROBLEMA & ARCHITETTURA DEL MODELLO

2.1 Definizione del Problema

Alla luce del Framework Teorico, descritto nel Capitolo 1, si è deciso di implementare un Agente di Double-Deep-Q-Network (DDQN). Il "problema" di RL che l'agente dovrà risolvere, sarà quello di massimizzare le *rewards* ottenute dal *trading* effettuato su di un singolo titolo. Nonostante si potrebbe vedere il *trading* come un episodio continuo, al fine di ottimizzare l'allenamento dell'Agente, si è deciso di suddividere il *training* in N episodi finiti.

Ad ogni episodio, l'Agente verrà chiamato a massimizzare le *rewards*, che per semplicità coincidono con il PNL generato ad ogni *step*, potendo scegliere tra 3 azioni: *buy*, *sell*, *do-nothing*. In particolare, l'ultima azione corrisponderà a non fare trading, qualora l'Agente iniziasse lo *step* con 0 quantità del titolo in portafoglio, oppure con il mantenere la posizione precedentemente presa sul titolo, qualora detenesse dapprima lo stesso in portafoglio. Al fine di evitare un comportamento passivo da parte dell'Agente, è stata inserita una penalizzazione per ogni *step* in cui non si detengono titoli.

In merito al set di *features*, si è deciso di optare per uno *zoo* bilanciato in termini di informazioni sul *path* del trend e sulla possibilità che lo stesso cambi direzione (i.e. *mean-reverting path*). Questo al fine di evitare che l'Agente si specializzi in una tipologia di trading piuttosto che un'altra, ricercando quindi la maggiore generalizzazione possibile. Si rimanda al 2.2.2 "*Features Engineering*" per un approfondimento in merito alla costruzione dello zoo di features.

La scelta di implementare il modello utilizzando il linguaggio Python, è motivata sia dalla indiscutibile diffusione del suddetto linguaggio, sia dall'ampia disponibilità di librerie funzionali allo scopo della parte sperimentale della tesi.

Si rimanda ai successivi paragrafi per una descrizione, accurata ed esaustiva, sulla costruzione dell'architettura del modello di DDQN.

2.2 Dati & Indicatori

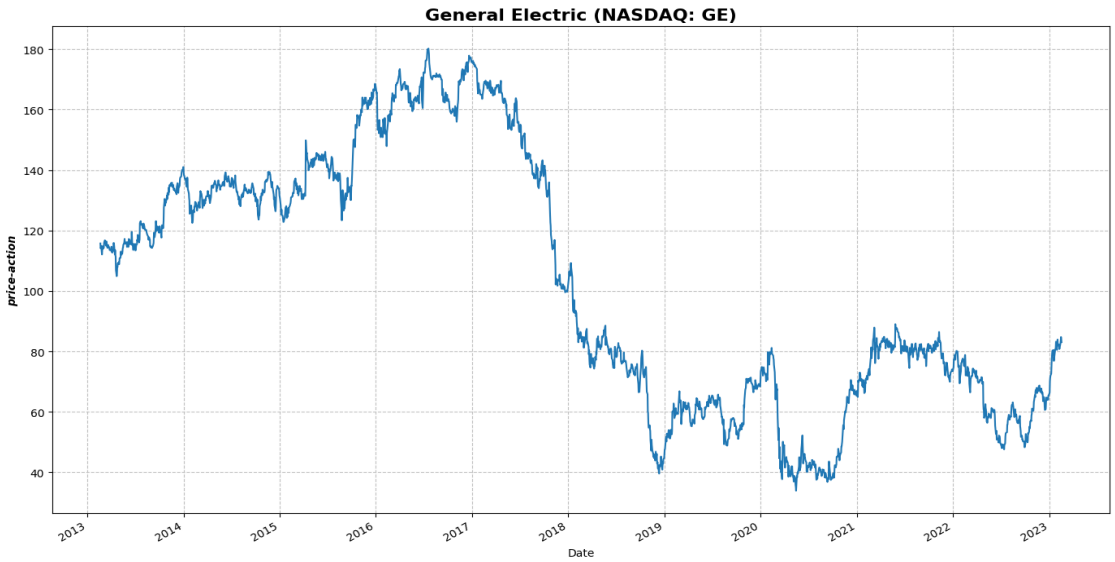
Nei successivi sotto-paragrafi, si andranno a descrivere la natura delle serie temporali finanziarie che sono state scelte per la parte sperimentale della presente tesi. Nel sottoparagrafo 2.2.2 "Features Engineering", si andranno invece ad esplodere singolarmente le *features* che sono state utilizzate per la creazione degli stati, ovvero di tutto ciò che è stato disponibile al DDQN Agent per l'apprendimento. In merito alle *features*, risulta importante specificare che, come accennato all'inizio della tesi, non esiste un *pool* "magico" di *features* che permettono all'algoritmo di disporre di tutte le informazioni necessarie per essere un "buon algoritmo". Bensì, la composizione del suddetto *pool* può (e deve) variare da caso a caso, in base alla natura dei dati che si stanno utilizzando, all'obiettivo che l'algoritmo deve raggiungere e, più spesso di quanto si crede, alla disponibilità di dati che si ha. Difatti, la *features selection*, risulta essere una branca di studi a sé stante, pertanto, al fine della costruzione del *pool* di *features* che è stato fornito all'algoritmo, si è optato per serie che dessero sia informazioni di *trend-following* sia di *mean-reverting*, al fine di evitare l'ancoraggio dell'Agente ad uno specifico *trading-style*, nonché per permettere all'intelligenza dell'algoritmo di avere una maggiore capacità di generalizzazione.

2.2.1 Dati

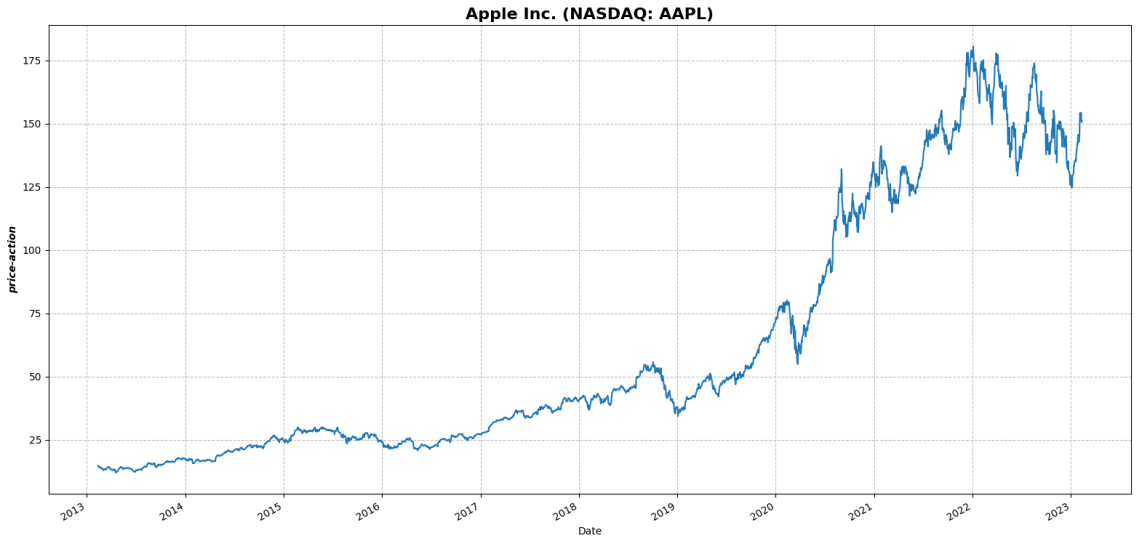
General Electric (NASDAQ: GE) è il titolo che è stato utilizzato per svolgere la simulazione ed il test del DDQN-Agent. Il dataset presenta le seguenti caratteristiche:

- La frequenza dei dati è giornaliera
- La serie storica in analisi inizia il 2013-02-19 e termina il 2023-02-17 (i.e. 10y)
- Il training-set inizia il 2013-02-19 e termina il 2020-12-23
- Il test-set inizia il 2021-01-04 e termina il 2022-06-01
- Il Data Provider è Yahoo Finance, in quanto l'API fornita da questo risulta essere molto versatile e flessibile nell'utilizzo.

La scelta di General Electric come titolo *target* è motivata dalla sua *price-action* nel periodo d'analisi. Questa risulta essere, tendenzialmente "bilanciata", come risulta evidente nel grafico sottostante che riporta i *closing-price*:



Per "bilanciato", si vuole intendere che la *price-action* presenta sia dei *bullish-paths*, come ad esempio il periodo 2013-2017 o anche metà2020-inizio2021, sia dei *bearish-paths*, come ad esempio il periodo 2017-2019 o anche inizio2020-metà2020. Tale comportamento dei prezzi, date anche le *features* fornitogli che verranno descritte nel successivo sotto-paragrafo, permette di costruire un algoritmo che è in grado di generalizzare su diverse *price-actions*. Questo non vuole significare che allenando l'Agente su *paths* similari si ottengono generalizzazioni migliori rispetto all'allenamento dello stesso su *bullish/bearish paths*, poiché "l'intelligenza" dell'agente rimane comunque ancorata ai dati fornitogli (i.e. alla serie storica sulla quale lo si allena), se il *training* avviene come in questa tesi. Idealmente, quello che si è voluto evitare è l'allenamento su serie storiche con un *trend* particolarmente forte. Si pensi ad esempio al titolo Apple Inc. (NASDAQ: AAPL) nello stesso periodo:



Risulta da subito evidente il fortissimo *bullish path* del titolo. Indubbiamente, è possibile allenare un Agente di DDQN anche su questa serie storica. Il risultato però, sarebbe quello di osservare l'Agente andare *long* nel più dei casi, oppure tentare di sfruttare anche i ribassi pagandone in *transaction costs*. Al fine di evitare lo studio di un'Agente che opera su di un titolo che ha subito un ~6x in 10 anni, si è deciso di scegliere un titolo più equilibrato, come General Electric (NASDAQ: GE).

Le considerazioni di cui sopra, sono frutto di analisi svolte nell'ambito della presente tesi, in quanto inizialmente le simulazioni sono state lanciate utilizzando Apple come titolo *target* e, alla luce di quanto appena detto, si è deciso di optare per un titolo con *price-action* differente.

2.2.2 Features Engineering

Innumerevoli potrebbero essere le *Features Engineering* (FE) che si possono costruire su di una serie storica. Questo è vero perché, contestualizzando lo studio su di un titolo finanziario, ogni *features* che in qualche modo fornisce, almeno ipoteticamente, informazioni su di una *security*, risulta essere un'ottima candidata. Si possono costruire features a partire da:

- *Analisi Tecnica*: queste aiutano ad eliminare il "rumore" o a cogliere informazioni sul comportamento di una serie nel tempo. Ad esempio: Bande di Bollinger, Medie mobili, RSI, etc.
- *Dummies*: provano ad insegnare ad un algoritmo come alcuni eventi/situazioni potrebbero essere ritenute rilevanti per descrivere il perché di una *price action*. Ad esempio, eventi Macro, *trading sessions*, stagionalità, etc.
- *Altri modelli*: utilizzare quindi l'output di altri modelli come *features*. Ad esempio si potrebbe usare un GARCH per fornire una stima della volatilità, oppure l'output di una Logit trasformato in una dummies per fornire una stima dalla quale apprendere.
- *Macro*: fornendo all'algoritmo informazioni in merito a dati macro, che potrebbero influenzare il comportamento di un titolo. Ad esempio: tassi d'interesse ufficiali, dati sul mercato del lavoro, etc. Questo è particolarmente vero per il mercato Forex.

- *Testuali, Video*: qui si sfocia nel mondo degli *Alternatives Data*, ovvero quei dati che non nascono con una natura quantitativa e che, tramite particolari modelli ed elaborazioni, riescono ad essere analizzati e a fornire informazioni numeriche. Un esempio sono le tecniche di NLP (*Natural Language Processing*) che vengono utilizzate per estrarre il *sentiment* del mercato, a partire da dati testuali. Si può pensare ai *tweets*, ai documenti pubblicati da enti accreditati quali Banche Centrali, Governi, summaries dell'azienda target, etc.

In particolare, all'interno del modello che è stato implementato, si è deciso di utilizzare unicamente FE derivanti dal mondo dell'analisi tecnica. Tale scelta non è motivata da un'effettiva preferenza per l'analisi tecnica o perché le FE proveniente da questa forniscono un'informazione migliore all'algoritmo, bensì, perché la presente tesi vuole concentrarsi sul funzionamento di un algoritmo di DDQN applicato al trading piuttosto che andare ad approfondire l'immenso mondo dell'ingegnerizzazione delle *features*.

Il *pool* di features che è stato costruito è il seguente:

- Medie Mobili – Moving Averages
    - SMA
    - MACD
- Relative Strength Index (RSI)
- Bande di Bollinger
- Price Volatility
    - EMA
- Momentum – ROC
- Lag Returns
- Sigma Events (*dummy*)

Si andranno ora a descrivere singolarmente le *features*, la loro costruzione e il loro utilizzo nel mondo del trading algoritmico.

2.2.2.1 Medie Mobili – Moving Averages

Le medie mobili sono una delle tecniche di analisi tecnica più utilizzate nel trading, sia manuale che algoritmico. Questi strumenti sono utilizzati per filtrare il rumore di mercato e individuare tendenze a lungo termine nei prezzi dei titoli.

In termini tecnici, una media mobile è un indicatore che calcola la media dei prezzi di un titolo su un periodo di tempo specifico, solitamente giorni o settimane. Ad esempio, una media mobile a 50 giorni calcolerà la media dei prezzi di un titolo sugli ultimi 50 giorni. La media mobile viene quindi tracciata sul grafico dei prezzi, creando una linea che segue l'andamento dei prezzi.

Le medie mobili possono essere utilizzate per identificare tendenze a lungo termine, ad esempio se un titolo sta tendendo al rialzo o al ribasso. Inoltre, le medie mobili possono essere utilizzate per identificare i punti di ingresso e di uscita del mercato, poiché i trader possono guardare per incroci tra la media mobile e i prezzi del titolo

Nel trading algoritmico, le medie mobili sono spesso utilizzate come parte di una strategia più complessa. Ad esempio, un algoritmo potrebbe utilizzare medie mobili per identificare tendenze a lungo termine e poi utilizzare altri indicatori tecnici, come l'oscillatore stocastico o il RSI, per confermare queste tendenze e identificare punti di ingresso e uscita.

È importante notare che le medie mobili non sono un indicatore perfetto e potrebbero essere influenzate da fattori esterni, come ad esempio *news* o fluttuazioni del mercato. Pertanto, è importante utilizzare una combinazione di indicatori tecnici e analisi fondamentale per prendere decisioni di trading informate.

Si possono creare diverse tipologie di medie mobili: *Simple Moving Average* (SMA), *Exponential Weighted Moving Average* (EWMA), KAMA (*Kaufman Adaptive Moving Average*), etc. Ognuna di queste avrà le proprie specifiche caratteristiche. Ad esempio la SMA la si potrebbe definire come la media al gusto "vaniglia", ovvero la più semplice; l'EWMA tende invece a pesare maggiormente le osservazioni più recenti, sarà quindi più ancorata al presente; la KAMA cerca invece di rimanere *flat* in periodi in cui vi è solo *noise* e non vi è trend, per poi reagire molto rapidamente non appena inizia un trend, è infatti una media con memoria.

SMA

Per la costruzione dello *zoo* di FE che verranno fornite al modello di RL, si è scelto di utilizzare la SMA, la formulazione è la seguente:

$$SMA_t(k) = \frac{1}{k} \sum_{t=T-k}^{T} x_t,$$

dove k sono i periodi che si decidono di includere nel calcolo della SMA ed x_t possono essere prezzi o rendimenti[29]. Ad esempio, avendo a disposizione un periodo lungo 1 mese e scegliendo una media mobile a 10 giorni, per ogni t a partire dal decimo giorno si andrà a calcolare tale media, ottenendo così una serie storica che rappresenta la $SMA(10)$. E' importante ricordare che la finestra con cui si decide di costruire una media mobile ne influenzerà l'informazione che essa fornisce. Una finestra ampia darà consistenza alla serie, rendendola allo stesso tempo meno reattiva. Viceversa, una finestra breve farà si che la media risulti molto reattiva, "pagando" un po' di *noise*.

Di seguito si riporta una SMA(1y) costruita sui dati giornalieri dell'SP500.

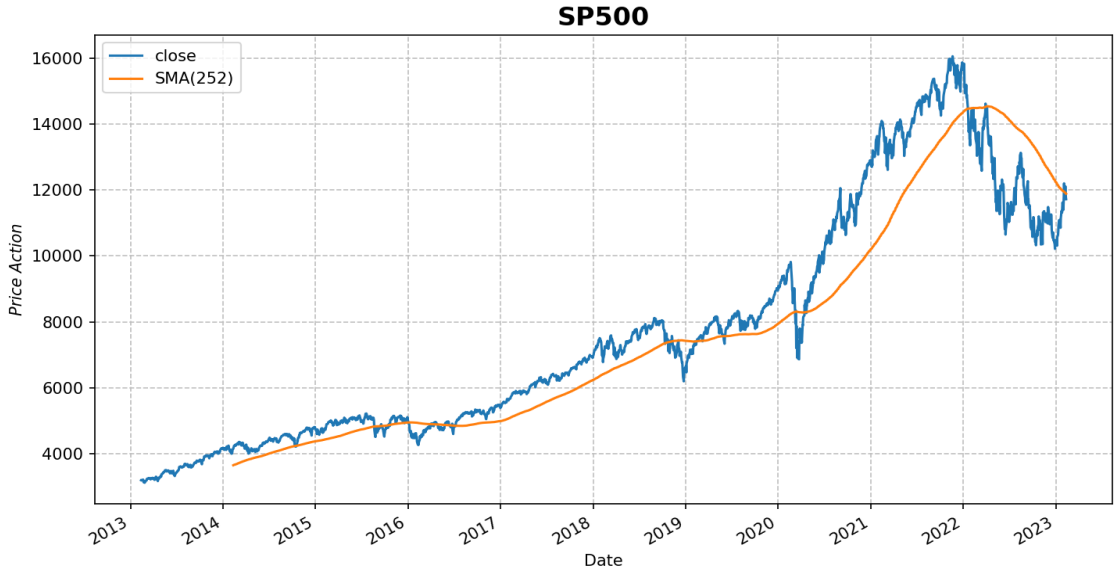


Figure 11: SMA(252) dell'SP500

È evidente come quando il prezzo è maggiore della SMA ci si trova in periodi c.d. *Bullish*, mentre quando il prezzo è inferiore alla SMA ci si trova in periodi c.d. *Bearish*.

MACD

Molto spesso, per evitare che il *noise* contenuto nella *price action* influenzi la strategia di trading derivante dall'utilizzo delle medie mobili, chiamata di *Trend-Following*, si tende ad "incrociare" due medie mobili ed utilizzare questa differenza per definire se ci si trova in

[29] Nulla vieta di applicare una SMA su degli indicatori. Ad esempio, si potrebbe voler fare *smoothing* sull'RSI.

un periodo rialzista o in un periodo ribassista. Se si utilizzano medie mobili esponenziali (EWMA) incrociate, piuttosto che medie mobili semplici (SMA), si parlerà di MACD (*Moving Average Convergence Divergence*).

Le due medie vengono costruite utilizzando finestre differenti, una corta e una lunga. Solitamente si prediligono 12 periodi per la corta e 26 per la lunga. Un modo "classico" di utilizzare l'MACD è quello di comprare il titolo quando la media mobile corta è maggiore di quella lunga, e viceversa.

Inoltre, il MACD viene anche utilizzato come oscillatore per identificare i punti di ipercomprato e ipervenduto. Quando la linea MACD si trova al di sopra del suo valore medio, si ritiene che il mercato sia ipercomprato, mentre quando si trova al di sotto del suo valore medio, si ritiene che il mercato sia ipervenduto. Queste informazioni possono essere utilizzate dai trader algoritmici per prendere decisioni di acquisto o vendita.

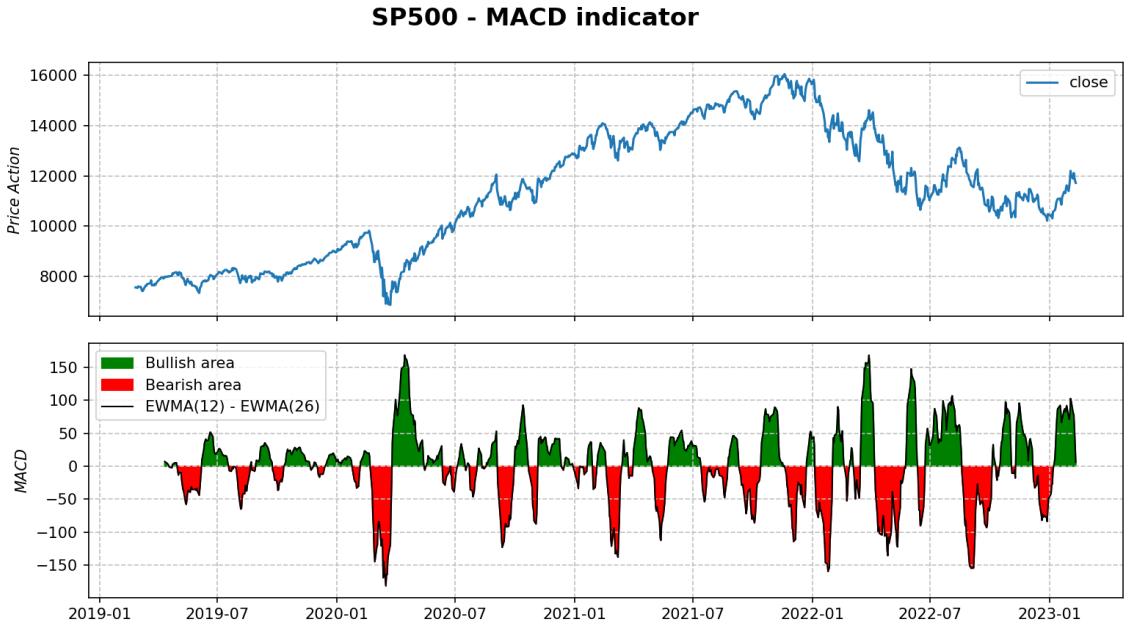


Figure 12: MACD dell'SP500

2.2.2.2 Relative Strength Index (RSI)

L'RSI (*Relative Strength Index*) è un indicatore tecnico molto comune, utilizzato nel trading algoritmico e finanziario. Il suo scopo è quello di misurare la forza relativa di una tendenza e identificare i punti di ipercomprato e ipervenduto.

L'RSI è calcolato come rapporto tra le barre che hanno chiuso in rialzo e quelle che hanno chiuso in ribasso, durante un determinato periodo di tempo. Il valore dell'RSI viene, solitamente, rappresentato in un range che va da 0 a 100. Un valore superiore a 70 indica che il mercato è ipercomprato, mentre un valore inferiore a 30 indica che il mercato è ipervenduto.

Il funzionamento dell'RSI può essere utilizzato per identificare i punti di inversione del mercato, ad esempio quando il mercato si trova in una tendenza al rialzo ma il valore dell'RSI inizia a scendere verso il suo valore medio, si può prevedere una potenziale inversione di tendenza.

Inoltre, l'RSI può essere utilizzato per confermare i segnali di altri indicatori tecnici. Ad esempio, se un trader algoritmico individua una potenziale inversione di tendenza utilizzando un altro indicatore tecnico e questo segnale viene confermato da un valore basso dell'RSI, il trader può essere maggiormente incline a prendere una decisione di acquisto.

La formulazione è la seguente:

$$RSI_t(k) = 100 - \frac{100}{1 + RS_t}$$

$$RS_t(k) = \frac{MA_t^{up}(X)}{MA_t^{down}(X)},$$

dove:

- X sono le k differenze dei prezzi (i.e. i prezzi integrati una volta)
- $MA_{t,k}^{up}$ e $MA_{t,k}^{down}$ sono medie mobili (di qualsiasi tipologia) calcolare al periodo t, che contengono le differenze nei prezzi che sono, rispettivamente, positive e negative.
- $RS_t(k)$ è il Relative Strength al periodo t costruito su una finestra lunga k

All'interno del modello implementato, l'RSI è stato normalizzato tra 0 e 1, utilizzando la seguente formula:

$$RSI_{new} = \left(\frac{RSI_{original}}{100} - 0.5\right) * 2$$

Il classico utilizzo dell'RSI è quello di identificare l'area di ipervenduto quando l'indicatore è inferiore a 30 (-0.4 per RSI_{new}) e di ipercomprato quando l'indicatore è maggiore di 70 (0.4 per RSI_{new}). Pertanto, quando l'RSI entrerà nell'area di ipervenduto e poi ritornerà in un'area "normale" (al di sopra di 30 o -0.4 per RSI_{new}) si andrà *long*; ragionamento analogo per l'ipercomprato.

Di seguito si riporta l'RSI (14) per l'SP500.

Figure 13: RSI(14) dell'SP500

2.2.2.3 Bande di Bollinger

Le bande di Bollinger sono uno strumento utilizzato in analisi tecnica per determinare la volatilità di un'azione o di un indice. Esse sono calcolate come N deviazioni standard intorno alla media mobile di una serie di prezzi. La media mobile viene calcolata come la media dei prezzi per un determinato periodo di tempo, ad esempio 20 giorni.

Le bande di Bollinger sono costituite da tre linee: una media mobile, una banda superiore e una banda inferiore. La banda superiore viene calcolata come media mobile più N deviazioni standard, mentre la banda inferiore come media mobile meno N deviazioni standard. La deviazione standard viene calcolata come la deviazione standard dei prezzi per un determinato periodo di tempo ed ha la seguente formulazione:

$$\sigma_t = \frac{1}{T}\sum_{t=1}^{T}(x_i - \mathbb{E}[X])^2,$$

dove x_i, nel caso delle bande di Bollinger, saranno i prezzi, mentre $\mathbb{E}[X]$ è la media dei prezzi nel periodo che va da $t = 1$ fino a $t = T$.

In genere, se i prezzi si avvicinano alla banda superiore, si considera un segnale di ipercomprato, il che significa che il prezzo potrebbe essere sovrastimato e potrebbe essere previsto un ribasso. D'altra parte, se i prezzi si avvicinano alla banda inferiore, si considera un segnale di ipervenduto, il che significa che il prezzo potrebbe essere sottostimato e potrebbe essere previsto un rialzo. Risulta quindi un indicatore molto utile per segnalare dei *paths* di *Mean-reverting* del prezzo.

Ad esempio, si potrebbe implementare una strategia algoritmica simile a quella dell'RSI:

- Vendere ogni qualvolta il prezzo "rompe" la banda superiore e "rientra" nel canale[30]
- Comprare ogni qualvolta il prezzo "rompe" la banda inferiore e "rientra" nel canale
- Chiudere la posizione long/short ogni qualvolta il prezzo "attraversa" la media mobile nel verso opposto alla banda che ha generato il *trading signal*

Di seguito si riportano le bande di Bollinger applicate all'SP500.

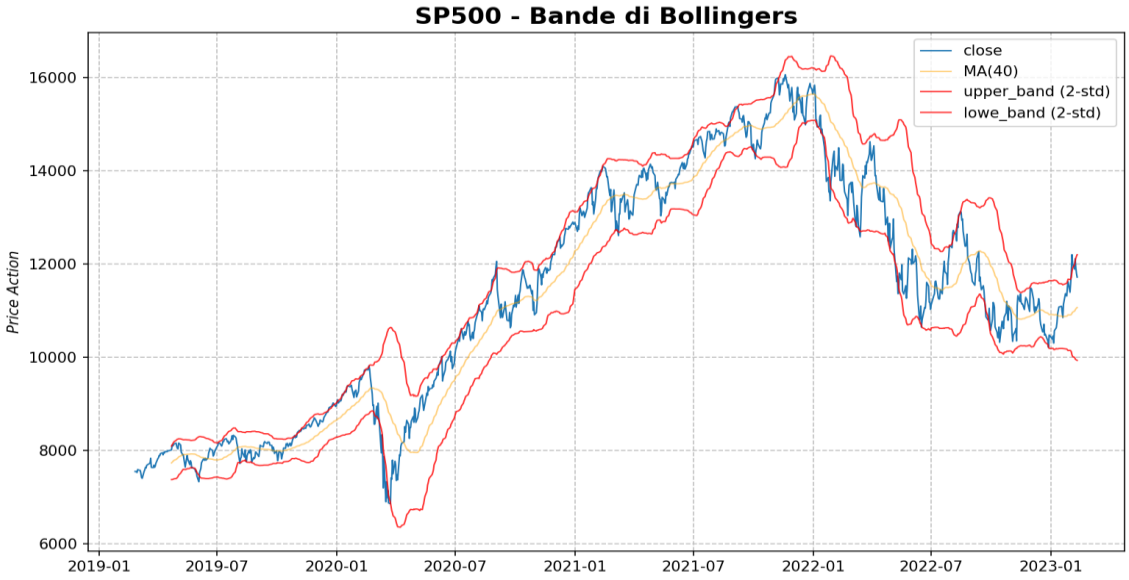


Figure 14: Bande di Bollinger dell'SP500

[30] Per "canale" si intende il *bound* che viene costruito dalla banda superiore e da quella inferiore. Risulta molto evidente guardando il grafico.

2.2.2.4 Momentum - ROC

Il *Momentum* è uno degli indicatori più semplici, ma allo stesso tempo viene ampiamente utilizzato sotto diverse forme. Esso misura la velocità del cambiamento dei prezzi. Si definisce come la differenza fra il prezzo attuale p_t e il prezzo a N timestep nel passato p_{t-n}. Calcolato come $M_t = p_t - p_{t-n}$, esso può assumere valori positivi nel caso in cui il prezzo istantaneo sia maggiore di quello a N periodi nel passato, negativi altrimenti. Valori estremamente alti o estremamente bassi del momentum indicano che un titolo, in quel momento, è in ipervenduto o in ipercomprato.

ROC (Rate of Change)

Il ROC è un indicatore tecnico del *Momentum* che misura la percentuale di cambio di prezzo fra il prezzo al timestep t (prezzo attuale) e il prezzo a N periodi nel passato, i.e. prezzo al timestep $t - n$.

Esso viene calcolato come:

$$ROC_t(N) = 100 * \frac{p_t - p_{t-n}}{p_{t-n}}$$

Intuitivamente, esso rappresenta la forza del *Momentum* del titolo, poiché indica la *pendenza* nel periodo di tempo preso in considerazione. Valori positivi di questo oscillatore indicano una tendenza positiva del mercato, invitando il trader ad aumentare la propria posizione investendo in buy. Valori negativi, al contrario, indicano una discesa del titolo e indicano al trader di andare in sell. Inoltre, come risulta evidente dal ROC calcolato sull'SP500, sono individuabili dei *cluster* di volatilità nell'indicatore che corrispondo a periodi di "turbolenza" nel mercato.

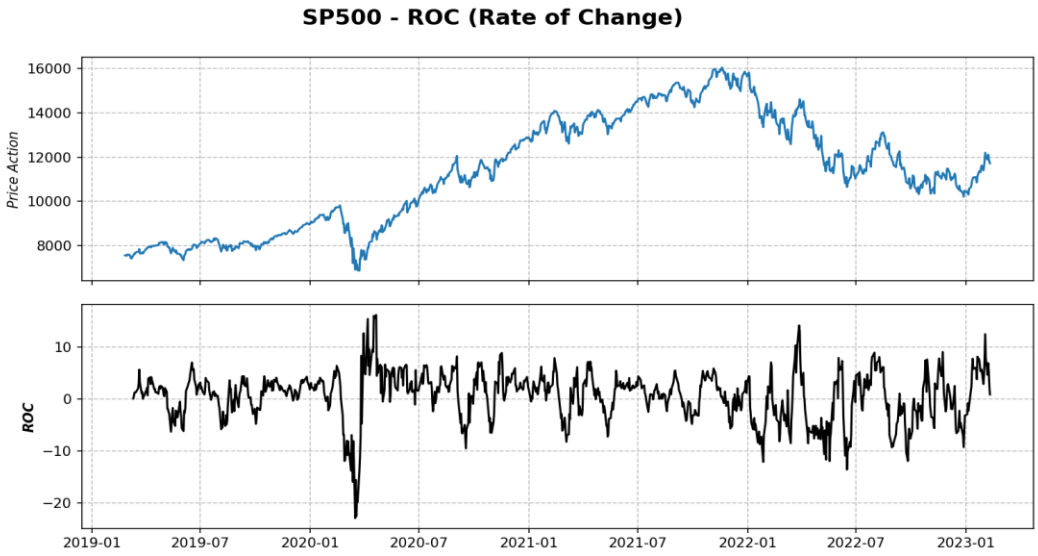


Figure 15: ROC dell'SP500

2.2.2.5 Sigma Events (*Dummy*)

Il *Sigma Events* non è un indicatore di analisi tecnica, bensì è un indicatore che è stato costruito all'interno di questa tesi e che, talvolta, viene utilizzato per nutrire dei modelli di trading algoritmico. La sua costruzione risulta abbastanza semplice:

- Si calcola la volatilità dei prezzi con una finestra K
- Ogni qual volta il movimento dei prezzi ($p_t - p_{t-1}$) risulta maggiore di N volatilità, si avrà un sigma event (1 nella serie dummy output).

Si potrebbe utilizzare questa serie quando, per esempio, si vogliono individuare situazioni di *over-reaction* da parte del mercato. Pensiamo all'annuncio, da parte di una Banca Centrale, di un rialzo inaspettato dei tassi ufficiali, accompagnato da un discorso *hawkish*[31] da parte del presidente della suddetta Banca Centrale. Nel mercato Forex, sarebbe immediata un'inondazione di ordini di acquisto sulla valuta. Solitamente però, parte di questo repentino rialzo tende a riassorbirsi, poiché esso viene anche scatenato da una mancanza di liquidità dal lato dell'Ask.

Di seguito si riportano i 3-Sigma events relativi all'SP500. In particolare, la *dummy* è stata costruita assegnandogli un valore pari ad 1, ogni qual volta vi è un sigma-event al rialzo, ed un valore pari a -1, ogni qual volta vi è un sigma-event al ribasso.

[31] Per Hawkish si intende una politica monetaria restrittiva

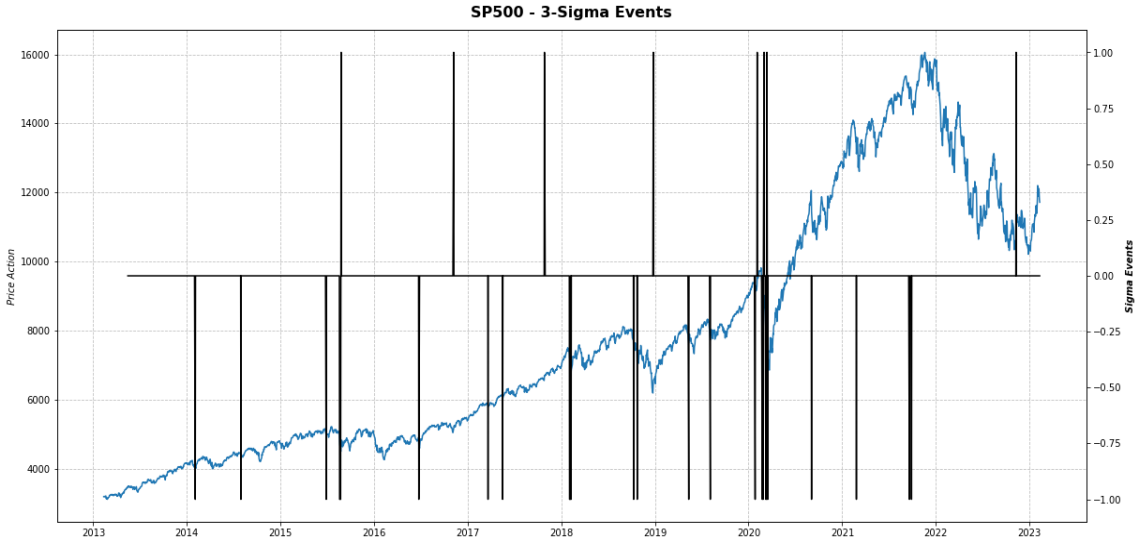


Figure 16: 3-Sigma Events dell'SP500

2.2.3 Standardizzazione e Normalizzazione

Nel mondo del machine learning assume un ruolo fondamentale la trasformazione dei dati utilizzati come *input feature*. Accade spesso infatti che essi siano totalmente differenti dal punto di vista della distribuzione o dell'ordine di grandezza, causando forti squilibri nella fase di apprendimento. Risulta necessaria e cruciale dunque una fase di *preprocessing*, per fare in modo che essi possano essere interpretati in modo opportuno dall'agente e che la conoscenza ricavata sia maggiormente rilevante. In particolare nelle Neural Networks, "scalare" i dati è importante per fare in modo di minimizzare il *bias* tra le *features* e, di conseguenza, ridurre drasticamente il tempo di training dell'algoritmo.

I classici metodi di standardizzazione prevedono la necessità di poter stimare i valori massimi e minimi del dataset, oppure di calcolare funzioni su tutto il dataset come deviazione standard, varianza o media. Questo risulta possibile nella situazione in cui le predizioni sono necessarie solamente *offline* e le scale di valori dei dati e la loro distribuzione sono ben definite. In questo modo è possibile calcolare tutte le metriche senza nessun tipo di problema, a vantaggio del training dell'algoritmo. Ciò non risulta affatto possibile nel momento in cui ci si trova in presenza di dati sconosciuti oppure i cui valori minimi e massimi non sono stabiliti a priori, come nel caso delle serie temporali in ambito finanziario. Il motivo principale di questa incompatibilità sta nel fatto che le predizioni devono essere effettuate su dati mai visti e in *real-time*, che dunque potrebbero eccedere i livelli di minimo e massimo dei dati conosciuti a priori. Per questo motivo è necessario

utilizzare delle tecniche di normalizzazione che non implichino conoscenza a priori di dati futuri nella fase di training. Esistono varie metodologie per raggiungere il risultato desiderato. Si può definire, ad esempio, una conoscenza a priori del dominio di applicazione e stabilire massimi e minimi a seconda di quest'ultimo. In questo caso si utilizza quindi una conoscenza intrinseca non dovuta necessariamente ai dati in quel momento, ma a una consapevolezza più generale del campo di applicazione.

Data la natura dei dati e del problema che si affronta nella presente tesi, si è deciso di utilizzare la tecnica di standardizzazione più utilizzata in questi ambiti: *la Z-Score Normalization*.

2.2.3.1 Z-Score Normalization

La normalizzazione Z-Score è una tecnica di normalizzazione utilizzata per trasformare i dati in modo che abbiano una distribuzione normale standard con una media di 0 e una deviazione standard di 1. Questa tecnica viene utilizzata in molte aree della scienza dei dati, inclusa l'analisi dei dati, la statistica e il machine learning, per normalizzare i dati e renderli più facilmente gestibili e interpretabili per i modelli.

$$a' = \frac{a - \mu(A)}{\sigma(A)}, \qquad dove\ \mu(A) = \frac{\sum_{i=1}^{N} a_i}{N};\ \ \sigma(A) = \sqrt{\frac{\sum_{i=1}^{N}\left(a_i - \mu(A)\right)^2}{N}}$$

Questo metodo risulta molto utile nel caso di dati stazionari di cui non si conoscono i valori massimi e minimi, ma di cui si può fare una stima della media e dello scarto quadratico medio. Il problema è che, anche in questo caso, si suppone una conoscenza a priori dei dati poiché nelle *timeseries* la media e la deviazione standard variano nel tempo.

2.3 Architettura del modello

Il presente paragrafo verrà invece dedicato ad una descrizione esaustiva dell'architettura che si è andata a costruire per la parte sperimentale della tesi.

Il modello di Reinforcement Learning che si è deciso di utilizzare è quello del *Double-Deep-Q-Network* (DDQN), la cui architettura teorica è stata ampiamente descritta nel Capitolo 1 "*Framework Teorico*".

L'approccio che si è deciso di utilizzare per la costruzione dell'architettura del modello segue una logica OOP (*Object-Oriented-Programming*), ovvero una logica di

programmazione che organizza un'architettura per oggetti (in questo caso classi Python), piuttosto che per funzioni o logica.

I 4 oggetti principali che compongono l'architettura sono:

- **Data Source**
- **Trading Simulator**
- **Trading Environment**
- **DDQN Agent**

Ognuno di questi oggetti svolge un ruolo specifico e, o viene integrato in un altro oggetto (chiamato figlio), oppure comunica con questo. Nello specifico, il Trading Environment avrà integrato sia il Data Source che il Trading Simulator, mentre Trading Environment e DDQN-Agent comunicheranno senza avere integrazioni reciproche.

Pertanto, il Data Source avrà il ruolo di fornire dati di prezzo/rendimenti e *features* ingegnerizzate; il Trading Simulator fornirà *feedbacks* delle azioni (i.e. *rewards*) intraprese dall'agente e memorizzerà l'andamento del portafoglio; il Trading Environment sarà l'unico ad interagire con l'Agente, fornendogli gli stati (i.e. dati e *features*), ricevendo da questo le azioni prescelte dato un determinato *set* di *features* e gli restituirà la *reward,* la quale verrà utilizzata per migliorare la *policy* dell'Agente (anche attraverso il meccanismo del *memory replay*).

Di seguito si riporta uno schema che riassume, sinteticamente, il processo generato dall'Architettura.

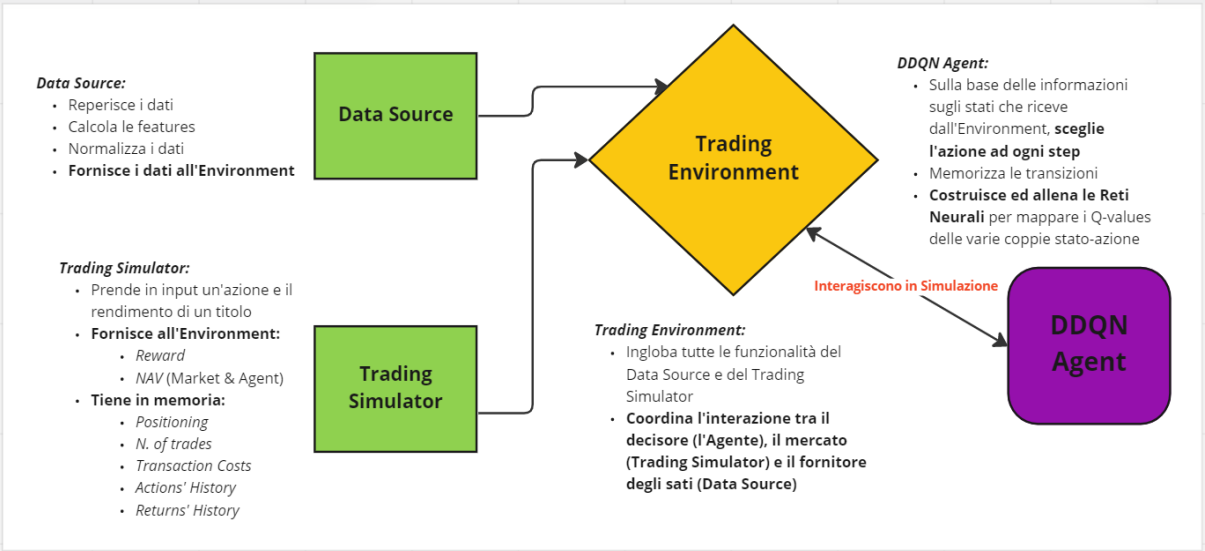


Figure 17: Architettura del modello di RL implementato nella tesi

L’allenamento dell’Agente avverrà, come da prassi per i modelli di RL, suddividendo il dataset di training in episodi. Pertanto, ad ogni episodio, verrà azzerata la memoria del Trading Simulator e del Trading Environment, ma non quella dell’Agente relativa al *memory replay* e ai pesi delle Reti Neurali. Ogni episodio è composto da N steps, ad ognuno di questi steps avverrà l’interazione sopra descritta.

Si rimanda ai successivi sotto-paragrafi per una descrizione approfondita dei singoli oggetti che compongono l’architettura, inoltre, si rimanda all’appendice “A. Codice Sorgente” per le restanti parti di codice.

2.3.1 Tecnologia e Librerie

Per lo sviluppo dell’architettura si è deciso di utilizzare Python 3.9 come linguaggio di programmazione e, in particolare, si è utilizzato PyCharm 2021.1.1 x64 come IDE.

Le librerie che vengono richiamate all’interno del Progetto sono le seguenti:

- **Pandas**
    - E’ una libreria *open-source* per la manipolazione e l'analisi dei dati in Python. Viene utilizzata comunemente in molte aree della *Data Science*, come la preparazione dei dati, la pulizia dei dati, la manipolazione dei dati, l'analisi statistica e la visualizzazione dei dati. Pandas introduce una struttura dati chiamata DataFrame che consente di rappresentare i dati in forma di tabelle e di eseguire operazioni di manipolazione su di essi in modo semplice e intuitivo. Ad oggi, risulta essere uno standard affermato nel mondo della Data Science.
- **Numpy**
    - E’ una libreria *open-source* per il calcolo scientifico in Python. È stato sviluppato per fornire un'efficiente implementazione di array multidimensionali e funzioni matematiche avanzate, che sono le basi per molte applicazioni di *Data Science* e di machine learning. Con NumPy, è possibile eseguire operazioni complesse su grandi quantità di dati in modo molto efficiente. Inoltre, NumPy offre un'interfaccia integrata con molte altre librerie popolari per la *Data Science*, come scikit-learn, matplotlib e pandas,

rendendolo un componente chiave per molte attività di scienza dei dati e machine learning.

- **Matplolib**
    - Matplotlib è una libreria *open-source* per la creazione di grafici e visualizzazioni di dati in Python. La libreria fornisce una vasta gamma di funzionalità per la creazione di grafici di diverse tipologie, tra cui linee, barre, istogrammi, scatter plot, grafici a torta e molto altro. Matplotlib è molto flessibile e permette agli utenti di personalizzare i grafici in molti modi, ad esempio modificando i colori, i tipi di linea, le etichette e le leggende.
- **TensorFlow**
    - È una libreria *open source* di machine learning sviluppata da Google. È una delle più popolari librerie di machine learning e viene utilizzata per sviluppare e formare modelli di Deep Learning, che sono una delle forme più avanzate di apprendimento automatico. TensorFlow è stato progettato per essere altamente scalabile e adattabile. TensorFlow supporta una vasta gamma di architetture di Deep Learning, tra cui reti neurali *feedforward*, reti convolutional neurali (CNN) e reti recurrenti (RNN).
- **OpenAI Gym**
    - È un'ambiente di simulazione per la ricerca e lo sviluppo di algoritmi di apprendimento automatico. È stato sviluppato da OpenAI ed è stato progettato per fornire un ambiente standardizzato per la valutazione e la comparazione di algoritmi di apprendimento automatico. Con Gym (così è come la libreria viene chiamata all'interno di Python), gli sviluppatori possono definire e creare ambienti di simulazione che rappresentano problemi di apprendimento automatico specifici.
- **Scikit-Learn**
    - Scikit-learn (Sklearn) è una libreria Python molto popolare e molto potente per l'analisi dei dati e per il machine learning. Fornisce una vasta gamma di algoritmi di machine learning, tra cui regressione, classificazione, *clustering*, *dimensionality reduction*, *model selection* e *pre-processing* dei dati. Uno dei punti di forza di Sklearn è la sua facile integrazione con altre librerie scientifiche Python, come NumPy e Pandas. Inoltre, Sklearn ha un'interfaccia coerente e semplice da usare, che rende facile l'apprendimento

e l'utilizzo degli algoritmi di machine learning. Questa libreria viene ampliamente utilizzata in molte industrie, tra cui la finanza, la salute, la tecnologia e la *Data Science*.

- **Ta-Lib**
    - TA-Lib (Technical Analysis Library) è una libreria *open-source* per il software di analisi tecnica. È scritta in C++ ed è disponibile per molte piattaforme, tra cui Windows, Linux e macOS. TA-Lib fornisce una vasta gamma di funzioni per l'analisi tecnica dei mercati finanziari, come la media mobile, l'indicatore di momentum, le bande di Bollinger e molte altre. Queste funzioni possono essere utilizzate per identificare *pattern* e tendenze nel prezzo di un asset, e prendere decisioni di trading informate. TA-Lib è molto popolare tra gli sviluppatori di software finanziario e viene utilizzato in molte piattaforme di trading algoritmico e in molte applicazioni *desktop* e *web*. Uno dei punti di forza di TA-Lib è la sua alta affidabilità e la vasta documentazione, che lo rende facile da usare per gli sviluppatori di *software* e per gli utenti esperti di analisi tecnica.
- **YFinance**
    - È una libreria che integra l'API fornita da Yahoo-Finance. Risulta una libreria molto comoda e di facile utilizzo per ottenere vasta gamma di serie storiche finanziare, seppure mancando un po' in termini di qualità dei dati. Ad esempio, anche la libreria Pandas, nella sua estensione pandas-datareader, integra l'API di Yahoo-Finance

2.3.2 Data Source

Il Data Source avrà il ruolo di fornire dati e *features* (i.e. stati) all'Environment, il quale le servirà all'Agente che prenderà delle azioni in base alla *policy*. All'inizio di ogni episodio, selezionerà randomicamente gli stati da fornire all'Environment, questo anche al fine di evitare che l'algoritmo si addestri legandosi ad una certa dipendenza di sequenzialità tra le esperienze. Inoltre, avrà anche il ruolo di tenere memoria degli *steps* relativi al singolo episodio, *steps* che verranno re-impostati a 0 ad ogni nuovo episodio.

I files relativi ai dati e alle *features* sono memorizzati in due files parquet[32], al fine di facilitarne la lettura per il Data Source. A differenza dell'approccio tradizionale, per cui le *features* vengono calcolate quando se ne ha necessità, all'interno di questo progetto vi è un *folder* dedicato alla costruzione, pertanto, queste vengono calcolate in modo indipendente dal Data Source e questo dovrà andare unicamente a leggere il file che le contiene.

In conclusione, quando il Data Source viene istanziato all'inizio della simulazione, dovrà leggere dati di prezzo (sui quali calcolerà i rendimenti) e *features*, normalizzare queste ultime e andare a fornire gli stati all'Environment durante i vari *steps* degli episodi.

2.3.3 Trading Simulator

Il Trading Simulator avrà il ruolo di tenere traccia della performance del trading e restituirà all'Agente (passando per l'Environment) la *reward* risultante da un'azione presa. Pertanto, è proprio all'interno di questo oggetto che si andrà a definire il premio (punizione) che l'Agente riceverà come risultante delle proprie azioni. Si è deciso di utilizzare un approccio standard, utilizzando il rendimento ottenuto tra uno step e l'altro, nettato dei costi di transazione, come *reward* per l'Agente.

Inoltre, il Trading Simulator, durante ogni episodio, terrà memoria di diverse serie temporali, tra cui: le azioni prese dall'agente, il NAV[33] dell'Agente e del mercato, i rendimenti della strategia, le posizioni prese sul titolo da parte dell'Agente, i *transaction costs* pagati, il numero di *trades* effettuati, i rendimenti del titolo oggetto di trading.

A richiesta, fornirà le serie di cui sopra. Queste verranno utilizzate al termine di ogni episodio per calcolare delle metriche di performance, al fine di valutare sia la bontà del trading svolto dall'Agente, sia la progressione nell'apprendimento dell'Agente.

2.3.4 Trading Environment

Il Trading Environment è effettivamente il "punto di snodo" principale dell'architettura che è stata costruita. Questo integrerà al suo interno sia il Data Source, responsabile della

[32] Parquet è un formato di file *columnar*, utilizzato per l'archiviazione di grandi quantità di dati strutturati in modo efficiente.
[33] In questo contesto, per NAV (Net Asset Value) s'intende il valore di 1€ investito nella strategia seguita dall'Agente. Ha l'utilità di fornire una serie temporale da poter analizzare per valutare la bontà della strategia di trading utilizzata dall'Agente.

fornitura dei dati e della selezione randomica degli stati per simulare gli episodi, sia il Trading Simulator, responsabile della valutazione del trading svolto dall'agente.

Essa consiste in un'estensione dell'interfaccia messa a disposizione da OpenAI Gym adattata al trading, utilizzabile tramite funzioni di *callback* da parte dell'agente per ricevere le osservazioni dello stato attuale e la reward associata alle azioni scelte. L'Environment sarà quindi l'unico che interagirà direttamente con l'Agente, fornendogli gli stati e le rewards ad ogni step di ogni episodio.

La configurazione dell'Environment avviene prima di iniziare la simulazione e richiede:

- Il nome del titolo sul quale l'Agente andrà a fare trading. Questo si rende necessario al fine di indicare al Data Source i nomi dei files dal quale prendere dati e *features*.
- Start/End, è un parametro che ha lo scopo di permettere all'utente di selezionare una porzione specifica dei dati che si stanno utilizzando. Risulta molto comoda quando si vuole utilizzare una parte dei dati per effettuare il training dell'Agente e lasciare un'altra porzione inesplorata per testare l'algoritmo.
- Il numero di steps per episodi, al fine di calibrare il training dell'Agente. Questa è una scelta che viene discrezionalmente presa dal ricercatore e varierà al variare del *timeframe*[34] dei dati che si utilizzano.
- Trading Costs, al fine di fornire al Trading Simulator i costi relativi alla compravendita del titolo.

2.3.5 DDQN-Agent

L'Agente risulta essere, ovviamente, il cuore dell'architettura e del modello che è stato costruito. Nei successivi sotto-paragrafi, si andranno a descrivere più nel dettaglio le componenti di cui si compone. Esso costruirà due Reti Neurali (Online e Target) composte da 3 *layers*, che saranno allenate durante tutto il training. Inoltre, avrà al proprio interno un *buffer*[35] di memoria che verrà utilizzato per memorizzare le transizioni (stato, azione,

[34] Per *timeframe* si intende la finestra di tempo specifica che viene utilizzata per definire un singolo punto di dati all'interno di una serie temporale di prezzi di un asset finanziario. Ad esempio, un *timeframe* di un'ora significa che un singolo dato conterrà informazioni relative ad un'intera ora.

[35] In informatica, un *buffer* di memoria è un'area di memoria che viene utilizzata come area temporanea per la memorizzazione di dati in fase di elaborazione. Viene quindi utilizzata come area "di transito" per dei dati che vengono trasferiti da un componente del sistema ad un altro.

reward, stato successivo) e, successivamente, applicare la metodologia dell'*experience replay* per allenare le Reti Neurali.

2.3.5.1 ε-greedy Policy

Un metodo[36] sarà dedicato alla *policy* e sarà responsabile di selezionare l'azione, dato un array di uno stato. Utilizzando una Epsilon-greedy *policy*, ampiamente discussa nel capitolo 1 "*Framework Teorico*", l'Agente con probabilità $1 - \varepsilon$ andrà a selezionare l'azione che massimizzerà il Q-value e, con probabilità pari ad ε, andrà a scegliere un'azione in modo randomico. Si è scelta una ε-greedy *policy* al fine di mediare il *trade-off* tra exploration e exploitation, descritto nel capitolo 1. Si riporta di seguito il codice relativo all'implementazione della *policy*

```
def epsilon_greedy_policy(self, state: np.ndarray) -> int:
    self.total_steps += 1
    if np.random.rand() <= self.epsilon:
        return np.random.choice(self.num_actions)
    else:
        q = self.online_network.predict(state, verbose=False)
        return np.argmax(q, axis=1).squeeze()
```

2.3.5.2 Rete Neurale

Il metodo responsabile della costruzione della Rete Neurale permette di definire la composizione di questa in modo flessibile, in termini di numero e dimensione dei *layers*, quindi di profondità della Rete Neurale stessa.

Il numero di *inputs* del primo *layer* saranno, ovviamente, pari al numero di elementi che compongono uno stato. In merito alla funzione di attivazione dei neuroni, dopo diverse prove volte a valutare la performance della Rete, si è deciso di scegliere la ReLu, ampiamente descritta nel Capitolo 1 "*Framework Teorico*". Si è inoltre deciso di applicare un fattore di penalizzazione alla rete, gestito dal parametro `kernel_regularizer`, al fine di prevenire il sovrapprendimento, che si verifica quando una rete neurale ha un'alta accuratezza sui dati di addestramento ma bassa accuratezza sui dati di test.

Una volta costruiti i *layers* che compongono la rete, viene aggiunto un altro livello attraverso la funzione Dropout di Tensorflow. Il Dropout è una tecnica di regolarizzazione che viene

[36] In Python un metodo altro non è che una funzione di una classe.

utilizzata in molte reti neurali profonde. Il suo scopo principale è prevenire l'*overfitting*, ovvero evitare che la rete sia troppo adattata al training set e di conseguenza non possa generalizzare bene con nuovi dati. Durante l'addestramento, una certa percentuale di neuroni viene casualmente ignorata, ovvero vengono "scartati". Questo crea una forma di diversità nella rete, impedendo ai neuroni di diventare troppo specializzati.

Viene infine definito un ultimo *layer* che fornirà l'output e, nel caso d'uso specifico del modello che si è costruito, gli outputs corrisponderanno al numero di azioni tra le quali l'Agente può scegliere.

Al termine della definizione dei vari *layers* e del Dropout, questi vengono inseriti all'interno di una Rete Neurale c.d. *feedforward* (i.e. interamente connessa) e viene "compilata" la rete. La funzione di perdita che viene utilizzata per calcolare gli errori di stima della Rete, quindi che viene usata per allenare la stessa, è il classico scarto quadratico medio. Questo ha il vantaggio di penalizzare in modo maggiore quegli errori che presentano una magnitudo più ampia.

Particolare attenzione va dedicata al parametro `trainable`, che viene inserito all'interno dei vari Dense (i.e. dei *layers*). Questo parametro accetta variabili di tipo booleano (i.e. *True*/*False*) e serve ad impostare se la Rete Neurale sarà allenabile o meno. Risulta particolarmente utile nel caso d'uso del DDQN, dove si ha la Rete Target che non verrà mai allenata, ma subirà unicamente un aggiornamento dei propri pesi ogni *K* steps, ed una Rete Online che invece verrà allenata all'interno dell'*experience replay*.

Di seguito si riporta il codice relativo all'implementazione della Rete Neurale appena descritta.

```
def build_model(self, trainable: bool = True):
    """
    Build an ANN model
    - number of layers is defined by self.architecture

    :param trainable: True for the online ANN and False for the target one
    :return: ANN model
    """
    layers = []
    # n = len(self.architecture)
    for i, units in enumerate(self.architecture, 1):
        layers.append(
            tf.keras.layers.Dense(
                units=units,
                input_dim=self.state_dim if i == 1 else None,
                activation='relu',
                kernel_regularizer=tf.keras.regularizers.l2(self.l2_reg),
```

```
            name=f'Dense_{i}',
            trainable=trainable
        )
    )
layers.append(tf.keras.layers.Dropout(.1))
layers.append(
    tf.keras.layers.Dense(
        units=self.num_actions,
        trainable=trainable,
        name='Output'
    )
)
model = tf.keras.models.Sequential(layers)
model.compile(
    loss='mean_squared_error',
    optimizer=tf.keras.optimizers.Adam(learning_rate=self.learning_rate)
)
return model
```

2.3.5.3 Experience Replay & Double Q-Learning

Molto importante risulta essere il metodo che implementa l'*experience replay* per l'allenamento delle due Reti Neurali, sia perché permette all'Agente di apprendere in modo indipendente dalla dipendenza temporale e dalla sequenzialità dei dati, sia perché, proprio all'interno di questa funzione, avviene l'utilizzo del Double Q-Learning.

Per il *buffer* in memoria che deve contenere le esperienze dell'Agente viene utilizzata una c.d. *deque*, ovvero una struttura dati molto simile ad una lista, contenuta nella libreria *built-in* di Python chiamata *collections*, che permette di effettuare inserimenti e rimozioni di elementi sia dalla parte iniziale che dalla parte finale della coda in modo molto efficiente. Tali caratteristiche, fanno sì che esso risulti essere l'oggetto adatto per un approccio di tipo FIFO[37], tale per cui si fissa una dimensione massima in termini di esperienze memorizzabili e, all'inserirsi di nuove tuple all'interno della *deque*, si vanno ed eliminare le più vecchie.

Come risulta evidente dal codice sorgente, il meccanismo dell'*experience replay* si attiva solo qualora si disponga di un numero di esperienze memorizzate nella *deque* maggiore della dimensione minima richiesta per avere un *batch*[38] adeguato ad allenare le Reti. Infatti, tale vincolo farà sì che i primi episodi non genereranno nessun training delle Reti.

[37] *First In First Out*

[38] In informatica e nel Machine Learning, con il termine "*batch*" ci si riferisce ad un insieme di dati che vengono elaborati insieme invece che singolarmente. Infatti, anche in questo caso, con il termini *batch* ci si riferisce ad un insieme di esperienze passate dell'Agente, che vengono utilizzate per allenare le Reti.

Una volta che si dispone di un *batch* di esperienze adeguatamente consistente, si andrà ad utilizzare un approccio vettorizzato[39] per allenare le reti. Tale scelta è motivata da una necessità di efficientamento del codice in termini di tempo computazionale poiché, è proprio in questa funzione che la simulazione impiega più tempo. Si parlerà del *computational burden* nel 2.4, paragrafo "*Computational Burden*".

Il primo punto chiave che merita di essere dischiuso è come viene composto il *batch* di esperienze che si andrà a fornire alle Reti per l'allenamento. Verranno prese tuple di esperienze (i.e. stato, azione, reward, stato successivo) in modo completamente randomico dalla *deque* (self.experience nella funzione) che, ovviamente, conterrà molte più esperienze rispetto alla dimensione richiesta dal *batch* per avviare il training.

Tralasciando i passaggi di natura squisitamente tecnica che manipolano le strutture dati per fornire in modo "comodo" l'input per il training, si può apprezzare l'utilizzo dell'approccio di Double Q-Learning, osservando come vengono stimati, utilizzando la funzione `predict_on_batch()`, due vettori di Q-Values: `next_q_values` e `next_q_values_target`. Il primo sarà un vettore di Q-values stimati utilizzando l'Online ANN, mentre il secondo è un vettore con la stima dei medesimi Q-Values ma effettuata con la Target ANN. La doppia stima dei Q-Values è ciò che permette di aggiornare i valori Q utilizzando il Double Q-Learning, approfonditamente discusso nel Capitolo 1 "*Framework Teorico*", ed è visibile quando si crea la variabile `targets` e successivamente la si inserisce all'interno dei Q-values che vengono utilizzati dalla ε-greedy *policy* per selezionare l'azione migliore per l'Agente.

Una volta aggiornati i Q-values con il meccanismo del Double Q-Learning, si andrà ad allenare la Rete c.d. Online con il *batch* di esperienze. Per la Rete Target invece, si possono apprezzare le ultime due righe della funzione che fanno sì di aggiornare la Target ogni τ (*tau*) steps. La funzione `update_target()` non fa altro che andare a prendere i pesi stimati dalla Rete Online e assegnarli alla Rete Target.

```
def experience_replay(self) -> NoReturn:
    """
    It trains ANN (both target and online) using the experience replay approach
    """
    # Make experience replay approach only if the stored experienced has at least self.batch_size elements
    if self.batch_size > len(self.experience):
```

[39] Scegliere un approccio c.d. "vettorizzato", significa effettuare calcoli direttamente su vettori e/o matrici piuttosto che su di essi all'interno di una routine, ovvero di un for loop. Ad esempio, in questa funzione si è preferito vettorizzare l'allenamento delle reti, piuttosto che inserire un for loop che iterava su tutte le tuple (stato, azione, reward, stato successivo).

```
        pass
    else:
        # ----- It uses a vectorize approach, instead of a for loop ----------------------------------------------

        # create a minibatch, taking N (self.batch_size) experiences (from self.experience). Using
        # random.sample will return a list of list, then zipping them we will have a list of 5 tuple where each
        # tuple will be a "category" (state, action, ...). Then mapping them with function np.array, we will have
        # the input for the model
        minibatch = map(np.array, zip(*random.sample(self.experience, self.batch_size)))
        states, actions, rewards, next_states, not_done = minibatch

        # Q-value (t + 1) with ONLINE-ANN
        next_q_values = self.online_network.predict_on_batch(next_states)
        best_actions = tf.argmax(next_q_values, axis=1)  # best action (i.e. greedy-online) for t+1

        # Q-value (t + 1) with TARGET-ANN
        next_q_values_target = self.target_network.predict_on_batch(next_states)
        # It's a way to choice between online-ANN choice and target-ANN choice. i.e. Double-DQN approach
        target_q_values = tf.gather_nd(
            next_q_values_target,
            # tf.stack: stacks a list of rank-R tensors into one rank-(R+1) tensor
            tf.stack(
                (
                    self.idx,
                    tf.cast(best_actions, tf.int32)  # tf.cast convert a type to another one (here to int32)
                ),
                axis=1
            )
        )
        # update q-value (t) using TARGET or ONLINE ANN. This is where we apply the DOUBLE-DQN approach
        targets = rewards + not_done * self.gamma * target_q_values

        q_values = self.online_network.predict_on_batch(states)  # current Q-value (t)
        q_values[self.idx, actions] = targets  # update q-values

        loss = self.online_network.train_on_batch(x=states, y=q_values)
        self.losses.append(loss)

        # update TARGET-ANN every self.tau steps. Better understand the code
        if self.total_steps % self.tau == 0:
            self.update_target()
```

2.3.6 main_train.py

Il file chiamato main.py è effettivamente il modulo python in cui avviene la simulazione e l'allenamento dell'agente di DDQN. Si riporta di seguito uno *snippet* (parziale) che ne riporta le componenti principali. Si rimanda all'appendice "A. Codice Sorgente" per il codice intero.

```
from time import time, perf_counter
import sys

import numpy as np
```

```
import pandas as pd
import tensorflow as tf
import gym

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

from DDQN_trading import DDQNAgent, TradingEnvironment
from DDQN_trading.config import config
from DDQN_trading import utils

import warnings
for i in (UserWarning, FutureWarning, RuntimeWarning, DeprecationWarning):
    warnings.simplefilter("ignore", i)

# ------------------------------------------------- Setup Gym Environment
gym.register(
    id='trading-v0',
    entry_point='trading_environment:TradingEnvironment',
    max_episode_steps=config["steps_per_episode"]
)
print(f'Trading costs: {config["trading_cost_bps"]:.2%} | Time costs: {config["time_cost_bps"]:.2%}')
trading_environment = gym.make(
    'trading-v0',
    ticker=config["ticker"],
    steps_per_episode=config["steps_per_episode"],
    trading_cost_bps=config["trading_cost_bps"],
    time_cost_bps=config["time_cost_bps"],
    start_end=config["start_end"]
)
trading_environment.seed(42)

# ------------------------------------------------- Get Environment Parameters
state_dim = trading_environment.observation_space.shape[0]
num_actions = trading_environment.action_space.n
max_episode_steps = trading_environment.spec.max_episode_steps



# ================================================================================================
# ------------------------------------------------- Create a DDQN-Agent ----------------------------------------------------
# ================================================================================================
tf.keras.backend.clear_session()
ddqn = DDQNAgent(
    state_dim=state_dim,
    num_actions=num_actions,
    learning_rate=config["learning_rate"],
    gamma=config["gamma"],
    epsilon_start=config["epsilon_start"],
    epsilon_end=config["epsilon_end"],
    epsilon_decay_steps=config["epsilon_decay_steps"],
    epsilon_exponential_decay=config["epsilon_exponential_decay"],
    replay_capacity=config["replay_capacity"],
    architecture=config["architecture"],
    l2_reg=config["l2_reg"],
    tau=config["tau"],
```

```
    batch_size=config["batch_size"]
)


# ------------ Initialize variables
episode_time, navs, market_navs, diffs, episode_eps = [], [], [], [], []

# ------------------------------------------------- TRAIN AGENT
results = []
for episode in range(1, config["n_episodes"] + 1):
    this_state = trading_environment.reset()  # reset to 0 the environment due to new episode was started
    # iterate over the episode's steps
    for episode_step in range(max_episode_steps):
        # to understand if this_state is a tuple or a list of tuple (i.e. vectorized or step by step). I think the
second one
        action = ddqn.epsilon_greedy_policy(this_state.reshape(-1, state_dim))  # get an action
        next_state, reward, done, _, __ = trading_environment.step(action)  # given the action get S', R(t+1)
and done

        ddqn.memorize_transition(
                s=this_state, a=action, r=reward, s_prime=next_state, not_done=0.0 if done else 1.0
        )

        # if we have to train ANN, do the experience replay approach to update ANNs models
        if ddqn.train:
            # if experience has enough obs (>= batch_size) re-train ANN each_step! (update target-ANN each
tau steps)
            ddqn.experience_replay()
        if done:
            break
        this_state = next_state  # update current state with the next one

    # get DataFrame with sequence of actions, returns and nav values
    result = trading_environment.env.simulator.result()

    # get results of last step
    final = result.iloc[-1]

    # apply return (net of cost) of last action to last starting nav
    nav = final.nav * (1 + final.strategy_return)
    navs.append(nav)

    # market nav
    market_nav = final.market_nav
    market_navs.append(market_nav)

    # track difference between agent an market NAV results
    diff = nav - market_nav
    diffs.append(diff)

    # every 10 episode, print the temporary-results
    if episode % 10 == 0:
        utils.track_results(
            episode,
            # show mov. average results for 100 (10) periods
            np.mean(navs[-100:]),
            np.mean(navs[-10:]),
```

```
            np.mean(market_navs[-100:]),
            np.mean(market_navs[-10:]),
            # share of agent wins, defined as higher ending nav
            np.sum([s > 0 for s in diffs[-100:]]) / min(len(diffs), 100),
            time() - start,
            ddqn.epsilon
        )

    if len(diffs) > 25 and all([r > 0 for r in diffs[-25:]]):
        print(result.tail())
        break

trading_environment.close()
```

Inizialmente viene registrato l'Environment, servendosi della libreria Gym di OpenAI, ampiamente descritta in precedenza. Il TradingEnvironment, infatti, è stato costruito dotandola di una c.d. *step_function*, che ne permette la compatibilità con l'ambiente di lavoro di gym. Successivamente, viene creato l'ambiente specifico per la simulazione che si è andata ad eseguire. Il suddetto ambiente viene istanziato utilizzando i parametri salvati nel *configuration file*; tali parametri verranno presentati e motivati nel Capitolo 3.

Lo stesso approccio viene utilizzato per la creazione dell'istanza dell'agente di DDQN.

All'interno del *for loop* si può apprezzare l'effettivo allenamento dell'Agente. Si può notare sia un *loop* "esterno" che itererà sugli episodi, sia un *loop* "interno" che itererà sugli *steps* dell'episodio. Ad ogni episodio, prima di entrare nell'iterazione dei singoli *steps*, l'Environment verrà resettato. Una volta all'interno del *loop* interno, ad ogni *step* di questo, si farà prendere una scelta all'Agente, una volta fornitogli il dataset degli stati. Successivamente si andranno a ricavare gli elementi mancanti per formare il vettore della transizione (stato successivo, *reward*, azione successiva), per poi andare a memorizzare il suddetto oggetto al fine di utilizzare l'approccio dell'*experience replay* una volta che la *deque* avrà una dimensione maggiore o uguale alla grandezza minima richiesta per il *batch* di training. Si andranno poi ad allenare le Reti Neurali, utilizzando sia il meccanismo dell'*experience replay* sia il meccanismo del Double-Learning. Il procedimento appena presentato verrà ripetuto per ogni *step* di cui si compone un episodio e per ogni episodio di cui si compone la simulazione.

Le righe finali invece, sono dedicate al *monitoring* delle prestazioni di apprendimento dell'Agente. Andranno a memorizzare i risultati ottenuti dall'agente durante i vari episodi, stampando in *console* un *summary* ogni 10 episodi.

Il file main.py contiene anche altre righe di codice, successive a quelle presentate in questo paragrafo, dedicate al *plotting* dei risultati post-simulazione. Si rimanda all'appendice "A. Source Code" per la letture delle suddette righe.

2.4 Computational Burden

Sicuramente l'esosa richiesta, in termini computazionali, legata all'allenamento del modello appena descritto è stato uno dei principali inconvenienti che si sono incontrati.

Andando a fornire una definizione squisitamente teorica del c.d. *Computational Burden*, è importante dire che esso è un termine che si riferisce alla quantità di risorse computazionali (ad esempio, tempo di elaborazione, memoria, larghezza di banda) richieste per eseguire un'operazione o un'applicazione specifica. In altre parole, il *computational burden* è una misura della complessità computazionale di un algoritmo o di un'applicazione. In molti contesti, la quantità di risorse computazionali richieste è un fattore critico per la progettazione e l'implementazione di sistemi e applicazioni. Ad esempio, in ambito scientifico e ingegneristico, il *computational burden* può essere un fattore limitante per la soluzione di problemi complessi o per la simulazione di processi fisici. Nell'apprendimento automatico, il *computational burden* può influire sul tempo necessario per addestrare un modello o sulla quantità di memoria richiesta per memorizzare i pesi del modello. Per ridurre il *computational burden*, i ricercatori e gli sviluppatori possono utilizzare tecniche come l'ottimizzazione dell'algoritmo, l'utilizzo di architetture di hardware più efficienti o l'utilizzo di tecniche di parallelizzazione e distribuzione del calcolo. In generale, il *computational burden* è un fattore critico da considerare nella progettazione e nell'implementazione di sistemi e applicazioni che richiedono un'elaborazione intensiva delle risorse computazionali. Una corretta valutazione del *computational burden* può aiutare a progettare sistemi e applicazioni che siano efficienti e che soddisfino le esigenze specifiche del contesto di utilizzo.

Quanto appena descritto, risulta essere esattamente il problema che si è riscontrato in fase di *set-up* del modello. Per fornire al lettore un esempio concreto, inizialmente si volevano utilizzare dati al minuto per sfruttare al massimo delle ipotetiche e temporanee inefficiente del mercato, disponendo unicamente di *features* tecniche e non Macro o Fondamentali. La simulazione di 10 episodi, composti da 1440 steps (i.e. 24h di trading), in cui si disponeva di un numero di esperienze tali da poter allenare le Reti, richiedeva dai 30 ai 50 minuti. Se

si considera che, data la configurazione prescelta per il modello, si richiedevano 1000 episodi per allenare il modello, risulterà evidente l'impossibilità di attendere dalle 30 alle 50 ore per ottenere un risultato. Il *Computational Burden* risulta essere ancor più evidente se si pensa che per ottenere una configurazione adeguata per il modello, o per svolgere Cross-Validation, si necessita di eseguire decine (se non centinaia) di simulazioni. Pertanto, non disponendo di macchine particolarmente potenti o, ancor meglio, di un *cluster* di macchine su cui lanciare le simulazioni, molte delle scelte che sono state prese in fase di calibrazione del modello e di selezione dei dataset da utilizzare sono dipese dalla necessità di svolgere simulazioni operativamente fattibili per un normale *laptop*.

Ad ogni modo, nel capitolo 4, "Conclusioni & Sviluppi Futuri" verranno comunque discusse delle ipotetiche raffinazioni che potrebbero essere inserite nel presente modello.

CAPITOLO 3

RISULTATI

3.1 Considerazioni Iniziali

Prima di presentare i risultati effettivamente ottenuti nella parte sperimentale della tesi, si vuole porre l'attenzione su una serie di aspetti.

In *primis*, all'interno del sotto-paragrafo 3.1.1 "*Fase di Training e fase di Test*", si andranno a descrivere le varie configurazioni che sono state studiate, sia dal punto di vista dello *zoo* di *features* che dal punto di vista degli *hyperparameters* utilizzati. Nel paragrafo 3.3 "*Setup del Modello"* si andrà a descrivere la configurazione finale che è stata effettivamente utilizzata nella simulazione che viene presentata all'interno di questa tesi.

In un secondo momento, all'interno del sotto-paragrafo 3.1.2 "*Considerazioni sulle tecniche utilizzate*", si andranno a svolgere alcune considerazioni su alcune tecniche che potevano essere utilizzate e che, a causa del *computational burden* e del tempo effettivo di implementazione, non è stato possibile utilizzare.

3.1.1 Fase di Training e fase di Test

A differenza di molti problemi classici di RL, in cui il *training-set* ed il *test-set* coincidono[40], nell'ambito dell'*algorithmic trading* si rende necessario dividere l'allenamento dell'agente dal *testing* della sua abilità. Questo è vero poiché, facendo coincidere *training* e *test* si andrebbe incontro ad un fortissimo *overfitting*, istruendo e valutando la *performance* (test) dell'agente con gli stessi dati. La vera sfida, infatti, risulta essere quella di allenare l'agente su di un set di dati (i.e. *training-set*) e poi ottenere ottime prestazioni anche su dati che non gli sono mai stati presentati (*test-set*). Qualora questo accadesse, si potrebbe dire di aver costruito un agente con una buona capacità di generalizzazione.

[40] Si pensi ad un classico gioco Gridworld, dove un agente deve imparare come muoversi all'interno di una griglia, al fine di arrivare da un punto iniziale A ad un punto finale B nel minor numero di mosse possibili. In questo problema, risulta ovvio come non vi è la necessità di dividere il *training* dal *test*, poiché l'apprendimento ottenibile nel periodi di *training* è direttamente spendibile in un'applicazione di *test*.

Sebbene spesso si divida il dataset in *training*, *validation* e *test*, all'interno della presente tesi si è deciso di testare direttamente l'agente dopo averlo allenato, eliminando la parte di *validation*.

Per la scelta della configurazione finale del modello, oltre ad aver svolto simulazioni su diversi titoli, come ampiamente descritto nel paragrafo 2.2.1 "*Dati*", si sono anche esplorate diverse configurazioni delle *features* e degli *hyperparameters* del modello. Di seguito si riporta un punto elenco con diverse considerazioni in merito:

- ***Features***
    - Inizialmente si è provato a fornire un *set* di *features* incentrato su *trading-style* preciso. Il risultato è stato che, sia componendo uno *zoo* sbilanciato verso uno stile di *trend-following*, sia costruendolo verso uno stile di *mean-reverting*, quello che si otteneva era un algoritmo parzialmente "miope", che tendeva a sfruttare (giustamente) solo una serie di *paths* della *price-action*. Pertanto, nella configurazione finale si è optato per uno *zoo* di *features* bilanciato tra i due *trading-styles*. Si rimanda al sotto-paragrafo 2.2.2 "*Features Engineering*" per un'ampia descrizione della composizione dello *zoo*.

- ***Hyper-parameters***

    Numero di episodi

    - Questo iper-parametro decide quanti sono gli episodi di cui si compone una simulazione. È stato un aspetto molto delicato, in quanto va considerato in congiunzione al numero di *steps* per episodio e alla grandezza del *batch* per l'allenamento dell'Online-ANN in fase di *experience replay*. Ovviamente, la delicatezza nella configurazione di questo parametro va ricondotta al già citato problema del *computational burden*, poiché inserendo un numero di episodi elevato, accompagnato da un discreto numero di *steps* per episodio e da un non piccolo *batch-size*, si incorreva in tempi di calcolo ingestibile (svariate decine di ore), rendendo impossibile l'ottenimento di un risultato in tempi pratici. Inoltre, si è notato che inserire in una simulazione un numero troppo elevato di episodi,

porta ad un rischio di *over-fitting* del modello, nonché ad un appiattimento nell'apprendimento dell'agente dopo un determinato numero di episodi.

Steps per episodio

- Questo iper-parametro decide quanti sono gli *steps* che l'agente "percorre" all'interno di ogni episodio. Allo stesso modo del numero di episodi, la selezione di questo parametro incide in modo particolarmente pesante sul tempo di calcolo di una simulazione. Ad esempio, prendendo due tipologie di simulazioni: 1) dati giornalieri, 1000 episodi con 252steps ad episodio (i.e. 1 anno); 2) dati al minuto, 1000 episodi con 1440steps ad episodio (i.1. 1 giorni nel FX); le due simulazioni genereranno un totale di 252,000 *steps* per il caso 1) e di 1,440,000 per il caso 2), richiedendo dei tempi di calcolo estremamente differenti. In linea di massima, la regola che si è voluta utilizzare nella definizione del numero di *steps* per episodio segue una logica di *trading*, pertanto, usando dati giornalieri si è provato ad usare 22-25 *steps* per simulare 1 mese di trading ad episodio, ~130 *steps* per simulare 6 mesi di *trading* e così via

Epsilon

- Avendo deciso di utilizzare una *policy* del tipo $\mathcal{E}$*-greedy*, questo iper-parametro va a determinare il livello di esplorazione dell'agente. Dopo aver provato diverse configurazioni, essendo molto ampio lo spazio continuo degli stati, si è notato che una maggiore esplorazione nella fase iniziale di simulazione portava a risultati nettamente migliori, permettendo un *mapping* migliore dei *q-values*. Inoltre, all'interno di questo iper-parametro vanno sicuramente citate le tecniche di decadimento dell'esplorazione in sé. In una prima parte della simulazione l'*epsilon* decade in modo lineare, per poi decadere in modo esponenziale fino a tendere a 0 (i.e. solo *exploitation*).

Gamma

- Questo parametro controlla quanto il "futuro" deve essere tenuto in considerazione durante l'aggiornamento dei *q-values*. Si può apprezzare la sua influenza nell'Equazione (1.9) all'interno del sotto-

paragrafo 1.4.2 "*Q-Learning*". Così come in molti problemi di RL, la forza di questa tipologia di modelli è proprio quella di tenere in considerazione l'effetto di un'azione presa oggi su ciò che l'agente si troverà a scegliere in un secondo momento, pertanto, dopo diverse prove si è notato come tale parametro deve essere impostato su un valore tendente ad 1, istruendo un agente che non massimizza solo l'azione successiva ma bensì un *path* di azioni che continuano nel futuro.

Tau

- Questo parametro controlla ogni quanti *steps* la Target-ANN viene aggiornata con i pesi dell'Online-ANN. Non si sono notate particolari differenze nella variazione di tale parametro, se non in termini di tempi di calcolo, poiché ad esempio, aggiornare la Target-ANN ogni 10 *steps* è diverso da aggiornarla ogni 10,000 *steps*.

ANN-*architecture*

- Questo parametro va a controllare la struttura delle Reti Neurali che si andranno ad allenare in fase di simulazione. Risulta essere un parametro particolarmente delicato, poiché all'aumentare della profondità della Rete Neurale si richiede un maggiore tempo di calcolo per il suo allenamento, nonché si rischia di avere un *overfitting*, tale per cui la Rete approssima bene i *Q-values* in fase di *training* ma ha una pessima qualità di generalizzazione in fase di *test*. Il rischio di *overfitting* viene in parte ridotto utilizzando il meccanismo del *Dropout* e della *Regularization*, ampiamente descritti nel sotto-paragrafo 2.3.5 "*DDQN-Agent*".

Learning rate

- Il Learning rate corrisponde all'alpha nell'Equazione (1.9), all'interno del sotto-paragrafo 1.4.2 "*Q-Learning*", e controlla quanto una nuova esperienza andrà a modificare il valore dei *Q-values*, ovvero quanto si aggiorneranno i suddetti valori al sopraggiungere di nuove informazioni (i.e. nuove *rewards*). Dopo diversi tentativi, si è notato che l'approccio migliore risulta essere quello di utilizzare un *learning rate* molto basso (quasi tendente a 0)

e di prediligere un numero molto elevato di esperienze, al fine di evitare che poche esperienze condizionino il valore dei *Q-values*.

Experience replay

- Gli iper-parametri che rientrano all'interno dell'*experience replay* sono: i) *replay capacity*; ii) *batch-size*. Il primo definisce quanto grande è la *deque* che andrà a memorizzare le transizioni, il secondo definisce invece quanto grande è il *batch* (i.e. il numero di esperienze) che vengono prese ad ogni *step* per allenare l'Online-ANN. Mentre la *replay capacity* non risulta essere un parametro cruciale (basta fornire uno spazio in memoria abbastanza grande da memorizzare molte esperienze), il secondo parametro diviene fondamentale poiché andrà a definire il numero di esperienze che verranno utilizzate per allenare la Rete. Scegliere un numero troppo grande per il *batch-size*, sebbene porti ad un vantaggio in termini di apprendimento delle Reti, conduce anche a tempi di calcolo molto elevati. Si sono quindi provate diverse configurazioni al fine di gestire il *trade-off* tra qualità di apprendimento e tempo di calcolo.

3.1.2 Considerazioni sulle tecniche utilizzate

In merito alle tecniche utilizzate, si è cercato di inserire tutte quelle metodologie che rendessero l'apprendimento dell'Agente quanto più efficiente, generalizzato e *unbiased* possibile. Le caratteristiche appena citate, sono state le motivazioni che hanno portato all'utilizzo del *memory replay*, del *Double-Learning*, del *Dropout* e della *Regularization* all'interno delle Reti Neurali.

Ciononostante, diverse sono le tecniche di allenamento che si sarebbero potute utilizzare per dare un *boosting* all'apprendimento dell'Agente. Tali tecniche risultano essere molto più indirizzate alla ricerca dell'*alpha*, piuttosto che al miglioramento dell'algoritmo di Reinforcement Learning in sé. Di seguito si riportano alcune migliorie che, in assenza di problemi di *computational burden* e di tempo di implementazione, si sarebbero potute attuare:

- *Features Selection*
    - Disponendo di un generatore di *features* molto più ampio, che contenesse non solo FE di analisi tecnica ma anche di natura Macro,

fondamentale e di *news*, si sarebbe potuto sviluppare un algoritmo di *features selection* al fine di fornire all'algoritmo informazioni più mirate. Si parlerà di questa tematica, in modo più approfondito, nel paragrafo 4.2 "*Sviluppi futuri*".

- *Training/Test alternato*
  - Una tecnica che spesso viene utilizzata in processi maggiormente industrializzati (i.e. in società che implementano strategie di *algorithmic trading*), è quella di allenare un algoritmo su di un periodo ristretto, testarlo e poi ricominciare il processo. Ad esempio, si potrebbe allenare l'agente su 3-6 mesi di dati (e.g. Gennaio-Giugno), testarlo *out-of-sample* su un periodo di 2 mesi (e.g. Giugno-Agosto), per poi spostare *training* e *test* di 2 mesi in avanti (e.g. *training* Marzo-Agosto, *test out-of-sample* Settembre-Ottobre); infine fare un *join* di tutti questi test a 2 mesi, per ottenere una serie storica di *performance out-of-sample* dell'algoritmo. Utilizzare questo approccio, permette all'algoritmo di imparare qual è la strategia più aderente al "passato più recente", evitando di insegnargli come fare *trading* con dati troppo vecchi che potrebbero contenere regimi di mercato passati che non si ripeteranno.

Pertanto, alla luce delle considerazioni appena svolte, si vuole sottolineare che la presente tesi ha voluto dare maggiore attenzione alle tecniche di allenamento di un agente che si riconducono direttamente al Reinforcement Learning, piuttosto che utilizzare metodologie che appartengono di più al mondo del Trading Algoritmico.

È comunque doveroso citare l'esistenza dei suddetti approcci, che permettono di migliorare di molto la *performance* di un algoritmo di trading. Si rimanda al paragrafo 4.2 "*Sviluppi futuri*" per una trattazione più dettagliata di queste tecniche.

3.2 Metodi e metriche di Valutazione

Di seguito si andranno a descrivere le metriche di valutazione che sono state utilizzate per valutare l'apprendimento dell'Agente, sia in fase di *training* che in fase di *testing*.

È stata fatta una differenziazione tra i due datasets per una serie di ragioni specifiche.

Essendo il training diviso in episodi, ognuno di questi produce diverse serie storiche che si necessita di analizzare per valutarne la *performance* ma che, ancor prima di analizzarle, si ha la necessità di memorizzare. Memorizzarle richiede, ovviamente, spazio nella memoria temporanea (i.e. la RAM). Dato che oltre ai già citati problemi di *computational burden*, si sono incontrati anche problemi di *overload* della memoria[41], motivo per il quale si è deciso di non salvare il *path* delle *performance* per ogni singolo episodio ma unicamente il *path* generale al termine di ognuno di essi.

Diverso è il caso del test, che lo si potrebbe interpretare come un unico episodio, il quale permette quindi di memorizzare tutte le informazioni e i dettagli di cui si necessita per fare un'approfondita analisi della *performance*.

Le ragioni appena descritte sono le motivazioni per cui si avrà una presentazione dei risultati differente tra il *training* e il *test*. Nel primo ci si concentrerà maggiormente sul concetto di apprendimento dell'Agente nel tempo, mentre nel secondo caso si andrà ad indagare in profondità la *performance* che l'Agente è in grado di ottenere dopo aver imparato una strategia, ovvero la sua abilità nella generalizzazione della *policy* applicata a dati mai incontrati.

3.2.1 Training-set

Come anticipato, quello che si vuole andare ad analizzare nella fase di *training* è la capacità di apprendimento dell'Agente durante l'evolversi degli episodi. Come si è già più volte affermato, l'allenamento di un agente nell'ambito del trading algoritmo, nonché su dati reali definiti in uno spazio continuo che costituiscono uno stato incompleto in termini informativi, genera dei risultati leggermente diversi dai più popolari problemi di RL. Al fine di rendere chiara la comprensione dei risultati che si andranno a presentare, risulta importante fare un esempio grafico dell'apprendimento di un agente, all'interno di un problema molto semplice. Di seguito si riporta l'evoluzione dell'apprendimento di un agente su di un problema di c.d. "GridWorld". Il grafico viene generato da uno *script*

[41] Per "*overload* della memoria", in informatica, si intende quella situazione in cui la memoria del sistema non è in grado di gestire tutti i dati che vengono richiesti. Ciò porta, nel caso specifico di un programma che si sta eseguendo, all'interruzione dello stesso, proprio a causa di un sovraccarico della memoria temporanea, ovvero della RAM.

presente nella Repo implementata per la tesi ed è locato all'interno del *folder* "others_application/TD_algos".

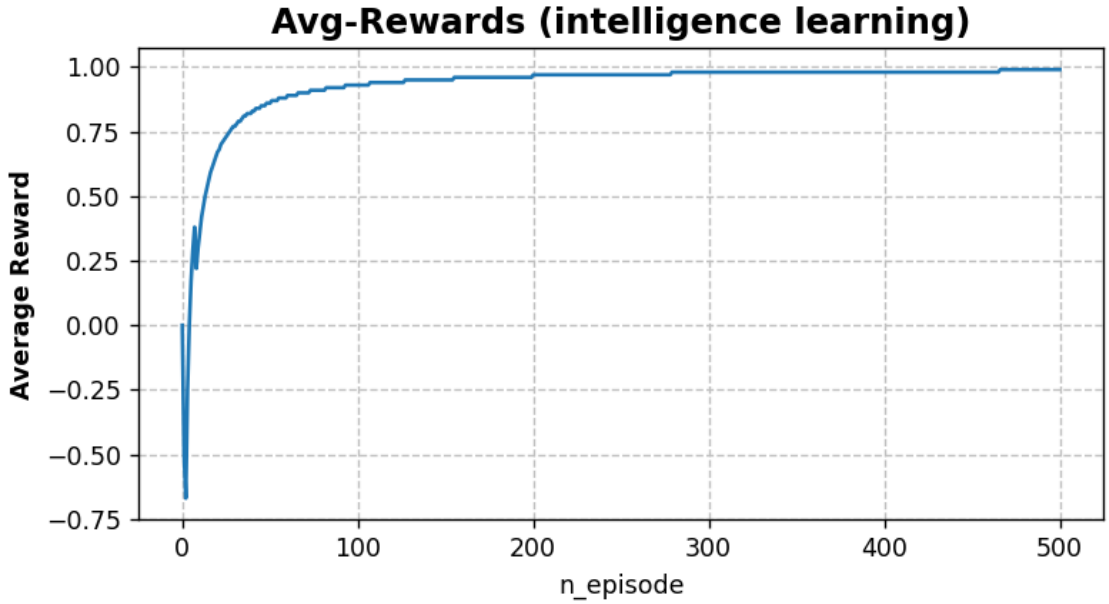


Figure 18: Evoluzione dell'apprendimento di un Agente in un problema di tipo GridWorld

All'interno dello *script* si troverà una descrizione accurata dell'algoritmo di Q-Learning utilizzato, nonché del problema che questo è andato a risolvere. In questo paragrafo si vuole porre l'attenzione sulla composizione del grafico. Sull'asse delle ascisse vengono riportati gli episodi, mentre sull'asse delle ordinate le *rewards* ottenute durante ogni episodio. Risulta evidente che il grafico vuole mostrare quanto l'algoritmo di RL riesce a migliorare all'aumentare della propria esperienza (i.e. dell'allenamento). In questo particolare caso, con uno spazio discreto degli stati, l'intelligenza dell'algoritmo converge all'ottimo, riuscendo a risolvere il problema nel miglior modo possibile. Come si vedrà in seguito, nel caso del trading algoritmo questo non è sempre vero, poiché non vi è una "strada" che l'Agente può prendere per ottenere la massima *reward* possibile, essendovi una fortissima presenza di incertezza. Questa è la ragione per cui il grafico di apprendimento dell'Agente di DDQN implementato nella presente tesi, risulterà molto più frastagliato ed instabile, nonostante porterà a dei risultati più che soddisfacenti.

La **curva di apprendimento dell'Agente** sarà definità dal NAV ottenuto al termine di ogni episodio, essendo questo frutto della cumulazione delle *rewards* ottenute ad ogni *step*. Questa curva verrà graficamente confrontata con quella ottenuta dal mercato, ovvero da una strategia *long-only*, che rappresenterà il *benchmark* che si vuole battere. In particolare, si

presenterà la Media Mobile[42] a 100 periodi, dove ogni periodo fa riferimento ad un episodio, al fine di effettuare uno *smoothing* sulla curva che permette una migliore visualizzazione.

Verrà inoltre presentato il c.d. ***Winning-Rate***, anch'esso con una Media Mobile a 100 periodi, calcolato come rapporto tra il numero di episodi in cui l'Agente ottiene un NAV maggiore del *benchmark* e il numero di volte in cui è il *benchmark* ad ottenere un NAV più elevato.

3.2.2 Test-set

Per quanto riguarda l'analisi della *performance* in fase di test, come precedentemente anticipato, si andranno a presentare un maggior numero di metriche. Questo è motivato sia da minori problemi di *overload* della memoria, sia dal fatto che è proprio in fase di *test* che si va a vedere quanto un algoritmo è in grado di generalizzare sui dati e quanto è "buona" la sua *performance*. Di seguito si riporta un elenco delle metriche che verranno utilizzate:

- **Average Annualized Returns**: si calcolerà la media dei rendimenti ottenuti in fase di test e lo si annualizzerà
- **Average Annualized Volatility**: si calcolerà la standard deviation dei rendimenti ottenuti in fase di test e lo si annualizzerà
- **Average Annualized Semi-Volatility**: si calcolerà la standard deviation solo per quei rendimenti che sono inferiori a 0, poi la si annualizzerà. Questa metrica permette di capire qual è la dispersione intorno alla media per quei rendimenti che generano perdite
- **Max Drawdown**: si riferisce alla perdita massima registrata da un investimento o un portafoglio di investimenti, misurando la percentuale di diminuzione dal massimo valore registrato al minimo. Questa metrica permette di capire quant'è la perdita massima dal massimo che ci si può aspettare da una strategia di trading. Risulta essere molto importante poiché, molto spesso, gli investitori definiscono una soglia massima di perdita al di là del quale si interrompe (*turn-off*) una strategia algoritmica

[42] Si rimanda al sotto-paragrafo 2.2.2.1 "*Medie Mobili – Moving Averages*" per una descrizione accurata delle stesse.

- **Var-99% (parametric and not)**: è una misura di rischio che indica qual è la perdita massima che ci si aspetta da una strategia, dato un livello di confidenza, in questo caso 99%. Pertanto, leggendo il Var-99% si può approssimativamente affermare che "vi è solo l'1% di probabilità di osservare una perdita maggiore a questo valore".
- **Daily Skewness:** indica l'asimmetria della distribuzione dei rendimenti giornalieri. Ciò che si ricerca, ovviamente, è un'asimmetria positiva che si traduce in rendimenti inattesi, rispetto ad una distribuzione normale, più positivi che negativi.
- **Daily Kurtosis:** è una misura statistica che indica quanto una distribuzione di rendimenti finanziari è "appuntita" o "piatta" rispetto alla distribuzione normale. In particolare, la *kurtosis* descrive la forma e la presenza di eventuali code pesanti nella distribuzione dei rendimenti, rispetto alla distribuzione normale, che ha una *kurtosis* pari a 3. Pertanto, una *kurtosis* elevata suggerità *fat-tails*, quindi la possibilità di osservare rendimenti estremi. È bene leggere la *kurtosis* insieme alla *skewness*, al fine di capire dove ci si aspetta che questi rendimenti inattesi potrebbero cadere.
- **Annualized Sharpe Ratio:** è una misura di rendimento ponderata per il rischio. Indica quante unità di rendimento ci si aspetta di ottenere per una singola unità di rischio. Permette di analizzare la profittabilità di una strategia, tenendo allo stesso tempo in considerazione la rischiosità della stessa.
- **Annualized Sortino:** è molto simile allo Sharpe, con la differenza che l'unità di rischio viene misurata in termini di semi-volatility piuttosto che di volatility.
- **Best day:** corrisponde al più alto rendimento giornaliero ottenuto dalla strategia
- **Worst day:** corrisponde al peggiore rendimento giornaliero ottenuto dalla strategia

Nonostante l'analisi della *performance* di una strategia di investimento potrebbe essere ampiamente estesa rispetto a quella che verrà presentata, la numerosità di metriche utilizzate è stata ritenuta idonea a descrivere la bontà della strategia imparata dall'Agente.

3.3 Setup finale del Modello

Alla luce delle considerazioni svolte nel sotto-paragrafo 3.1.1 "*Fase di training e di test*" sugli *hyper-parameters* dell'Environment e dell'Agente, si andrà a presentare la

configurazione finale che si è deciso di utilizzare per l'allenamento dell'Agente di DDQN implementato.

- `number_of_episodes`: 2000
- `steps_per_episode`: 25 (i.e. 1-month)
- epsilon[43]
    - `starting_value`: 1
    - `decay_steps`: 750
    - `final_value`: 0.01
    - `exponential_decay`: 0.99
- `gamma`: 0.99
- `tau`: 100
- ANN-architecture
    - 1° layer: 15-input, 128 neurons
    - 2° layer: 256 neurons
    - 3° layer: Dropout
    - `activation_function`: 'ReLu'
    - `loss_function`: MSE
    - optimizer: ADAM
- `learning_rate`: 0.001
- experience_replay
    - `replay_capacity`: 1,000,000
    - `batch_size`: 1500

3.4 Risultati della configurazione migliore

Nei successivi sotto-paragrafi si andranno a descrivere i risultati ottenuti utilizzando la configurazione descritta nel paragrafo precedente. Come accennato in precedenza, si dividerà la trattazione tra risultati in fase di *training* e risultati in fase di *test*.

[43] `decay_steps` e `final_value` indicano in quanti steps l'epsilon passerà dallo `starting_value` al `final_value`, prima di iniziare a "decadere" in modo esponenziale

S ricorda che il periodo di *training* e di *test* è diviso come segue:

- Il training-set inizia il 2013-02-19 e termina il 2020-12-23
- Il test-set inizia il 2021-01-04 e termina il 2021-06-01

3.4.1 Risultati sul Training-set

Si ricorda che, la funzione di *reward* che l'Agente tende a massimizzare consiste nel PNL ottenuto da questo durante un episodio. Pertanto, quando si parlerà di PNL per episodio, si farà riferimento alla funzione di *reward* massimizzata dall'Agente.

Nella figura 19 si possono apprezzare i risultati della fase di *training*. I due grafici in alto rappresentano il PNLs, rispettivamente dell'Agente di DDQN e del Benchmark[44]. Con "*point-in-time*" ci si riferisce al PNL ottenuto nel singolo episodio, mentre la MA(150) consiste in una media mobile su 150 episodi. Ricordando che un episodio si compone di 25 *steps* (~1 mese), guardando alla media sull'intero periodo di allenamento, risulta da subito evidente come l'Agente performi meglio del benchmark, ottenendo una media di PNL pari a 2.8% contro lo 0.2% della *long-only*. Inoltre, bisogna considerare che inizialmente l'Agente tende a non fare un "buon *trading*", avendo inserito una configurazione tale per cui vi è molta esplorazione nei primi 500-750 episodi. Infatti, guardando al grafico in basso che confronta le medie mobili a 150 episodi per l'Agente e per il benchmark, si può notare con maggiore chiarezza come l'Agente sovra-performi il benchmark.

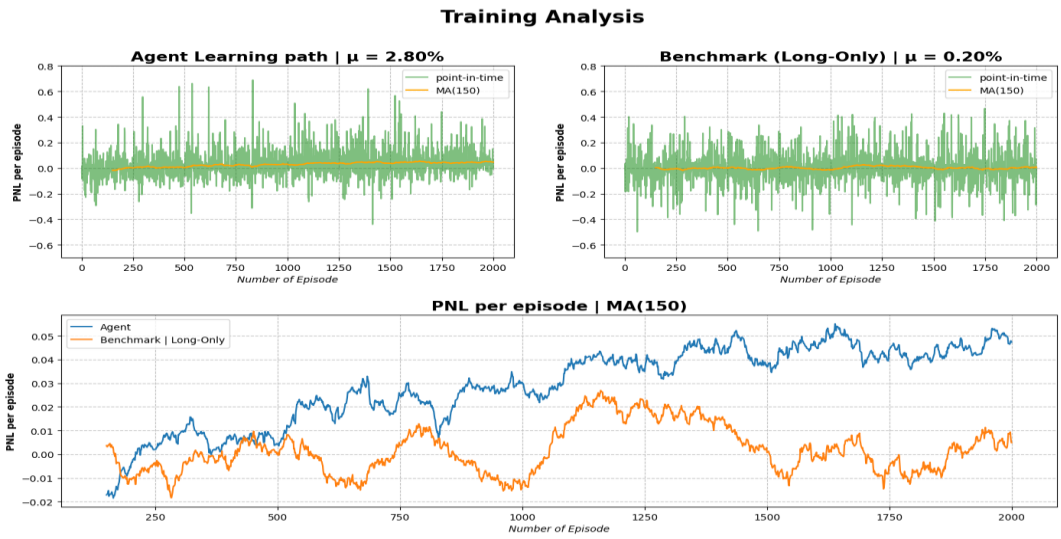


Figure 19: Analisi dell'apprendimento dell'Agente nella fase di Training

[44] Come benchmark si è voluto utilizzare una semplice strategia Long-Only, ovvero andare *long* sul titolo per tutti e 25 gli *steps* (i.e. i giorni) di cui si compone un episodio.

Particolarmente interessante risulta essere il comportamento della MA(150) dopo l'episodio 1250. Seppur con un po' di varianza, vi è un evidente "appiattimento" dell'apprendimento, facilmente riscontrabile in una diminuzione della pendenza della curva. Questo è sicuramente causato da un "limite" di apprendimento, probabilmente perché l'Agente ha appreso tutto ciò che poteva apprendere, dato il *set* di *features* fornitogli. Vi è una leggera pendenza positiva, ma si può tranquillamente affermare che un miglioramento nelle *performance* così lieve nella parte finale, è sicuramente causato da un fenomeno di *over-fitting*.

Di seguito, nella Figura 20, si può apprezzare nel dettaglio una MA(150) del Delta tra i PNLs per episodio ottenuti dall'Agente e dal *benchmark*. Ricordando che fino al 750esimo episodio vi è una forte componente di esplorazione dell'Agente, si può notare un iniziale peggioramento post-esplorazione, probabilmente dovuto ad un adattamento dell'Agente ad una strategia più *greedy*, che poi si traduce in ulteriore apprendimento e miglioramento della strategia. Così come per la Figura 19, si può apprezzare un "appiattimento" dei miglioramenti da dopo l'episodio 1250.

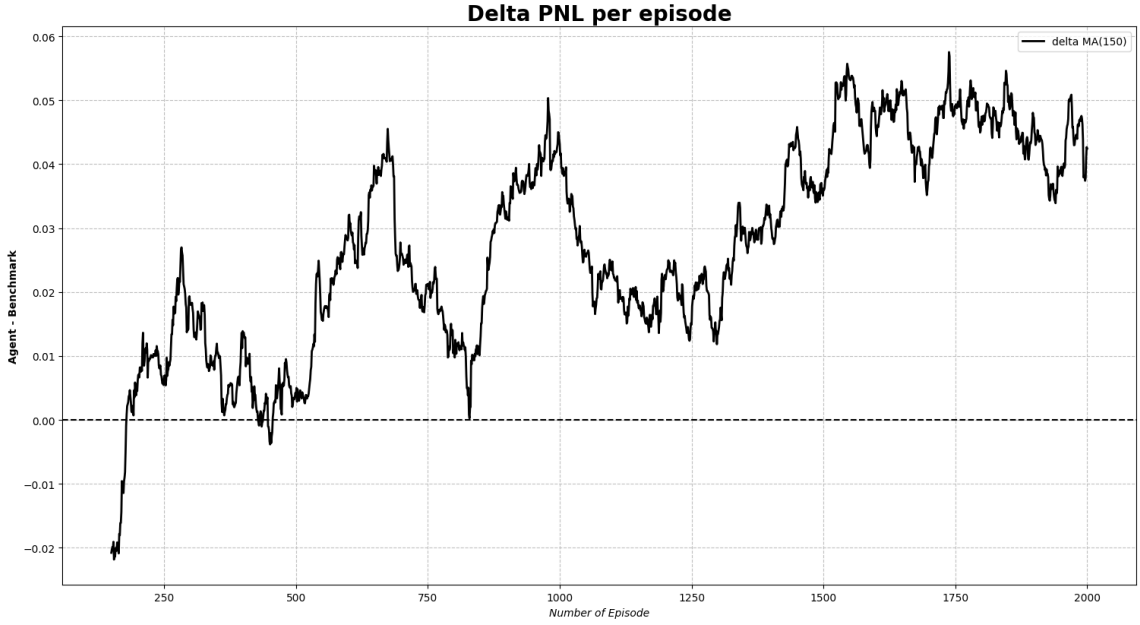


Figure 20: MA(150) del Delta tra i PNL per episode ottenuti dall'agente e dal benchmark

Infine, risulta utile analizzare il c.d. Winning-rate, ovvero la percentuale di episodi in cui l'Agente batte il *benchmark* in termini di PNL ottenuto. Anche in questo caso, viene riportata una media mobile a 100 episodi, al fine di fare *smoothing* sulla serie ed ottenere

una migliore visualizzazione del *path* evolutivo. Anche nella Figura 21, così come nella Figura 19 e 20, risulta evidente un *path* in miglioramento al trascorrere degli episodi.

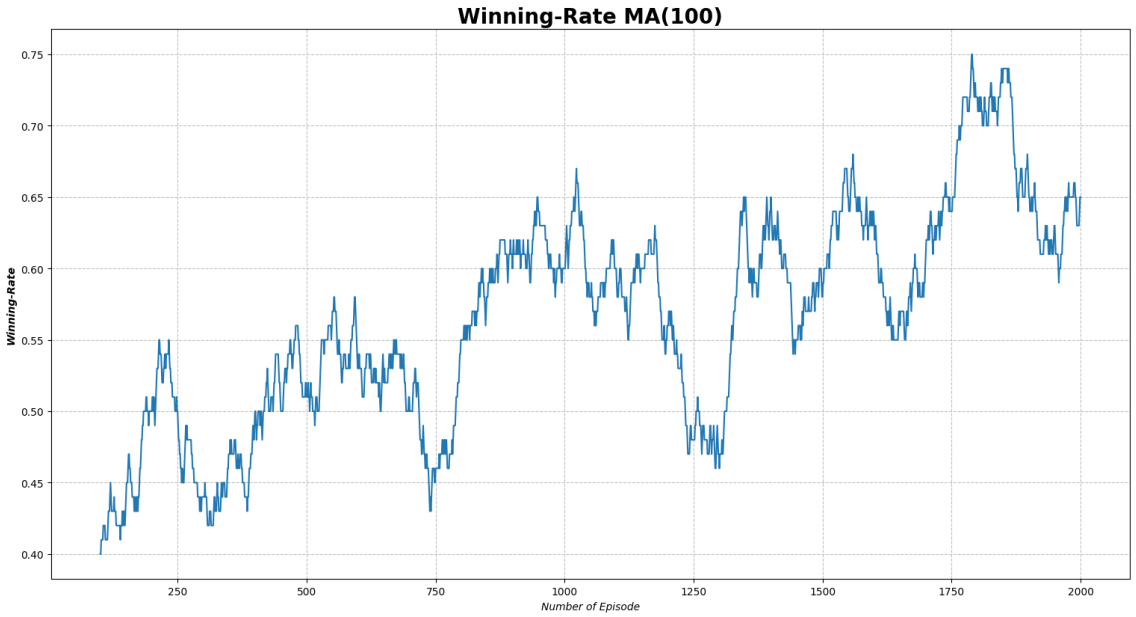


Figure 21: MA(100) sul Winning-Rate dell'Agente contro il benchmark

3.4.2 Risultati sul Test-set

Per svolgere il *test* dell'Agente di DDQN, si è salvato il modello (i.e. la Rete Neurale) post *training*, per poi andarla ad utilizzare all'interno di `main_test.py`, svolgendo un'analisi di *performance*.

Nella Figura 22, si riporta il risultato generato dall'Agente di DDQN nel periodo di *testing*. Il grafico posizionato nella parte superiore riporta il PNL ottenuto dall'Agente e dal Benchmark che, come nel caso del *training*, risulta essere una strategia *long-only*. Viene riportato sia il Gross-PNL che il Net-PNL, al fine di evidenziare come l'ottimizzazione dei costi di transazione, nonché un utilizzo ponderato dei *trades* in termini di numerosità, possa essere un punto cruciale sul quale fare leva per dare un *boosting* alla profittabilità della strategia. Risulta evidente come l'Agente riesce a sovra-performare il *benchmark*, nonostante sconti una penalizzazione data dai costi di transazione, penalità che non tocca la strategia *long-only*, poiché questa compra il titolo e lo detiene in portafoglio fino al termine.

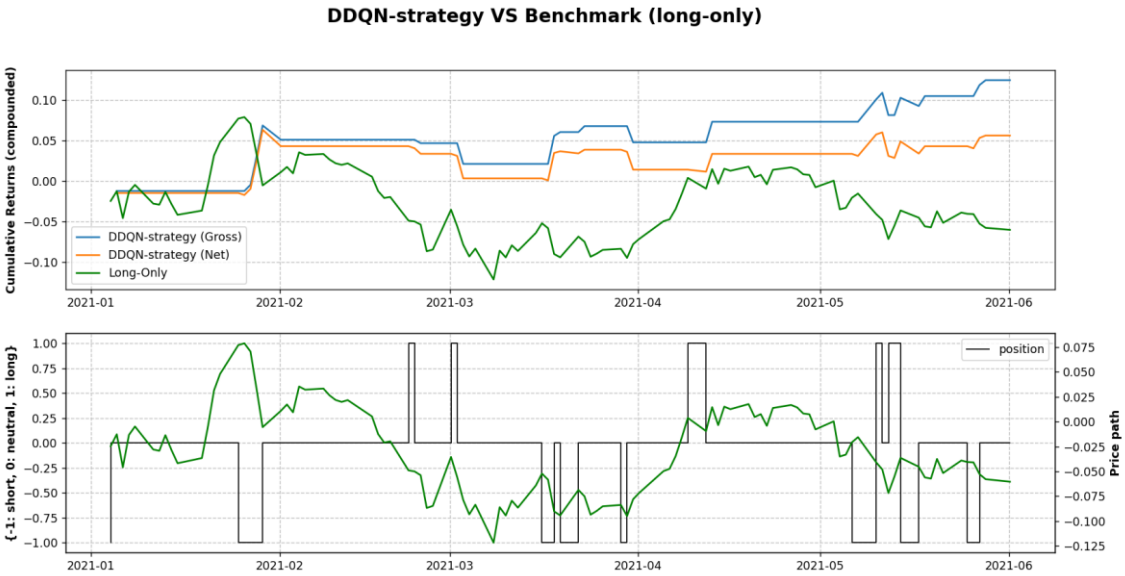


Figure 22: Test dell'Agente di DDQN

Invece, il grafico posizionato nella parte inferiore riporta la strategia di *trading* utilizzata dall'Agente di DDQN. Sull'asse delle ordinate di sinistra si possono osservare le posizioni prese dal *trader* {-1: *short*; 0: *neutral*; 1: *long*}, mentre nell'asse delle ordinate di destra si osserva l'andamento del prezzo del titolo. Da questo secondo grafico, risulta evidente come l'Agente è in grado di sfruttare una strategia *long-short* sul titolo, rendendola profittevole.

	DDQN-Agent (Gross)	**DDQN-Agent (Net)**	**Benchmark (*long-only*)**
Total Return	**12.5%**	**5.7%**	**-6.1%**
Avg Ann. Return	**30%**	**14.5%**	**-11%**
Avg. Ann. Volatility	**14%**	**13.5%**	**29.5%**
Avg. Ann. Semi-Volatility	12.3%	14.8%	17.9%
Max Drawdown	**-4.4%**	**-5.8%**	**-19%**
VaR (99%) Parametric	-2.1%	-2.12%	-4.3%
VaR (99%) Not Parametric	-2.4%	-2.69%	-3.73%
Daily Skewness	1.55	1.26	0.13
Daily Kurtosis	7.61	7.37	0.05
Ann. Sharpe	**2.14**	**1.23**	**-0.37**
Ann. Sortino	2.44	1.12	-0.61
Best Day	3.75%	3.75%	5.38%
Worst Day	-2.5%	-2.75%	-4.17%

La tabella di cui sopra, riporta un'analisi della *performance* dell'Agente di DDQN, sia considerando i costi di transazione che escludendoli, e del benchmark. Al lettore risulterà evidente sin da subito la sovra-performance dell'Agente. Si riportano di seguito alcune considerazioni in merito ai risultati ottenuti:

- L'Agente ottiene un *Total Return* nettamente maggiore (e positivo) rispetto al *benchmark*
- Annualizzando i rendimenti, si ottiene un *net-annualized-return* del 14.5%, non considerando i dividendi
- L'Agente riesce a ridurre la rischiosità del titolo. Ciò è osservabile sia in una minore volatilità, che registra un dimezzamento rispetto al *benchmark*, sia in un *max-drawdown* nettamente ridotto, precisamente pari ad un quarto di quello del *benchmark*. Ragionamento analogo vale per il VaR al 99%, che passa dal -4.3% del benchmark al -2.12% dell'Agente
- Il benchmark presenta una *Skewness* praticamente pari a 0, così come un'*excess-Kurtosis* assimilabile a quella di una distribuzione Normale Standard. L'Agente invece, presenta una *Kurtosis* piuttosto elevata, pari a 7.61, accompagnata da una *Skewness* positiva, pari a 1.55. Questo significa che nella strategia si osserveranno code più grasse, con una maggiore probabilità di osservare rendimenti positivi.
- Lo Sharpe della strategia, al netto dei costi di transazione, risulta piuttosto soddisfacente e pari a 1.23. Questo indica che l'Agente è in grado di bilanciare il concetto di rischio rendimento

In conclusione, si può apprezzare un ottimo risultato in fase di *test* per quanto riguarda la strategia appresa dall'Agente di DDQN. Si rimanda al Capitolo 4 "*Conclusioni e Sviluppi Futuri*" per maggiori considerazioni in merito a come si sarebbe potuto migliorare la strategia.

CAPITOLO 4

CONCLUSIONI & SVILUPPI FUTURI

4.1 Conclusioni

Giunti al termine della tesi, risulta ora necessario trarre delle conclusioni in merito all'utilizzo dei modelli di Reinforcement Learning per l'Algorithmic Trading e, in particolare, all'utilizzo del DDQN per tale scopo.

Come è stato ampiamente discusso nel Capitolo 1 "*Framework Teorico*", i modelli di RL applicati al Trading Algoritmico possono risultare piuttosto efficaci. Molte sono le migliorie che potrebbero essere apportate all'algoritmo presentato all'interno di questa tesi; migliorie che verranno trattate nel successivo paragrafo 4.2 "*Sviluppi Futuri*". Però, nonostante la parte sperimentale della tesi si è voluta concentrare maggiormente sulla ricerca di una configurazione dell'Agente e del *training* efficiente per il *trading*, tralasciando alcuni aspetti di natura operativa, i risultati ottenuti sono stati più che soddisfacenti. Ciò dimostra, effettivamente, come il mondo del RL, e più ampiamente il mondo del Machine Learning, possono (e sono!) una famiglia di tecniche e di modelli che risultano profittevoli per il mondo degli investimenti e, in particolare, per il mondo del *trading*.

È stato dimostrato come una calibrazione ponderata e bilanciata, accompagnata da un'accurata costruzione di un'architettura ed una selezione mirata dell'universo delle *features*, possono creare quegli ingredienti sufficienti ad allenare un algoritmo di DDQN che risulta profittevole nel *trading* di titoli finanziari. Tale affermazione è individuabile all'interno del Capitolo 3 "*Risultati*" e, in particolare, all'interno del paragrafo 3.4.2 "*Risultati sul Test-set*", che mostra come l'Agente è in grado di generalizzare anche su dati mai visti prima, punto particolarmente dolente quando si vogliono utilizzare algoritmi di ML e RL per il *trading*.

Si riassumono di seguito i punti chiave che sono stati affrontati durante la stesura della presente tesi:

- Sono stati descritti e approfonditi i concetti chiave del ML4T[45] e del Reinforcement Learning

[45] *Machine Learning for Trading*

- Un particolare *focus* è stato dedicato al Q-Learning e alle sue varie estensioni, quali Deep Q-Learning, Double-Learning, etc.
- Si è costruita un'architettura, sviluppata in Python, al fine di implementare un modello di DDQN. Tale architettura, non solo implementa l'algoritmo di RL appena citato, ma lo nutre di *features*, lo allena, lo testa e ne analizza le *performance* in entrambi gli ambienti
- Si è definita una configurazione, ritenuta ottimale, al fine di verificare l'effettiva profittabilità dell'algoritmo implementato

Le aspettative che inizialmente hanno alimentato questo progetto erano molto alte. Purtroppo, si è dovuto far fronte al già citato problema del *computational burden* che ha drasticamente ridotto l'universo di simulazione che inizialmente si voleva esplorare. La presente tesi però, risulta essere solo un punto di partenza per lo studio di questa famiglia di algoritmi applicati al Trading Algoritmico, studio che verrà indubbiamente perpetuato anche dopo la conclusione del percorso di Laurea che si corona con la consegna di questo documento, poiché il limite che esiste nello sviluppo di architetture algoritmiche di trading è quello che la nostra mente può vedere, bisogna solo "alzare l'asticella" e continuare a sperimentare nuove tecniche e a migliorare quelle già esistenti.

4.2 Sviluppi Futuri

Tra gli sviluppi futuri che potrebbero estendere lo studio svolto all'interno della presente tesi, vale la pena citare:

- Training e test "a step"

  Questa tecnica consiste nell'allenare l'algoritmo per un determinato periodo (e.g. 3 mesi) e testarlo nel periodo immediatamente successivo (e.g. 1 mese), poi scorrere in avanti del periodo del test (1 mese) e ri-allenare l'algoritmo su questo nuovo set di *training* per poi testarlo sul periodo successivo. Questa è una tecnica molto utilizzata nel mondo del *trading* algoritmico, in quanto permette di mantenere l'intelligenza dell'algoritmo molto più "aderente" al presente, rispetto ad una classica tecnica di allenamento statica, come quella utilizzata all'interno della presente tesi.

- *Features Selection* guidata da algoritmi di causalità

  Potendo disporre di un universo di *features* molto più vasto di quello utilizzato all'interno della presente tesi (e.g. 100,000 features per un titolo), risulterebbe molto interessante sviluppare un algoritmo che va a selezionare un *pool* di *features* ristretto da fornire all'Agente. Unendo questa tecnica alla tecnica di allenamento/training "a step" di cui sopra, si potrebbe fornire un *set* di *features* differente ad ogni allenamento, utilizzando di volta in volta quelle serie storiche che meglio descrivono un titolo in un determinato periodo storico

- Costruzione di *features* "intelligenti"

  Si potrebbero utilizzare algoritmi che ottimizzano la costruzione delle *features*. Si pensi, ad esempio, ad una semplice media mobile, *features* estremamente influenzata dalla dimensione della finestra che si utilizza. Si immagini di disporre di un algoritmo (e.g. un Genetic Algorithm) che va a calibrare, di volta in volta, la finestra con la quale viene calcolata la media mobile. Questo permetterebbe di fornire all'algoritmo un *set* di *features* ottimizzato ed "intelligente", migliorando idealmente la *performance* che lo stesso potrebbe ottenere "studiando" da tale *set*.

- L'utilizzo di algoritmi di RL alternativi, quali il PPO o A2C, che risultano particolarmente performanti alla luce degli ultimi studi che si stanno facendo sull'utilizzo di questi all'interno del trading algoritmico

- L'utilizzo di una funzione di *reward* differente dal rendimento della strategia. Premiando/punendo l'Agente con una funzione più complessa, come ad esempio lo Sharpe Ratio o il Gain/Loss ratio, gli si potrebbe insegnare che sebbene il rendimento che si ottiene dal *trading* è importante, lo è anche il rischio che si corre nell'eseguire determinati *trades*.

A. Codice Sorgente

Nella presente appendice si riporta il codice sorgente che è stato implementato per lo sviluppo dell'architettura del modello, descritto e presentato nella parte sperimentale della tesi.

Il progetto si compone come segue:

DDQN_for_trading

- data
  - datasets
    - … some datasets, extension .parquet
  - get_data.py
  - resampler.py
- DDQN_trading
  - results
    - … several folder, each folder contains a simulation's result
  - __init__.py
  - agent.py
  - config.py
  - data_source.py
  - main_train.py
  - main_test.py
  - graph_analysis.py
  - performance_analysis.py
  - tester.py
  - trading_environment.py
  - trading_simulator.py
  - utils.py
- Features_engineering
  - costants
    - parametrization_1.py
  - __init__.py
  - fe_generator.py
  - features.py
  - main.py
  - utils.py
- others_application
  - some RL application (no finance)

A.1 DDQN_trading folder

Nei successivi sotto-paragrafi si riporta il codice sorgente relativo al folder "DDQN_trading", che contiene i moduli python relativi al modello di RL implementato per

la parte sperimentale della tesi. E' stato escluso il file config.py, contenente la configurazione utilizzata per i risultati presentati nel capitolo 3 "Risultati", in quanto rappresenta un file di configurazione e non codice sorgente.

A.1.1 DataSource class

```
                import os
import logging
import pandas as pd
import numpy as np
from sklearn.preprocessing import scale

from typing import *


logging.basicConfig()
log = logging.getLogger(__name__)
log.setLevel(logging.INFO)
log.info('%s logger started.', __name__)


class DataSource:
    """
    Data source for TradingEnvironment
    - Loads & preprocesses prices
    - Provides data for each new episode.
    """

    def __init__(
            self,
            steps_per_episode: int = 60 * 24,
            ticker: str = 'EURUSD2022_1m',
            normalize: bool = True,
            start_end: Optional[Tuple[str, str]] = None
    ):
        self.ticker = ticker
        self.steps_per_episode = steps_per_episode
        self.start_end = start_end
        self.normalize = normalize
        self.data = self.load_data()
        self.load_features()
        self.min_values = self.data.min()
        self.max_values = self.data.max()
        self.step = 0
        self.offset = None

    def load_data(self) -> pd.DataFrame:
        log.info('loading data for {}...'.format(self.ticker))
        file_path = os.path.dirname(os.getcwd()) + rf"\data\dataset\{self.ticker}.parquet"
        df = pd.read_parquet(file_path)
        if self.start_end:
            start, end = self.start_end
            df = df.loc[start: end]
        log.info('got data for {}...'.format(self.ticker))
        return df[['close']]
```

```
    def load_features(self) -> NoReturn:

        file_path = os.path.dirname(os.getcwd()) + rf"\data\dataset\features_{self.ticker}.parquet"
        fe_df = pd.read_parquet(file_path).loc[self.data.index]

        self.data = pd.concat([self.data, fe_df], axis=1).dropna()
        rets = self.data['close'].pct_change()

        if self.normalize:
            self.data = pd.DataFrame(
                data=scale(self.data),
                columns=self.data.columns,
                index=self.data.index
            )
        self.data.drop(['close'], axis=1, inplace=True)
        self.data['returns'] = rets  # not scale rets
        self.data = self.data.loc[:, ['returns'] + list(self.data.columns.drop('returns'))]
        self.data = self.data.dropna()
        log.info(self.data.info())

    def reset(self) -> NoReturn:
        """Provides starting index for time series and resets step"""
        high = len(self.data.index) - self.steps_per_episode
        self.offset = np.random.randint(low=0, high=high)
        self.step = 0

    def take_step(self) -> tuple:
        """Returns data for current trading day and done signal"""
        obs = self.data.iloc[self.offset + self.step].values
        self.step += 1
        done = self.step > self.steps_per_episode
        return obs, done
```

A.1.2 TradingSimulator class

```
import pandas as pd
import numpy as np


class TradingSimulator:
    """ Implements core trading simulator for single-instrument univ """

    def __init__(self, steps: int, trading_cost_bps: float, time_cost_bps: float):
        """
        :param steps: int, steps for episode
        :param trading_cost_bps: float, transaction-cost. Bid-Ask/2
        :param time_cost_bps: float, penalization (i.e. cost) when agents doesn't make any trade
        """
        # invariant for object life
        self.trading_cost_bps = trading_cost_bps
        self.time_cost_bps = time_cost_bps
        self.steps = steps
```

```
        # change every step
        self.step = 0
        self.actions = np.zeros(self.steps)
        self.navs = np.ones(self.steps)
        self.market_navs = np.ones(self.steps)
        self.strategy_returns = np.ones(self.steps)
        self.positions = np.zeros(self.steps)
        self.costs = np.zeros(self.steps)
        self.trades = np.zeros(self.steps)
        self.market_returns = np.zeros(self.steps)

    def reset(self) -> None:
        self.step = 0
        self.actions.fill(0)
        self.navs.fill(1)
        self.market_navs.fill(1)
        self.strategy_returns.fill(0)
        self.positions.fill(0)
        self.costs.fill(0)
        self.trades.fill(0)
        self.market_returns.fill(0)

    def take_step(self, action: int, market_return: float) -> tuple:
        """
        Calculates NAVs, trading costs and reward based on an action and latest market return and returns the
reward and
        a summary of the day's activity

        :param action: int,
        :param market_return: float
        """
        start_position = self.positions[max(0, self.step - 1)]
        start_nav = self.navs[max(0, self.step - 1)]
        start_market_nav = self.market_navs[max(0, self.step - 1)]
        self.market_returns[self.step] = market_return
        self.actions[self.step] = action

        end_position = action - 1  # short, neutral, long
        n_trades = end_position - start_position
        self.positions[self.step] = end_position
        self.trades[self.step] = n_trades

        # roughly value based since starting NAV = 1
        trade_costs = abs(n_trades) * self.trading_cost_bps
        time_cost = 0 if n_trades else self.time_cost_bps
        self.costs[self.step] = trade_costs + time_cost
        reward = start_position * market_return - self.costs[self.step]  # reward
        self.strategy_returns[self.step] = reward

        if self.step != 0:
            self.navs[self.step] = start_nav * (1 + self.strategy_returns[self.step])
            self.market_navs[self.step] = start_market_nav * (1 + self.market_returns[self.step])

        info = {
            'reward': reward,
            'nav'   : self.navs[self.step],
            'costs' : self.costs[self.step]
```

```
        }
        self.step += 1
        return reward, info

    def result(self) -> pd.DataFrame:
        """returns current state as pd.DataFrame """
        return pd.DataFrame(
            {
                'action'        : self.actions,  # current action
                'nav'           : self.navs,  # starting Net Asset Value (NAV)
                'market_nav'    : self.market_navs,
                'market_return'  : self.market_returns,
                'strategy_return': self.strategy_returns,
                'position'      : self.positions,  # eod position
                'cost'          : self.costs,  # eod costs
                'trade'         : self.trades  # eod trade)
            }
        )
```

A.1.3 TradingEnvironment class

```
import gym
from gym import spaces
from gym.utils import seeding

from DDQN_trading import DataSource, TradingSimulator

from typing import *


class TradingEnvironment(gym.Env):
    """
    A simple trading environment for RL.
    - Provides observations for a security price series
    - An episode is defined as a sequence of N steps with random start
    - Each 'step' allows the agent to choose one of three actions:
        - 0: SHORT
        - 1: HOLD
        - 2: LONG
    - Trading has an optional cost (default: 10bps) of the change in position value.
    - Going from short to long implies two trades.
    - Not trading also incurs a default time cost of 1bps per step.
    - An episode begins with a starting Net Asset Value (NAV) of 1 unit of cash.
    - If the NAV drops to 0, the episode ends with a loss.
    - If the NAV hits 2.0, the agent wins.
    - Trading simulator tracks a buy-and-hold strategy as benchmark.
    """
    # metadata = {'render.modes': ['human']}

    def __init__(
            self,
            steps_per_episode: int = 60 * 24,
            trading_cost_bps: float = 1e-3,
            time_cost_bps: float = 1e-4,
            ticker='EURUSD2022_1m',
```

```
        start_end: Optional[Tuple[str, str]] = None
    ):
        self.trading_days = steps_per_episode
        self.trading_cost_bps = trading_cost_bps
        self.ticker = ticker
        self.time_cost_bps = time_cost_bps
        self.data_source = DataSource(
            steps_per_episode=self.trading_days,
            ticker=ticker,
            start_end=start_end
        )
        self.simulator = TradingSimulator(
            steps=self.trading_days,
            trading_cost_bps=self.trading_cost_bps,
            time_cost_bps=self.time_cost_bps
        )
        self.action_space = spaces.Discrete(3)  # three actions: {0: HOLD, 1: BUY, 2: SELL}
        self.observation_space = spaces.Box(self.data_source.min_values.values,
self.data_source.max_values.values)   # to understand
        self.reset()

    def seed(self, seed=None) -> list:
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def step(self, action) -> tuple:
        """Returns state observation, reward, done and info"""
        assert self.action_space.contains(action), '{} {} invalid'.format(action, type(action))
        observation, done = self.data_source.take_step()
        reward, info = self.simulator.take_step(action=action, market_return=observation[0])
        return observation, reward, done, None, info

    def reset(self):
        """Resets DataSource and TradingSimulator; returns first observation"""
        self.data_source.reset()
        self.simulator.reset()
        return self.data_source.take_step()[0]
```

A.1.4 Agent class

```
import tensorflow as tf
import numpy as np
import random
from collections import deque

from typing import *


class DDQNAgent:
    def __init__(
        self,
        state_dim: int,
        num_actions: int,
        learning_rate: float,
```

```
        gamma: float,
        epsilon_start: float,
        epsilon_end: float,
        epsilon_decay_steps: int,
        epsilon_exponential_decay: float,
        replay_capacity: int,
        architecture: tuple,  # tuple of int, each int is the number of neurons for layers
        l2_reg,
        tau: int,
        batch_size: int
    ):
        self.state_dim = state_dim
        self.num_actions = num_actions
        self.experience = deque([], maxlen=replay_capacity)
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.architecture = architecture
        self.l2_reg = l2_reg

        self.online_network = self.build_model()
        self.target_network = self.build_model(trainable=False)
        self.update_target()

        self.epsilon = epsilon_start
        self.epsilon_decay_steps = epsilon_decay_steps
        self.epsilon_decay = (epsilon_start - epsilon_end) / epsilon_decay_steps
        self.epsilon_exponential_decay = epsilon_exponential_decay
        self.epsilon_history = []

        self.total_steps = 0
        self.train_steps = 0
        self.episodes = 0
        self.episode_length = 0
        self.train_episodes = 0
        self.steps_per_episode = []
        self.episode_reward = 0
        self.rewards_history = []

        self.batch_size = batch_size
        self.tau = tau
        self.losses = []
        self.idx = tf.range(batch_size)  # tf.range is equal to range() built-in python function
        self.train = True

    def build_model(self, trainable: bool = True):
        """
        Build an ANN model
        - number of layers is defined by self.architecture

        :param trainable: True for the online ANN and False for the target one
        :return: ANN model
        """
        layers = []
        # n = len(self.architecture)
        for i, units in enumerate(self.architecture, 1):
            layers.append(
                tf.keras.layers.Dense(
```

```
                units=units,
                input_dim=self.state_dim if i == 1 else None,
                activation='relu',
                kernel_regularizer=tf.keras.regularizers.l2(self.l2_reg),
                name=f'Dense_{i}',
                trainable=trainable
            )
        )
    layers.append(tf.keras.layers.Dropout(.1))
    layers.append(
        tf.keras.layers.Dense(
            units=self.num_actions,
            trainable=trainable,
            name='Output'
        )
    )
    model = tf.keras.models.Sequential(layers)
    model.compile(
        loss='mean_squared_error',
        optimizer=tf.keras.optimizers.Adam(learning_rate=self.learning_rate)
    )
    return model

def update_target(self) -> NoReturn:
    self.target_network.set_weights(self.online_network.get_weights())

def epsilon_greedy_policy(self, state: np.ndarray) -> int:
    self.total_steps += 1
    if np.random.rand() <= self.epsilon:
        return np.random.choice(self.num_actions)
    else:
        q = self.online_network.predict(state, verbose=False)
        return np.argmax(q, axis=1).squeeze()

def memorize_transition(
        self,
        s: np.ndarray,       # state_features (t)
        a: int,              # action taken (t)
        r: float,            # reward gained (t + 1)
        s_prime: np.ndarray,  # state_features (t + 1)
        not_done: float      # 0 or 1, i.e. like boolean, to understand if episode is finished
) -> NoReturn:

    if not_done:
        self.episode_reward += r  # increment episode's reward
        self.episode_length += 1  # increment episode's length
    else:
        if self.train:
            # decrease epsilon (i.e. exploration) until # of episodes is less than # epsilon decay steps
            if self.episodes < self.epsilon_decay_steps:
                self.epsilon -= self.epsilon_decay
            # otherwise decrease epsilon using an exponential approach
            else:
                self.epsilon *= self.epsilon_exponential_decay

        self.episodes += 1  # increment # of episodes when the episode is finished
        self.rewards_history.append(self.episode_reward) # store reward history
```

```
        # print("reward_history size:", round(sys.getsizeof(self.rewards_history) / 1e6, 3), "MB")
        self.steps_per_episode.append(self.episode_length)  # store episode's length
        # print("steps_per_episode size:", round(sys.getsizeof(self.steps_per_episode) / 1e6, 3), "MB")
        self.episode_reward, self.episode_length = 0, 0  # reset to 0 both episode's reward and episode's length

    self.experience.append((s, a, r, s_prime, not_done))  # store transition data (i.e. one step)

  def experience_replay(self) -> NoReturn:
    """
    It trains ANN (both target and online) using the experience replay approach
    """
    # Make experience replay approach only if the stored experienced has at least self.batch_size elements
    if self.batch_size > len(self.experience):
      pass
    else:
      start_ = perf_counter()
      # ----- It uses a vectorize approach, instead of a for loop ------------------------------------------------

      # create a minibatch, taking N (self.batch_size) experiences (from self.experience). Using random.sample
      # will return a list of list, then zipping them we will have a list of 5 tuple where each tuple will be a
      # "category" (state, action, ...). Then mapping them with function np.array, we will have the input for the model
      minibatch = map(np.array, zip(*random.sample(self.experience, self.batch_size)))
      states, actions, rewards, next_states, not_done = minibatch

      next_q_values = self.online_network.predict_on_batch(next_states)  # Q-value (t + 1) with ONLINE-ANN
      best_actions = tf.argmax(next_q_values, axis=1)  # best action (i.e. greedy-online) for t+1

      next_q_values_target = self.target_network.predict_on_batch(next_states)  # Q-value (t + 1) with TARGET-ANN
      # It's a way to choice between online-ann choice and target-ann choice. i.e. Double-DQN approach
      target_q_values = tf.gather_nd(
        next_q_values_target,
        # tf.stack: stacks a list of rank-R tensors into one rank-(R+1) tensor
        tf.stack(
          (
            self.idx,
            tf.cast(best_actions, tf.int32)  # tf.cast convert a type to another one (here to int32)
          ),
          axis=1
        )
      )
      # update q-value (t) using TARGET or ONLINE ANN. This is where we apply the DOUBLE-DQN approach
      targets = rewards + not_done * self.gamma * target_q_values

      q_values = self.online_network.predict_on_batch(states)  # current Q-value (t)
      q_values[self.idx, actions] = targets  # update q-values

      start = perf_counter()
      loss = self.online_network.train_on_batch(x=states, y=q_values)
      self.losses.append(loss)
```

```
        # update TARGET-ANN every self.tau steps. Better understand the code
        if self.total_steps % self.tau == 0:
            self.update_target()
```

A.1.5 main_train.py

```
from time import time, perf_counter
import sys

import numpy as np
import pandas as pd
import tensorflow as tf
import gym

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

from DDQN_trading import DDQNAgent, TradingEnvironment
from DDQN_trading.config import config
from DDQN_trading import utils

import warnings
for i in (UserWarning, FutureWarning, RuntimeWarning, DeprecationWarning):
    warnings.simplefilter("ignore", i)

# ---------------------------------------------------- Settings
np.random.seed(42)
tf.random.set_seed(42)
sns.set_style('whitegrid')

dt_str = utils.get_timestamp_for_file()
results_path = utils.get_results_path()
if not results_path.exists():
    results_path.mkdir(parents=True)

# ---------------------------------------------------- Setup Gym Environment
gym.register(
    id='trading-v0',
    entry_point='trading_environment:TradingEnvironment',
    max_episode_steps=config["steps_per_episode"]
)
print(f'Trading costs: {config["trading_cost_bps"]:.2%} | Time costs: {config["time_cost_bps"]:.2%}')
trading_environment = gym.make(
    'trading-v0',
    ticker=config["ticker"],
    steps_per_episode=config["steps_per_episode"],
    trading_cost_bps=config["trading_cost_bps"],
    time_cost_bps=config["time_cost_bps"],
    start_end=config["start_end"]
)
trading_environment.seed(42)

# ---------------------------------------------------- Get Environment Parameters
```

```
state_dim = trading_environment.observation_space.shape[0]
num_actions = trading_environment.action_space.n
max_episode_steps = trading_environment.spec.max_episode_steps

# ====================================================================================
# ---------------------------------------------- Create a DDQN-Agent ----------------------------------------------
# ====================================================================================
tf.keras.backend.clear_session()
ddqn = DDQNAgent(
    state_dim=state_dim,
    num_actions=num_actions,
    learning_rate=config["learning_rate"],
    gamma=config["gamma"],
    epsilon_start=config["epsilon_start"],
    epsilon_end=config["epsilon_end"],
    epsilon_decay_steps=config["epsilon_decay_steps"],
    epsilon_exponential_decay=config["epsilon_exponential_decay"],
    replay_capacity=config["replay_capacity"],
    architecture=config["architecture"],
    l2_reg=config["l2_reg"],
    tau=config["tau"],
    batch_size=config["batch_size"]
)
# ---------------------------------------------- print DDQN-Online-ANN architecture
print(ddqn.online_network.summary())

# ---------------------------------------------- Run-experiment

# ------------ Initialize variables
episode_time, navs, market_navs, diffs, episode_eps = [], [], [], [], []

# ---------------------------------------------- TRAIN AGENT
start = time()
results = []
for episode in range(1, config["n_episodes"] + 1):
    this_state = trading_environment.reset()  # reset to 0 the environment due to new episode was started
    # iterate over the episode's steps
    start_ = perf_counter()
    for episode_step in range(max_episode_steps):
        action = ddqn.epsilon_greedy_policy(this_state.reshape(-1, state_dim))  # get an action
        next_state, reward, done, _, __ = trading_environment.step(action)  # given the action get S', R(t+1)
and done

        ddqn.memorize_transition(s=this_state, a=action, r=reward, s_prime=next_state, not_done=0.0 if
done else 1.0)

        # if we have to train ANN, do the experience replay approach to update ANNs models
        if ddqn.train:
            # if experience has enough obs (>= batch_size) re-train ANN each_step! (update target-ANN each
tau steps)
            ddqn.experience_replay()
        if done:
            break
        this_state = next_state  # update current state with the next one

    # get DataFrame with sequence of actions, returns and nav values
    result = trading_environment.env.simulator.result()
```

```
    # get results of last step
    final = result.iloc[-1]

    # apply return (net of cost) of last action to last starting nav
    nav = final.nav * (1 + final.strategy_return)
    navs.append(nav)

    # market nav
    market_nav = final.market_nav
    market_navs.append(market_nav)

    # track difference between agent an market NAV results
    diff = nav - market_nav
    diffs.append(diff)

    # every 10 episode, print the temporary-results
    if episode % 10 == 0:
        utils.track_results(
            episode,
            # show mov. average results for 100 (10) periods
            np.mean(navs[-100:]),
            np.mean(navs[-10:]),
            np.mean(market_navs[-100:]),
            np.mean(market_navs[-10:]),
            # share of agent wins, defined as higher ending nav
            np.sum([s > 0 for s in diffs[-100:]]) / min(len(diffs), 100),
            time() - start,
            ddqn.epsilon
        )

    if len(diffs) > 25 and all([r > 0 for r in diffs[-25:]]):
        print(result.tail())
        break

trading_environment.close()

# ------------------------------------------------------------ STORE results
results = pd.DataFrame(
    {
        'n_episode': list(range(1, episode + 1)),
        'agent_nav': navs,
        'mkt_nav': market_navs,
        'delta': diffs
    }
).set_index('n_episode')

results['Strategy Wins (%)'] = (results["delta"] > 0).rolling(100).sum()
results.info()

utils.store_results(config=config, results=results, path=utils.get_results_path())

with sns.axes_style('white'):
    sns.distplot(results.delta)
    sns.despine()
plt.show()
```

```
results.info()

fig, axes = plt.subplots(ncols=2, figsize=(14, 4), sharey=True)

df1 = (results[['agent_nav', 'mkt_nav']]
      .sub(1)
      .rolling(100)
      .mean())
df1.plot(
   ax=axes[0],
   title='Annual Returns (Moving Average)',
   lw=1
)

df2 = results['Strategy Wins (%)'].div(100).rolling(50).mean()
df2.plot(
   ax=axes[1],
   title='Agent Outperformance (%, Moving Average)'
)

for ax in axes:
   ax.yaxis.set_major_formatter(
      FuncFormatter(lambda y, _: '{:.0%}'.format(y)))
   ax.xaxis.set_major_formatter(
      FuncFormatter(lambda x, _: '{:,.0f}'.format(x)))
axes[1].axhline(.5, ls='--', c='k', lw=1)

dt_str = utils.get_timestamp_for_file()
sns.despine()
fig.tight_layout()
fig.savefig(results_path / f'{dt_str}_performance', dpi=300)
plt.show()
```

A.1.6 Tester class

```
import numpy as np
import pandas as pd
from tensorflow import keras
from DDQN_trading import DataSource
from typing import *


class Tester:

   def __init__(self, model_path: str, ticker: str, start_end: Tuple[str, str]):
      self.model_path = model_path
      self.ticker = ticker
      self.start_end = start_end

      self.ann = self.get_model()
      self.test_data = self.get_test_data()
```

```
    def get_model(self) -> keras.models.Sequential:
        return keras.models.load_model(self.model_path)


    def get_test_data(self) -> pd.DataFrame:

        data_source = DataSource(
            ticker=self.ticker,
            normalize=True,
            start_end=None
        )
        return data_source.data.loc[self.start_end[0]: self.start_end[1]]


    def predict_actions(self) -> pd.Series:
        predictions = self.ann.predict_on_batch(self.test_data)
        actions = np.argmax(predictions, axis=1)
        return pd.Series(data=actions, index=self.test_data.index).sub(1)  # sub(1) because {0: short, 1:
neutral, 2: long}


    def get_test_returns(self, tc_bps: float = 0.0001) -> Tuple[pd.Series, pd.Series]:
        """
        Computes the returns of the strategy, using the weights of the ann for the estimation.
        """
        actions = self.predict_actions()
        asset_rets = self.test_data['returns']

        gross_rets = actions.shift(1).mul(asset_rets)
        tc = actions.diff().abs() * tc_bps
        net_rets = gross_rets.sub(tc)

        return  gross_rets, net_rets


    @staticmethod
    def get_cumulative_rets(rets: pd.Series, comp: bool = True) -> pd.Series:
        if comp:
            return (1 + rets).cumprod() - 1
        else:
            return rets.cumsum()
```

A.1.7 main_test.py

```
import os.path
import matplotlib as mpl

from DDQN_trading import Tester, Analyst, plot_final_graph


if __name__ == "__main__":

    import warnings
    warnings.simplefilter("ignore", FutureWarning)
    mpl.use('TkAgg')
```

```
model_name = "ddqn_target_ann.h5"
model_path = os.path.join(os.getcwd(), "results", "2023_02_19__10_52", model_name)
ticker = "AAPL_10y_1d"
start_end = ("2021-01-04", "2021-06-01")

tester = Tester(
    model_path=model_path,
    ticker=ticker,
    start_end=start_end
)

strategy_gross_rets, strategy_net_rets = tester.get_test_returns(tc_bps=0.0025)
strategy_gross_cumrets = tester.get_cumulative_rets(strategy_gross_rets, comp=True)
strategy_net_cumrets = tester.get_cumulative_rets(strategy_net_rets, comp=True)
plot_final_graph(
    strategy_gross_cumrets,
    strategy_net_cumrets,
    tester.get_cumulative_rets(tester.test_data['returns'], comp=True),
    tester.predict_actions()
)

analyst = Analyst(
    strategy_gross_rets=strategy_gross_rets,
    strategy_net_rets=strategy_net_rets,
    benchmark_rets=tester.test_data['returns']
)
performance_analysis = analyst.get_performance_analysis()

print(performance_analysis)
```

A.1.8 utils.py

```
import datetime as dt
from pathlib import Path

import pandas as pd


def format_time(t):
    m_, s = divmod(t, 60)
    h, m = divmod(m_, 60)
    return '{:02.0f}:{:02.0f}:{:02.0f}'.format(h, m, s)


def track_results(
        episode: int,
        nav_ma_100: float,
        nav_ma_10: float,
        market_nav_100: float,
        market_nav_10: float,
        win_ratio: float,
        total: float,
        epsilon: float
):
    # time_ma = np.mean([episode_time[-100:]])
    # T = np.sum(episode_time)
```

```
    template = '{:>4d} | {} | {} | Agent: MA(100):{:>6.1%}; MA(10):{:>6.1%} | '
    template += 'Market: MA(100):{:>6.1%}; MA(10):{:>6.1%} | '
    template += 'Wins: {:>5.1%} | eps: {:>6.3f}'
    print(template.format(
        episode,
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        format_time(total),
        nav_ma_100 - 1,
        nav_ma_10 - 1,
        market_nav_100 - 1,
        market_nav_10 - 1,
        win_ratio,
        epsilon)
    )


def get_timestamp_for_file() -> str:
    return str(dt.datetime.now())[:16].replace("-", "_").replace(" ", "__").replace(":", "_")


def store_results(config: dict, results: pd.DataFrame, path) -> None:
    dt_str = get_timestamp_for_file()
    writer = pd.ExcelWriter(path / f'{dt_str}_results.xlsx', engine='xlsxwriter')
    pd.Series(config).to_excel(writer, sheet_name="configuration")
    results.to_excel(writer, sheet_name="results")
    results.to_csv(path / f'{dt_str}_results.xlsx')
    writer.close()


def get_results_path():
    return Path('results', get_timestamp_for_file())
```

A.1.9 Analyst class

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np


class Analyst():

    def __init__(self, strategy_gross_rets: pd.Series, strategy_net_rets: pd.Series, benchmark_rets:
pd.Series):

        self.strategy_gross_rets = strategy_gross_rets
        self.strategy_net_rets = strategy_net_rets
        self.benchmark_rets = benchmark_rets

    @staticmethod
    def _calc_total_return(rets: pd.Series) -> float:
        return rets.add(1).cumprod().sub(1).iloc[-1]

    @staticmethod
    def _calc_avg_ann_rets(rets: pd.Series) -> float:
```

```
    """
    Assuming daily returns
    """
    return rets.mean() * 252

@staticmethod
def _calc_avg_ann_volatility(rets: pd.Series) -> float:
    """
    Assuming daily returns
    """
    return rets.std() * np.sqrt(252)

@staticmethod
def _calc_max_dd(rets: pd.Series) -> float:
    equity_curve = (1 + rets).cumprod()
    return float((equity_curve / equity_curve.cummax() - 1).min())

@staticmethod
def _calc_ann_semivolatility(rets: pd.Series) -> float:
    """
    Assuming daily returns
    """
    return float(rets[rets < 0].std() * np.sqrt(252))

@staticmethod
def _calc_daily_skew(rets: pd.Series) -> float:
    return rets.skew()

@staticmethod
def _calc_daily_kurt(rets: pd.Series) -> float:
    return rets.kurt()

@staticmethod
def _calc_ann_sharpe(rets: pd.Series) -> float:
    """
    Assuming daily returns
    """
    return (rets.mean() / rets.std()) * np.sqrt(252)

@classmethod
def _calc_ann_sortino(cls, rets: pd.Series) -> float:
    """
    Assuming daily returns
    """
    return (rets.mean() * 252) / cls._calc_ann_semivolatility(rets)

@staticmethod
def _calc_best_day(rets: pd.Series) -> float:
    return float(rets.max())

@staticmethod
def _calc_worst_day(rets: pd.Series) -> float:
    return float(rets.min())

@staticmethod
def _calc_var(rets: pd.Series, parametric: bool, var_confidence: float = 0.99) -> float:
    if parametric:
```

```
            return -rets.std() * 2.33
        else:
            return rets.quantile(q=1-var_confidence, interpolation='linear')

    def get_performance_analysis(self):
        results = []
        for series in (self.strategy_gross_rets, self.strategy_net_rets, self.benchmark_rets):
            tmp_res = pd.Series(
                {
                    'Total Return':           self._calc_total_return(series),
                    'Avg Ann. Return':        self._calc_avg_ann_rets(series),
                    'Avg Ann. Volatility':    self._calc_avg_ann_volatility(series),
                    'Avg Ann. Semi-Volatility': self._calc_ann_semivolatility(series),
                    'Max Drawdown':           self._calc_max_dd(series),
                    'VaR(99%) Parametric':    self._calc_var(series, parametric=True),
                    'VaR(99%) Not Parametric': self._calc_var(series, parametric=False, var_confidence=0.99),
                    'Daily Skewness':         self._calc_daily_skew(series),
                    'Daily Kurtosis':         self._calc_daily_kurt(series),
                    'Ann. Sharpe':            self._calc_ann_sharpe(series),
                    'Ann. Sortino':           self._calc_ann_sortino(series),
                    'Best Day':               self._calc_best_day(series),
                    'Worst Day':              self._calc_worst_day(series)
                }
            )
            results.append(tmp_res)

        return pd.concat(results, axis=1, keys=['strategy (gross)', 'strategy (nt)', 'benchmark'])


def plot_final_graph(
        strategy_gross_cumrets: pd.Series,
        strategy_net_cumrets: pd.Series,
        benchmark_cumrets: pd.Series,
        actions: pd.Series
) -> None:

    fig, axs = plt.subplots(nrows=2)
    plt.suptitle('DDQN-strategy VS Benchmark (long-only)', fontsize=16, fontweight='bold')

    axs[0].plot(strategy_gross_cumrets)
    axs[0].plot(strategy_net_cumrets)
    axs[0].plot(benchmark_cumrets, color='green')
    axs[0].grid(linestyle='--', color='silver')
    axs[0].set_ylabel('Cumulative Returns (compounded)', fontweight='bold')
    axs[0].legend(['DDQN-strategy (Gross)', 'DDQN-strategy (Net)', 'Long-Only'])

    actions = actions.copy(deep=True)
    axs[1].step(x=actions.index, y=actions.values, color='black', linewidth=1)
    axs[1].legend(['position'])
    axs[1].grid(linestyle='--', color='silver')
    axs[1].set_ylabel('{-1: short, 0: neutral, 1: long}', fontweight='bold')
    axx = axs[1].twinx()
    axx.plot(benchmark_cumrets, color='green')
    axx.set_ylabel('Price path', fontweight='bold')

    plt.show()
```

A.2 features_engineering folder

Nei successive sotto-paragrafi si riporta il codice sorgente relativo al folder "features_engineering", che contiene i moduli python relativi alla costruzione del set di *features*, che solo in parte viene utilizzato durante l'allenamento del modello di DDQN. Al fine di riportare solo la parte *core* del codice sorgente relativo al presente folder, sono stati esclusi i seguenti moduli python:

- main.py: che permette di creare un set di features, dato un dataset di prezzi come input, sfruttando la classe FeaturesEngineering definita nel modulo features_engineering.py
- utils.py: vuole essere una libreria di c.d. *utils* che sono di supporto alla libreria FeaturesEngineering

A.2.1 FeaturesEngineering class

```
import pandas as pd
import numpy as np
import talib as ta
import hurst
from typing import Tuple, Union, Optional

# from aot import aot_module as aot
from features_engineering import Utils

"""
Migliorie & Commenti:
- velocizzare Hurst (troppo lento ora)
"""


class Features(object):

    class Overlap:

        @staticmethod
        def sma(s: pd.Series, timeperiod: int) -> pd.Series:
            """
            Returns Simple Moving Average (SMA)

            :param s: pd.Series
            :param timeperiod: int, window
            :return: pd.Series
            """
            Utils.check_min_obs(s, min_len=timeperiod)
            return ta.SMA(s, timeperiod)

        @staticmethod
        def ema(s: pd.Series, timeperiod: int) -> pd.Series:
            """
            Returns Exponential Moving Average (EMA)
```

```
        :param s: pd.Series
        :param timeperiod: int, window
        :return: pd.Series
        """
        Utils.check_min_obs(s, min_len=timeperiod)
        return ta.EMA(s, timeperiod)

    @staticmethod
    def rsi(s: pd.Series, timeperiod: int) -> pd.Series:
        """
        Calc Relative Strength Index (RSI)

        :param s: pd.Series, closing price
        :param timeperiod: int, window
        :return: pd.Series
        """
        Utils.check_min_obs(s, min_len=timeperiod + 1)
        return ta.RSI(s, timeperiod)

    @staticmethod
    def bollinger(s: pd.Series, timeperiod: int, ndevup: float, ndevdn: float) -> pd.DataFrame:
        """
        Return the Bollinger Bands.

        :param s: pd.Series, prices
        :param timeperiod: int, window
        :param ndevup: number of std for the upper band
        :param ndevdown: number of std for the lower band
        :return: pd.DataFrame with columns ['uband', 'mband', 'lband']
        """
        output = pd.concat(ta.BBANDS(s, timeperiod, ndevup, ndevdn), axis=1)
        output.columns = ['uband', 'mband', 'lband']
        return output

class Vola:

    @staticmethod
    def vola_sma(s: pd.Series, timeperiod: int, on_rets: bool) -> pd.Series:
        """
        Returns Rolling-Volatility using Simple Moving Average (SMA)

        :param s: pd.Series, prices
        :param timeperiod: int, window
        :param on_rets: bool, True if you want to calc vola on returns, otherwise on the prices
        :return: pd.Series
        """
        Utils.check_min_obs(s, min_len=timeperiod + 1 if on_rets else timeperiod)
        if on_rets:
            tgt = s.pct_change()
        else:
            tgt = s
        return tgt.rolling(timeperiod).std()

    @staticmethod
    def vola_ema(s: pd.Series, timeperiod: int, on_rets: bool) -> pd.Series:
        """
        Returns Rolling-Volatility using Exponential Moving Average (EMA)
```

```
        :param s: pd.Series, prices
        :param timeperiod: int, window
        :param on_rets: bool, True if you want to calc vola on returns, otherwise on the prices
        :return: pd.Series
        """
        Utils.check_min_obs(s, min_len=timeperiod + 1 if on_rets else timeperiod)
        if on_rets:
            s_arr = Utils.from_series_to_numpy(s.pct_change())
        else:
            s_arr = Utils.from_series_to_numpy(s)

        ema = np.repeat(np.nan, repeats=len(s_arr))

        for t in range(timeperiod, len(s_arr) + 1):

            tgt = s_arr[t - timeperiod: t]
            tmp_ema = ta.EMA(tgt, timeperiod)

            ema[t - 1] = tmp_ema[-1]

        return pd.Series(s_arr, index=s.index)

    @staticmethod
    def atr(data: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Returns the Average-True-Range (ATR)

        :param data: pd.DataFrame, with at least columns: ['high', 'low', 'close']
        :param timeperiod: int, window
        :return: pd.Series
        """
        Utils.check_min_obs(data, min_len=timeperiod + 1)
        return ta.ATR(data['high'], data['low'], data['close'], timeperiod=timeperiod)

    @staticmethod
    def bars_dispersion(data: pd.DataFrame) -> pd.Series:
        """
        It returns the dispersion of the bars. It's computed as follow:
            - (HIGH - LOW) / LOW
        Thus, it's bounded between 0 and 1 --> [0, 1]

        :param data: pd.DataFrame, with at least the columns: ['high', 'low']
        :return: pd.Series
        """
        return (data['high'] - data['low']) / data['low']

    @classmethod
    def bars_dispersion_rolling(cls, data: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Returns SMA of the dispersion of the bars. See FeaturesEngineering.bars_dispersion func's docs for
more details.

        :param data: pd.DataFrame, with at least the columns: ['high', 'low']
        :param timeperiod: int, window for SMA
        :return: pd.Series
        """
```

```
        Utils.check_min_obs(data, min_len=timeperiod)

        bars_dispersion =  cls.bars_dispersion(data)
        if timeperiod == 1:
            return bars_dispersion
        else:
            return Features.Overlap.sma(bars_dispersion, timeperiod=timeperiod)

    @staticmethod
    def hurst_exp(s_price: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Returns Hurst Exponent Indicator. It detects if the series shows persistent pattern (i.e. trend period)
or
        anti-persistent one (i.e. mean-reverting period). For further references see:
            - https://towardsdatascience.com/introduction-to-the-hurst-exponent-with-code-in-python-
4da0414ca52e
            - https://en.wikipedia.org/wiki/Hurst_exponent

        For computing Hurst Exponent exist several formulas. This function use the R/S procedure.

        WARNING.1: Due to a constraint from the formula, minimum obs required are 100

        :param s_price: pd.Series, closing_price
        :param timeperiod: int, window
        :return: pd.Series
        """
        if timeperiod < 100:
            raise Exception("Due to a Hurst constraint, timeperiod cannot be less then 100")
        Utils.check_min_obs(s_price, min_len=100 if timeperiod < 100 else timeperiod)
        return  s_price.rolling(timeperiod).apply(lambda s: hurst.compute_Hc(s, kind='price',
simplified=False)[0])

class Momentum:

    @staticmethod
    def macd(s: pd.Series, fastperiod: int, slowperiod: int, signalperiod: int) -> pd.Series:
        """
        Return macd, macdsignal, macdhist
        """
        return pd.concat(ta.MACD(s, fastperiod, slowperiod, signalperiod), axis=1)[2].rename('MACD')

    @staticmethod
    def cross_sma_perc_distance(s_price: pd.Series, lookback: Tuple[int, int]) -> float:
        """
        Returns the percentage distance between a shorter SMA and a longest one, computed on the
closing prices.
        It's computed as follow:
              Shorter_SMA / Longer_SMA - 1

        It should be useful to feed some ML models. It's a market features that trying to suggest if we're in a
bullish
        market (shorter > longer) or into a bear market (longer < shorter), and the magnitude of the trend
(this is the
        reason why it returns a distance percentage, instead of a simple dummy {1: bullish; 0: bearish).

        :param data:, pd.DataFrame, with at least closing prices (columns named 'close')
        :param lookback:, Tuple[int, int], windows for SMA. First elements will be assigned to the shorter
```

```
SMA
        :return: float
        """
        Utils.check_min_obs(s_price, min_len=lookback[1])
        short_ma = Features.Overlap.sma(s_price, timeperiod=lookback[0])
        long_ma  = Features.Overlap.sma(s_price, timeperiod=lookback[1])
        return short_ma / long_ma - 1

    @staticmethod
    def roc(s: pd.Series, timeperiod: int) -> pd.Series:
        """
        Calc Return Of Change (ROC)

        :param s: pd.Series, closing price
        :param timeperiod: int, window
        :return: pd.Series
        """
        Utils.check_min_obs(s, min_len=timeperiod + 1)
        return ta.ROCP(s, timeperiod)

    @staticmethod
    def adx(data: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Return the Average-Directional-Index (ADX)

        :param data: pd.DataFrame, with at least columns ['high', 'low', 'close']
        :param timeperiod: int
        :return: pd.Series
        """
        return ta.ADX(data.high, data.low, data.close, timeperiod)

    @staticmethod
    def plus_di(data: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Return the +DI

        :param data: pd.dataFrame, with at least columns ['high', 'low', 'close']
        :param timeperiod: int
        :return: pd.Series
        """
        return ta.PLUS_DI(data.high, data.low, data.close, timeperiod=timeperiod)

    @staticmethod
    def minus_di(data: pd.DataFrame, timeperiod: int) -> pd.Series:
        """
        Return the -DI

        :param data: pd.dataFrame, with at least columns ['high', 'low', 'close']
        :param timeperiod: int
        :return: pd.Series
        """
        return ta.MINUS_DI(data.high, data.low, data.close, timeperiod=timeperiod)

class Others:

    @staticmethod
    def returns(s: pd.Series, lags: Optional[Union[int, list]] = None) -> Union[pd.Series, pd.DataFrame]:
```

```
        """
        It calc price returns with no lag and with them, depending on the parameter lags

        :param s: pd.Series, prices
        :param lags: Optional[Union[int, list]], type:int if you want all the lags between 0 and lags. type: list
                                        if you want to specify the lags period.
        """
        if not isinstance(s, pd.Series):
            raise Exception("s must be a pandas.Series object")
        rets = s.pct_change()
        if lags:
            if isinstance(lags, int):
                lags = list(range(lags + 1))

            if len(rets.dropna()) < max(lags):
                raise Exception("Observations must be at least equal to max(lags) + 1")

            output = pd.concat([rets.shift(n) for n in lags], axis=1)
            output.columns = [f'lag.{n}' for n in lags]
            return output
        else:
            return rets

    class Dummy:

        @staticmethod
        def extreme_events(s: pd.Series, std_threshold: float, std_window: int, diff: bool = False) -> pd.Series:
            """
            Returns a dummy series {-1, 0, 1} where there was an extreme event, i.e. a price movement greater
    than N
            standard deviation. It might be use as a reversal signal

            :param s: pd.Series, prices
            :param std_threshold: float, number of standard deviations to identify the extreme event
            :param std_window: int, window for building the rolling standard deviation threshold
            :param diff: bool, True to use differences for price movement, False for percentage returns.
            :return: pd.Series with {-1: extreme-negative event; 0: no extreme event; 1: extreme-positive event}
            """
            if diff:
                delta = s.diff()
            else:
                delta = s.pct_change()

            s_roll_std = delta.rolling(std_window).std() * std_threshold

            output = delta.\
                mask(delta >  s_roll_std,  1).\
                mask(delta < -s_roll_std, -1).\
                mask((delta <= s_roll_std) & (delta >= -s_roll_std), 0)
            output.iloc[:std_window] = np.nan

            return output
```

A.2.2 fe_generator.py

```
from features_engineering import Features

from typing import *


def generate_features_dataframe(dataset: Union[pd.Series, pd.DataFrame], parametrization: dict) ->
pd.DataFrame:

    fe = {}

    for k,v in parametrization.items():
        func = v['func']
        tgt_cols = v['tgt_cols'] if len(v['tgt_cols']) > 1 else v['tgt_cols'][0]
        tmp_data = dataset[tgt_cols]
        kwargs = v['kwargs']

        fe[k] = func(tmp_data, **kwargs)
    fe_df = pd.concat(fe, axis=1)
    cols = [f"{a}.{b}" for a, b in fe_df.columns]
    fe_df.columns = [
        col.split(".")[0]
        if flag == False else col
        for col, flag
        in zip(
            cols,
            fe_df.columns.get_level_values(0).duplicated(keep=False)
        )
    ]
    return fe_df
```

Bibliografia

- John C. Hull, *Machine Learning in Business "Un'introduzione alla Scienza dei Dati"*, Seconda Edizione, 2019.

- Stefan Jensen, "*Machine Learning for Algorithmic Trading*", Second Edition, 2020

- S.Sutton and G.Barto, "*Reinforcment Learning, an introduction*", 2nd, 2020

Sitografia

- www.medium.com
- www.towardatascience.com
- www.investopedia.com

Indice delle Figure