
Abstract

This paper aims to propose a comprehensive framework for the simulation and automatic evaluation of trading strategies on synthetic markets. In the first part, we define three stochastic models (Geometric Brownian Motion, Ornstein–Uhlenbeck, logistic map) for generating price series, then we implement a modular Gym-type environment that supports both discrete and fractional actions, with commissions and minimum holding periods.

In this virtual environment, we compare four families of agents: a random behavior agent, heuristic agents based on price thresholds, two agents based on neural networks, namely an agent based on the REINFORCE algorithm and an agent based on the PPO algorithm. Two main scripts are used to organize the experiments, one for the initial comparison and one for the robustness analysis, and the results are presented in the form of portfolio equity curves, final bar charts, and statistical tables (means and standard deviations). While the REINFORCE algorithm offers good but more variable returns, the results show that PPO agents perform better, especially the "expert" variant, which combines high returns with stability.

The paper highlights the importance of reproducibility (seed fixing) and proposes extending it to real data, increasing the number of episodes, and integrating prediction modules for a hybrid approach.

Cuprins

Cuprins	1
Lista Figurilor	4
1 Introducere	5
1.1 Descrierea aplicatiei	5
1.2 Motivatie	6
1.3 Obiective	6
1.4 Structura lucrării	7
2 Fundamente teoretice și tehnologii utilizate	8
2.1 Învățare automată (Machine learning)	8
2.2 Rețele neuronale	9
2.3 Python	11
2.3.1 Programarea orientată pe obiecte	11
2.3.2 PyTorch	12
2.3.3 Biblioteca Gym	12
2.3.4 Biblioteca NumPy	13
2.3.5 Biblioteca Pandas	14

2.3.6	Biblioteca Matplotlib	14
2.4	CSV (Comma separated values)	15
2.5	Modele stocastice de generare a prețurilor	15
2.5.1	Mișcarea geometrică browniana(GBM)	16
2.5.2	Procesul Ornstein–Uhlenbeck	16
2.5.3	Harta logistică haotică	17
2.6	Modelul REINFORCE	17
2.7	Optimizarea politii proximale (PPO)	18
3	Implementarea aplicației	20
3.1	Logica de funcționare	20
3.2	Reproductibilitate și configurare	21
3.3	Manipularea datelor	22
3.3.1	Clasa de bază a pieței	22
3.3.2	Piață discretă	23
3.3.3	Piață fracționară	24
3.4	Agenții virtuali	25
3.4.1	Agentul aleator	25
3.4.2	Agentul bazat pe o regulă	25
3.4.3	Agentul bazat pe rețele neuronale	26
4	Orchestrarea experimentelor și rezultate	31
4.1	Algoritmi utilizați	31
4.2	Vizualizarea rezultatelor	34
4.2.1	Curbele de echitate (equity curves)	34
4.2.2	Comparația finală a valorilor de portofoliu	35
4.2.3	Curbe de echitate per seed	35
4.3	Analiză statistică	36

4.3.1	Tabel pivot per seed	36
4.3.2	Medii și deviații standard	37
4.3.3	Interpretarea statistică	37
4.3.4	Performanța relativă a agentilor	38
4.3.5	Compromisul dintre medie și variabilitate	38
4.3.6	Impactul parametrilor: seed și număr de episoade	39
5	Concluzii	40
A	Anexa 1	41
A.1	Curbe de etichetare per seed	41
Referințe bibliografice		46

Lista Figurilor

3.1 Logica de funcționare a aplicației	20
4.1 Evoluția portofoliilor pentru toti agentii pe univers complet de 500 de zile.	34
4.2 Valoarea finală a portofoliului după 500 de pași pentru fiecare agent.	35
4.3 Evoluția portofoliului pentru agentii PPO-E și RN-I la seed = 42.	36
A.1 Seed = 0	41
A.2 Seed = 7	42
A.3 Seed = 5	42
A.4 Seed = 9	42
A.5 Seed = 10	43
A.6 Seed = 14	43
A.7 Seed = 42	43
A.8 Seed = 51	44
A.9 Seed = 68	44
A.10 Seed = 87	44
A.11 Seed = 97	45
A.12 Seed = 123	45
A.13 Seed = 999	45

Capitolul 1

Introducere

1.1 Descrierea aplicației

În mediul finanțiar actual, aflat în continuă schimbare, este mai important ca niciodată să înțelegem modul în care dinamicile diferite ale pieței influențează strategiile de tranzacționare. Această aplicație oferă un mediu de simulare versatil, în care utilizatorii pot genera serii sintetice de prețuri ale acțiunilor folosind trei modele distincte: mișcarea browniană geometrică (GBM), care surprinde drift-ul și volatilitatea clasice a randamentelor acțiunilor, procesul Ornstein-Uhlenbeck (OU), folosit adesea pentru a modela activele cu revenire la medie, cum ar fi ratele dobânzilor sau răspândirea mărfurilor, și o abordare de tip hartă haotică, care explorează dinamici neliniare, deterministe, dar imprevizibile. Prin comutarea între aceste generatoare de date, agenții și cercetătorii pot studia performanța într-o gamă largă de condiții teoretice de piață.

Pe lângă această piață simulată, aplicația utilizează patru agenți de tranzacționare autonomi. Un agent cu comportament aleatoriu care plasează ordine de cumpărare și vânzare fără nicio rațiune de piață, servind ca un control pentru a evalua eficacitatea strategiei. Apoi, un agent bazat pe reguli euristici simple. În cele din urmă, doi agenți bazați pe rețele neuronale care utilizează tehnici moderne de învățare automată: unul antrenat prin învățare clasice prin întărire, care învață să maximizeze randamentele pe termen lung prin încercări și erori, iar altul utilizând optimizarea politicii proximale (PPO), o metodă de ultimă generație de gradient de politică care echilibrează explorarea și stabilitatea pentru a obține performanțe robuste.

Împreună, aceste module permit utilizatorilor să compare modul în care modelele diferite de formare a prețurilor interacționează cu strategii de tranzacționare naive și sofisticate.

1.2 Motivatie

Am ales această temă, deoarece tranzacționarea automată și algoritmii de învățare automată au câștigat rapid teren în domeniul finanțier, oferind posibilitatea de a analiza volume mari de date și de a executa strategii complexe în timp real. Cu toate acestea, accesul la piețe reale presupune costuri semnificative și riscuri de execuție greu de controlat pentru experimente repetabile. De aceea, există o cerere crescută pentru un cadru de cercetare care să simuleze condițiile de piață într-un "univers de jucării" flexibil, unde putem compara și calibra algoritmi de tranzacționare fără a depinde de date sau infrastructuri costisitoare online.

1.3 Obiective

Această lucrare își propune să construiască și să valideze un pipeline complet pentru generarea unor serii sintetice de prețuri pe baza modelelor stocastice (GBM, procesul OU și harta logistică), implementarea unui mediu în stil Gym modular, cu opțiuni discrete și fractionare, taxe de tranzacție și cooldown, dezvoltarea și compararea a patru familii de agenți de tranzacționare și anume comportament aleator, bazat pe reguli, REINFORCE și PPO, pe același set de date sintetice și evaluarea robustă a performanțelor prin rulări multiple cu diverse seed-uri și analiza trade-off-ului medie vs. variabilitate.

1.4 Structura lucrării

Lucrarea este organizată în cinci capituloare principale, fiecare abordând o componentă esențială a pipeline-ului de simulare și evaluare a agentilor:

- ▷ **Capitolul 1 – Tehnologii și fundamente teoretice:** în care este prezentat limbajul Python și bibliotecile folosite (NumPy, Pandas, Matplotlib, PyTorch, Gym), apoi dezvoltăm modelele matematice pentru generarea de serii sintetice (GBM, OU, harta logistică) și algoritmii de tip policy-gradient (REINFORCE și PPO).
- ▷ **Capitolul 2 – Implementarea aplicăției:** detaliem arhitectura generală a sistemului (Logica de funcționare), configurarea mediului și reproducibilitatea experimentelor, manipularea datelor în mediul Gym (clasa de bază, părțile discrete și fracționale) și definirea agentilor (RandomAgent, RuleBased, NNAgent și PPOAgent).
- ▷ **Capitolul 3 – Orchestrarea experimentelor și rezultate:** descriem scripturile de rulare (main_comparison și seed_sweep), prezentăm graficele de curbe de capital propriu și tabelele cu bare comparative, precum și analiza statistică a rezultatelor (tabele pivot, medii și deviații standard).
- ▷ **Capitolul 4 – Discuții și concluzii preliminare:** sintetizăm performanța relativă a agentilor, discutăm compromisurile medie vs. variabilitate și impactul parametrilor (seed, număr de episoade) asupra stabilității și randamentului.
- ▷ **Capitolul 5 – Concluzii și direcții viitoare:** recapitulează contribuțiile principale, evidențiază limitările implementării curente și propune îmbunătățiri pentru extinderea cadrului (mai multe episoade, date reale, algoritmi avansați).

Capitolul 2

Fundamente teoretice și tehnologii utilizate

2.1 Învățare automată (Machine learning)

¹ În ultimii ani, termenul de învățare automată a devenit un cuvânt la modă, fiind prezent peste tot, de la titlurile din domeniul tehnologiei la anunțurile de angajare și chiar în conversațiile ocazionale despre asistenții inteligenți, mașinile care se conduc singure și recomandările personalizate. Dar, dincolo de agitație, despre ce este vorba cu adevărat despre învățarea automată? În esență să, învățarea automată este o metodă de a învăța computerele să învețe din date să recunoască tipare, să ia decizii și să își îmbunătățească performanța în timp, toate acestea fără a fi programate în mod explicit pentru fiecare sarcină specifică. În mod tradițional, dacă doriți ca un computer să rezolve o problemă, ar trebui să scrieți un set detaliat de instrucțiuni, un algoritm, pentru a-i spune exact ce să facă în fiecare situație posibilă. Acest lucru funcționează bine atunci când sarcina este bine definită și regulile sunt clare. Dar cum rămâne cu sarcinile care sunt prea complexe pentru regulile manuale, cum ar fi recunoașterea unui chip într-o mulțime, traducerea unei propoziții într-o altă limbă sau detectarea fraudei în milioane de tranzacții cu carduri de credit? Aici intervine învățarea automată. În loc să îi spunem computerului cum să rezolve problema, îi dăm exemple de probleme și îl lăsăm să își dea seama singur de reguli.[1]

Cum funcționează? Imaginea-vă că învățați un copil să recunoască merele. Nu îi înmânați o listă de verificare a proprietăților merelor ci, în schimb, îi arătați o mulțime de mere. Mari, mici, roșii, verzi, strălucitoare, cu vânătăi. În timp, copilul începe să recunoască singur merele, chiar dacă nu a mai văzut exact unul înainte. Sistemele de învățare automată funcționează în același mod. Le oferim date, împreună cu un anumit context cum ar fi etichete sau rezultate dorite, iar acestea încep să învețe modele care leagă intrările de ieșiri.

¹ Machine Learning introducere, https://books.google.ro/books?hl=ro&lr=&id=uZnSDwAAQBAJ&oi=fnd&pg=PR7&dq=machine+learning+introduction&ots=xOuVpwEqzQ&sig=lprbDZXgI_Q9pe5a115TnmTb5FA&redir_esc=y#v=onepage&q&f=false

Cele trei tipuri principale de învățare automată

1. Învățarea supervizată: Aceasta este cel mai comun tip. Sistemul este antrenat pe un set de date care include atât intrările, cât și ieșirile corecte (etichetele). De exemplu, învățarea unui model să recunoască cifrele scrise de mână prin prezentarea a mii de exemple etichetate. Modelul învăță să asocieze intrările cu ieșirile.
2. Învățarea nesupervizată: În acest caz, sistemul primește date fără etichete. Aceasta încearcă să găsească singur modele sau grupări ascunse. Un caz tipic de utilizare este segmentarea clientilor, în care o întreprindere dorește să descopere grupări naturale în comportamentul clientilor.
3. Învățarea prin consolidare: În această configurație, sistemul învăță prin încercări și erori, primind feedback sub formă de recompense sau penalizări. Gândiți-vă la un robot care învăță să meargă sau la un AI care joacă un joc video - încearcă acțiuni, vede ce funcționează și se îmbunătățește în timp.

2.2 Rețele neuronale

² Termenul de rețele neuronale poartă o greutate mare, îți provoacă imagini de mașini cu creier sau inteligență artificială, poate chiar nuante de science fiction, deși există un sămbure de adevăr în comparația cu creierul uman, rețelele neuronale se intersectează și cu domenii precum ingineria, matematica și informatica. Pentru a începe, haideți să analizăm o definiție de lucru a ceea ce înțelegem printr-o rețea neuronală și apoi să descompunem câțiva termeni cheie pentru a ne consolida noțiunea de rețea neuronală.

În esență să, o rețea neuronală este o rețea de unități simple de procesare adesea numite noduri sau neuroni care se inspiră din comportamentul neuronilor biologici. Inteligența rețelei nu constă în nicio unitate individuală, ci mai degrabă în puterea conexiunilor dintre ele, adesea denumite ponderi. Aceste ponderi nu sunt fixe ele sunt învățate și adaptate în timp prin antrenarea rețelei cu date de exemplu.

Pentru a putea aprecia rădăcinile biologice ale acestui concept, să facem o scurtă deviere bazele neuroștiinței. Creierul uman conține aproximativ 100 de miliarde de neuroni. Acești neuroni comunică prin impulsuri electrice scurte, sau vârfuri, care traversează membrana celulară. Conexiunile dintre neuroni au loc la joncțiuni specializate numite sinapse, care se află pe structuri ramificate cunoscute sub numele de dendrite. Fiecare neuron primește de obicei informații de la mii de alți neuroni. Aceste semnale sunt combinate în corpul celulei, iar dacă semnalul total depășește un anumit prag, neuronul trage, trimițând un impuls pe axonul său către alți neuroni. Semnalele pot fi stimulatoare, încurajând neuronii să tragă, sau inhibitoare, descurajându-i. Echilibrul și puterea acestor conexiuni determină comportamentul unic al neuronului. Acest stil distribuit și interconectat de procesare este cel pe care

² Rețele neuronale, <https://www.taylorfrancis.com/books/mono/10.1201/9781315273570/introduction-neural-networks-kevin-gurney>

rețelele neuronale artificiale încearcă să îl imite. Deoarece accentul este pus pe relațiile dintre unități, această abordare este adesea denumită conexionism, iar rețelele neuronale sunt uneori numite sisteme conexioniste în special în contextul modelării proceselor cognitive umane.

Când vorbim despre o rețea, ne putem referi la orice de la un singur nod, la o rețea complexă de mai multe noduri, toate interconectate. Ideea esențială este însă aceeași: structura învăță din date. În sistemele biologice, puterea sinaptică se poate modifica în funcție de experiență. În mod similar, în rețelele artificiale, ponderile sunt ajustate prin antrenare, un proces de prezentare a unor exemple, de modificare a ponderilor și de repetare până când rețeaua produce răspunsuri care corespund rezultatelor așteptate. Acest proces, cunoscut sub numele de algoritm de antrenare, presupune să decidă cum să prezinte datele, când să opreasă formarea și cum să se evaluateze succesul. În mod ideal, odată antrenată, rețeaua nu va memora doar exemplele, ci va fi capabilă să generalizeze, ceea ce înseamnă că poate clasifica sau răspunde corect la intrări pe care nu le-a mai văzut înainte. Fără această capacitate de generalizare, o rețea nu este mai utilă decât un tabel de căutare sofisticat. Si acesta este esența problemei: o rețea neuronală bună nu doar învăță, ci înțelege modelele suficient de bine pentru a face predicții exacte în situații noi. Această capacitate este ceea ce le face instrumente puternice în atât de multe domenii.[2]

Gândiți-vă la o rețea neuronală ca la o tablă albă la început. Are toate componente sale structurale noduri, conexiuni și ponderi dar nu știe încă nimic. Sarcina sa este să învețe cum să ia decizii sau să facă predicții pe baza exemplelor pe care i le dăm. Aici intervine învățarea automată. În învățarea mecanică, obiectivul este de a învăța un sistem să îndeplinească o sarcină, cum ar fi clasificarea imaginilor sau traducerea limbilor străine, prezentându-i date, mai degrabă decât programându-l cu instrucțiuni pas cu pas. Este exact ceea ce facem cu rețelele neuronale: le antrenăm folosind exemple etichetate perechi de intrare și ieșire dorită astfel încât, în cele din urmă, să poată face singure predicții bune. Începem cu date de intrare de exemplu, o imagine a unei cifre scrise de mână și o ieșire corectă cunoscută, cum ar fi eticheta 3. Această pereche este un exemplu din setul de instruire. Imaginea este introdusă în rețea. Aceasta se deplasează prin straturile de noduri, fiecare efectuând un mic proces de calcul multiplicând intrările prin ponderi, adăugând biasuri și trecând rezultatele printr-o funcție de activare. În cele din urmă, aceasta produce o ieșire, care este presupunerea sa: poate 7. Acum comparăm presupunerea rețelei cu răspunsul real. Această diferență este eroarea și este un semnal crucial. Rețeaua nu ridică pur și simplu din umeri și merge mai departe, ci folosește această eroare pentru a se ajusta. Aici intervine partea de învățare. Reteaua pornește de la eroare, calculând cât de mult a contribuit fiecare conexiune sau pondere la eroare. Aceasta utilizează un algoritm numit backpropagation pentru a atribui vina și a face corecții. Fiecare pondere este împinsă puțin în direcția care ar reduce eroarea data viitoare. Acest proces este repetat pe mii sau chiar milioane de exemple. Cu fiecare trecere, rețeaua își ajustează ponderile, învățând din date, până când rezultatele sale încep să corespundă rezultatelor așteptate de cele mai multe ori.

2.3 Python

³ Limbajul de bază folosit în acest proiect este Python, ales datorită ecosistemului vast de biblioteci dedicate învățării automate, în special celor de rețele neurale. Comunitatea activă și resursele bogate din jurul Python facilitează dezvoltarea și antrenarea modelelor ML, accelerând semnificativ procesul de implementare și testare a algoritmilor utilizați în lucrare. [3]

În plus, Python suportă atât paradigma orientată pe obiecte, cât și cea funcțională, oferind astfel flexibilitate maximă în structurarea codului și în alegerea stilului de programare potrivit pentru fiecare modul al sistemului de inteligență artificială.

Gestionarea pachetelor și a dependențelor se realizează foarte simplu prin instrumente precum pip sau Conda, care automatizează instalarea și actualizarea bibliotecilor necesare. Biblioteca standard a lui Python este generos dotată cu funcții utile, reducând dependențele externe. Fiind un limbaj versatil, Python stă la baza nu doar a soluțiilor AI, ci și a dezvoltării de aplicații web, analiză de date ori prototipuri de jocuri. Astfel, competența în Python deschide posibilitatea integrării fără efort a algoritmilor de inteligență artificială în proiecte web moderne.

2.3.1 Programarea orientată pe obiecte

⁴ Programarea orientată pe obiecte (POO) este o abordare de dezvoltare software în care componentele programului sunt definite ca obiecte și interacționează între ele. Programarea orientată pe obiect devine mai convenabilă, mai ușor de înțeles și mai extensibilă. Cele patru principii ale programării orientate pe obiect sunt: încapsularea, moștenirea, polimorfismul și abstractizarea. În redactarea acestei lucrări, am folosit două principii și anume încapsularea și moștenirea.[4]

Încapsularea se referă la gruparea datelor și a metodelor care operează asupra acestor date într-o singură unitate care se numește obiect. Permite ca datele să fie ascunse și să fie accesate numai prin intermediul metodelor definite, cunoscute sub numele de getters și setters.

Moștenirea este un mecanism care permite unei clase să moștenească proprietăți și comportamente de la o altă clasă. Clasa care moștenește se numește subclasă, în timp ce clasa de la care se moștenește se numește superclasă. Permite reutilizarea codului și crearea relațiilor ierarhice între clase.

³ Limbajul Python, <https://docs.python.org/3/tutorial/index.html?>

⁴ POO și principii, <https://zenodo.org/records/13845918>

2.3.2 PyTorch

⁵ PyTorch este un framework open-source de învățare automată dezvoltat inițial de Meta AI și întreținut în prezent de Fundația PyTorch. Acesta a fost conceput pentru a oferi o interfață imperativă care combină flexibilitatea Python cu performanța accelerată de hardware. Modelul de programare permite construirea și modificarea graficului de calcul în timpul execuției, facilitând depanarea, crearea rapidă de prototipuri și experimentarea iterativă cu arhitecturi neuronale. În centrul PyTorch se află obiectul Tensor, dar cu suport nativ pentru operațiunile CUDA, care asigură o execuție a unității de procesare grafică optimizată. Mecanismul autograd urmărește automat lanțul de operații efectuate asupra tensorilor și calculează gradienții necesari pentru formare prin metoda backpropagation cu un simplu apel.[5]

Începând cu versiunea 2.0, PyTorch oferă o soluție de compilare pentru optimizare în timp, plus suport pentru formare distribuită, precizie mixtă (AMP) și funcții de profilare integrate. Ecosistemul său include, de asemenea, biblioteci specializate (TorchVision, TorchText, TorchAudio) și instrumente pentru producție (TorchScript), făcându-l un instrument complet pentru cercetarea și ingineria deep learning.

Biblioteca PyTorch este utilizată pe scară largă în comunitatea de cercetare și industrie pentru dezvoltarea aplicațiilor de învățare automată, cum ar fi clasificarea imaginilor, recunoașterea vocală sau traducerea automată. Aduce la îndemână o abordare flexibilă și intuitivă pentru construirea și modificarea rețelelor neuronale, facilitând experimentarea și inovarea în domeniul învățării automate, beneficiază de o comunitate activă și de resurse ample de învățare, inclusiv documentație bogată, tutoriale și exemple de cod, care îi permit dezvoltatorului să exploreze cu ușurință.

2.3.3 Biblioteca Gym

⁶ Biblioteca Gym, dezvoltată de OpenAI, este un set de instrumente adoptat pe scară largă pentru cercetarea și experimentarea învățării prin consolidare. La modul cel mai simplu, Gym oferă o interfață standardizată pentru o colecție diversă de medii simulate în care un agent inteligent poate acționa, observă și învăță. Indiferent dacă vrei să înveți un robot virtual să echilibreze un stâlp, să antrenezi un agent să joace jocuri Atari clasice sau să creezi prototipul propriei tale probleme de control personalizate, Gym oferă un API consecvent, astfel încât să te poți concentra pe proiectarea și compararea algoritmilor de învățare, mai degrabă decât să te luptă cu detaliile specifice mediului.[6]

Fiecare mediu Gym implementează două metode principale: reset() și step(action). Apelul la reset() initializează mediul și returnează o observație inițială (de exemplu, o matrice de pixeli de pe ecranul unui joc sau un vector de stare numeric de la o simulare fizică). Apoi, de fiecare dată când agentul alege o acțiune de exemplu, să se deplaseze spre stânga, să accel-

⁵ Biblioteca Pytorch, <https://docs.pytorch.org/docs/stable/index.html>

⁶ Biblioteca Gym, <https://github.com/openai/gym/blob/master/README.md>

ereze înainte sau să aplique un cuplu apelați step(action). Această metodă returnează un tuplu de patru elemente: următoarea observație, o recompensă scalară, un indicator boolean done care indică dacă episodul s-a încheiat și un dicționar optional cu informații suplimentare de diagnosticare. Această buclă de feedback simplă și coerentă acțiune, observație, recompensă, finalizat acțiune se află în centrul designului Gym, ceea ce facilitează conectarea oricărui algoritm de învățare prin întărire care face bucle peste episoade și etape de timp. De-a lungul timpului, nenumărate lucrări de cercetare, tutoriale și proiecte open-source au adoptat Gym ca interfață standard pentru mediul lor. Ca urmare, orice algoritm nou pe care îl găsiți în literatura de specialitate a fost aproape sigur evaluat pe un mediu Gym. Desigur, Gym nu este lipsit de limitările sale, unele medii pot fi deterministe sau simpliste, ceea ce duce la adaptarea excesivă a algoritmilor la sarcini limitate. Altele pot necesita resurse de calcul semnificative, simulările de robotică, de exemplu, pot fi redate lent sau pot trece prin calcule fizice.

2.3.4 Biblioteca NumPy

⁷ NumPy este biblioteca fundamentală pentru calculul științific în Python, oferind un mediu vectorial multidimensional eficient și o multitudine de rutine pentru operații numerice rapide. Spre deosebire de listele native Python, vectorii NumPy impun un singur tip de date pentru toate elementele, permitând o dispunere compactă și continuă a memoriei și o interfață directă cu limbaje de nivel scăzut precum C și Fortran. Această alegere de proiectare stă la baza performanței sale superioare, în special atunci când se efectuează calcule vectoriale la scară largă.[7]

Un punct forte al bibliotecii NumPy constă în colecția sa cuprinzătoare de funcții matematice și statistice, de la aritmetică elementară și algebra liniară la transformările Fourier și generarea numerelor aleatorii. Operații precum difuzarea permit vectorilor de diferite forme să interacționeze fără probleme, reducând nevoia de bucle explicate și producând un cod concis și ușor de citit.

Biblioteca servește drept bază pentru o mare parte a ecosistemului PyData. Biblioteci precum pandas, SciPy, scikit-learn și chiar framework-uri de învățare profundă precum TensorFlow se bazează pe interfața matricei NumPy pentru schimbul de date și calcul. Prin intermediul interfeței sale C-API, matricele NumPy pot fi partajate dincolo de granițele limbajului fără a fi copiate, ceea ce le face ideale pentru extensiile de înaltă performanță și interoperabilitate cu unitățile de procesare grafică sau biblioteci de matrice dispersată.

În acest proiect, NumPy susține fiecare etapă a manipulării datelor, de la încărcarea seriilor de preturi sintetice în matrici la calculul vectorial al randamentelor și al parametrilor de portofoliu, permitând atât claritate, cât și viteză în fluxurile noastre de lucru experimentale.

⁷ Biblioteca NumPy, <https://numpy.org/devdocs//user/whatisnumpy.html?>

2.3.5 Biblioteca Pandas

⁸ Pandas este o bibliotecă Python de nivel înalt concepută pentru manipularea eficientă a datelor relaționale sau etichetate, bazându-se pe structurile de matrice NumPy pentru a furniza două obiecte principale, Series (matrice unidimensionale, etichetate cu indici) și DataFrame (date tabulare bidimensionale cu rânduri și coloane etichetate). Aceste结构uri de date suportă un set bogat de operații, de la aliniere și agregare la funcționalități de îmbinare, remodelare și serii temporale, permitând transformări concise și expresive ale datelor fără bucle Python explicite.[8]

O componentă de bază a setului său de instrumente I/O pandas.read_csv(), care citește fișiere separate prin virgulă în DataFrames printr-un singur apel, cu opțiuni pentru analizarea datelor, selectarea subseturilor de coloane, gestionarea valorilor lipsă și transmiterea fișierelor mari în bucăți. Pandas se integrează perfect cu NumPy și bibliotecile C optimizate pentru calcul numeric rapid, în timp ce metodele sale DataFrame returnează noi vizualizări sau copii după cum este necesar pentru siguranța memoriei.

În proiectul nostru, folosim Pandas pentru a încărca serii de prețuri sintetice, pentru a calcula statistici de rulare, pentru a uni mai multe CSV-uri în seturi de date unificate și pentru a pregăti intrări în loturi pentru mediile noastre Gym, simplificând în mod dramatic preprocesarea datelor și analiza exploratorie.

2.3.6 Biblioteca Matplotlib

⁹ Matplotlib este biblioteca standard de facto pentru trasarea 2D în Python, oferind atât o interfață procedurală pyplot (modelată după MATLAB), cât și un API complet orientat pe obiecte pentru controlul detaliat al figurilor și axelor. Aceasta suportă crearea de imagini statice, animații și vizualizări interactive care pot face zoom, panoramare și actualizare în timp real, ceea ce îl face potrivit pentru analiza exploratorie a datelor, precum și pentru crearea de grafice pentru publicare.[9]

La bază, Matplotlib se bazează pe matrici NumPy, traducând comenzi de trasare de nivel înalt în apeluri de desen optimizate prin intermediul unor backend-uri precum Agg, Cairo sau PostScript. Se poate personaliza practic fiecare aspect al unei linii de trasare, stiluri, marcatori, culori, fonturi, plasarea bifelor și multe altele, fie prin comenzi pyplot cu stare, fie prin manipularea directă a obiectelor.[9]

Rezultatele pot fi exportate în toate formatele comune (PNG, PDF, SVG, EPS) și încorporate într-o varietate de contexte: Jupyter notebooks, aplicații GUI (prin Tk, Qt, GTK), cadre web sau documente LaTeX. Integrarea strânsă a Matplotlib cu ecosistemul SciPy mai larg îl face

⁸ Biblioteca Pandas, https://pandas.pydata.org/docs/getting_started/overview.html

⁹ Biblioteca Matplotlib, <https://matplotlib.org/stable/users/index.html>

alegerea ideală pentru vizualizarea seriilor de timp, histogramelor, diagramele de dispersie, hărților termice sau a tipurilor de diagrame personalizate - esențiale atât pentru depanarea comportamentului algoritmilor, cât și pentru prezentarea rezultatelor în teză.

2.4 CSV (Comma separated values)

Fișierele CSV prescurtarea de la Comma-Separated Values (valori separate prin virgulă) reprezintă o modalitate simplă de stocare și schimb de date tabulare. În esență, un fișier CSV este doar text simplu, fiecare linie reprezintă un rând de date, iar în cadrul fiecărei linii, câmpurile individuale (sau coloanele) sunt separate de un delimitator ales, cel mai adesea o virgulă. Acest format ușor face ca fișierele CSV să fie ușor de citit, editat și procesat, indiferent dacă vă uitați rapid la conținut într-un editor de text sau utilizați un script pentru a încărca mii de rânduri în program.

2.5 Modele stocastice de generare a prețurilor

Pentru a antrena și testa agenții fără a ne baza pe date reale de pe piață, am construit de la zero propriul univers format din 27 acțiuni. În acest fel, știm exact ce determină mișcările prețurilor care evită dependențele de feed-uri din piețe reale și incertitudinile asociate. Am folosit trei modele simple pentru a acoperi principalele comportamente observate pe piețe:

1. Mișcarea geometrică browniana(GBM) - face ca prețurile să crească sau să scadă în timp, cu creșteri și scăderi aleatorii care cresc în dimensiune pe măsură ce prețul devine mai mare.
2. Procesul Ornstein–Uhlenbeck aplicat pe logaritmul prețului – pentru a captura efectul de revenire la o valoare de echilibru, obișnuit în piețe atunci când prețurile deviază prea mult de la valoarea fundamentală.
3. Harta logistică haotică – pentru a genera salturi imprevizibile și evenimente rare cu impact puternic, simulând crize sau știri neașteptate care pot declanșa reacții brusăre pe piață.

Mai întâi voi prezenta, la nivel teoretic, ecuațiile diferențiale și recurențele folosite, iar apoi voi detalia implementarea numerică, variantele Euler–Maruyama pentru GBM și OU, respectiv schema discretă pentru harta logistică și modul în care am calibrat parametrii pentru a obține o serie cu proprietăți statistice apropriate de cele observate empiric.

2.5.1 Mișcarea geometrică browniana(GBM)

¹⁰ La baza pieței se află modelul matematic GBM pentru generarea datelor. Din numărul total de active, 20 fișiere CSV din depozitul nostru este de fapt rezultatul discretizării unei ecuații diferențiale stochastice în timp continuu. Mai exact, modelăm fiecare activ S_i prin intermediul unui proces de mișcare browniană geometrică[10]

$$dS_{i,t} = \mu_i S_{i,t} dt + \sigma_i S_{i,t} dW_{i,t},$$

unde μ_i este drift-ul acelui activ, σ_i volatilitatea sa, iar $dW_{i,t}$ este un increment Wiener standard. În codul nostru, aproximăm

$$S_{i,t+1} = S_{i,t} \exp((\mu_i - \frac{1}{2}\sigma_i^2)\Delta t + \sigma_i \sqrt{\Delta t} \epsilon_{i,t}),$$

cu Δt zi și $\epsilon_{i,t} \sim \mathcal{N}(0, 1)$. Prin rularea acestei recursiuni timp de 500 de pași, producem 500 de prețuri zilnice și le scriem ca stock.i.csv. Pentru a produce salturi ocazionale (de exemplu, șocuri bruse de știri), uneori multiplicăm actualizarea cu un factor de salt aleatoriu $J_{i,t}$. Concret:

$$S_{i,t+1} = S_{i,t} \exp((\mu_i - \frac{1}{2}\sigma_i^2)\Delta t + \sigma_i \sqrt{\Delta t} \epsilon_{i,t}) \times J_{i,t}, \text{ unde}$$

$$J_{i,t} = \begin{cases} 1, & \text{cu probabilitatea } 1 - \lambda, \\ \exp(K + \xi_{i,t}), & \text{cu probabilitatea } \lambda. \end{cases}$$

unde λ este probabilitatea de salt pe zi, K este dimensiunea fixă a saltului (în log-spațiu), iar $\xi_{i,t}$ adaugă zgromot suplimentar în jurul saltului respectiv. Această extensie simplă de difuzie a salturilor creează mișcări rare, dar mari, făcând universul nostru sintetic puțin mai apropiat de piețele reale.

2.5.2 Procesul Ornstein–Uhlenbeck

¹¹ Pentru a surprinde tendința prețurilor de a reveni la un anumit nivel pe termen lung, modelăm prețul logaritmnic $X_t = \ln S_t$ cu un proces Ornstein-Uhlenbeck (OU). În timp continuu, dinamica OU satisface

$$\frac{dX_t}{dt} = k(\theta - X_t) + \sigma \frac{dW_t}{dt},$$

unde $k > 0$ este viteza de reversie medie, θ este media logaritmică pe termen lung spre care este tras X_t , σ este volatilitatea, iar W_t este procesul Wiener standard.[11] Discretizăm cu

¹⁰ Modelul matematic GBM, https://www.researchgate.net/publication/202924343_Stochastic_Differential_Equations_An_Introduction_with_Applications

¹¹ Procesul OU, <https://stevencjxie8.com/files/refs/ref7.pdf>

$\Delta t = 1, \varepsilon_t \sim \mathcal{N}(0, 1)$ folosind schema Euler-Maruyama:

$$\begin{aligned} X_{t+1} &= X_t + k(\theta - X_t)\Delta t + \sigma\sqrt{\Delta t}\varepsilon_t, \\ S_{t+1} &= \exp(X_{t+1}) \end{aligned}$$

Schema Euler-Maruyama este cel mai simplu mod de a transforma o ecuație diferențială stocastică în timp continuu într-o actualizare în timp discret pe care o putem simula pe un computer.

2.5.3 Harta logistică haotică

¹² Pentru a simula oscilații rare, dar mari și imprevizibile (evenimente de tip lebădă neagră) în universul nostru sintetic, folosim harta logistică clasică, un sistem dinamic simplu în timp discret care prezintă haos pentru anumite valori ale parametrilor. Trebuie să generăm o secvență de tipul $x_t \in (0, 1)$, iterând

$$x_{t+1} = rx_t(1 - x_t), \quad 3,57 \leq r \leq 4.0,$$

unde r este parametrul de creștere. Atunci când r depășește aproximativ 3,57, harta suferă o succesiune de dublări ale perioadei până la haosul total, făcând ca x_t să fie extrem de sensibil la condiția sa initială x_0 . În ciuda determinismului său, pentru aceste valori ale r , secvența se comportă ca un semnal zgomotos, imprevizibil.[12]

Transformăm apoi x_t în şouri de preț în jurul unui preț de bază S_{base} prin

$$S_t = S_{base}[1 + c(x_t - 0.5)], \quad c \in (0, 1),$$

unde c scalează amplitudinea fiecărui soc, astfel încât, atunci când x_t este aproape de 1, se obține o mișcare ascendentă de până la $+c \times \frac{S_{base}}{2}$, iar când x_t este aproape de 0, o mișcare descendente de magnitudine similară. Alegând un c mic (de exemplu, 0,05), ne asigurăm că aceste oscilații haotice sunt vizibile, dar nu atât de mari încât să eclipseze comportamentele GBM și OU.

Spre deosebire de zgomotul Gaussian pur, rezultatul hărții logistice este reproductibil cu același seed, dar efectiv imprevizibil fără aceasta. Pentru r în regimul haotic, secvența vizitează întregul interval $(0, 1)$ într-un model de tip fractal, injectând varietate în piață. O singură înmulțire și scădere per pas face ușoară simularea chiar și a sute de active.

2.6 Modelul REINFORCE

¹³ Cadrul general al învățării prin consolidare cuprinde o multitudine de probleme, de la controlul învățării la diferite forme de optimizare a funcțiilor. Cu toate acestea, cercetarea

¹² Model matematic pentru harta logistică haotică <https://doi.org/10.1038/261459a0>

¹³ Modelul reinforce, <https://doi.org/10.1007/BF00992696>

În aceste domenii particulare se concentrează adesea pe o serie de probleme distincte, este probabil ca metodele de învățare prin consolidare eficiente pentru agenții autonomi care lucrează în circumstanțe reale vor trebui să abordeze toate problemele în ansamblu. În consecință, deși concentrarea asupra unor tipuri limitate de probleme de învățare prin consolidare doar pentru a menține problemele ușor de rezolvat continuă să fie o strategie de cercetare utilă, este important să se țină seama de faptul că rezolvarea problemelor cele mai dificile va necesita probabil integrarea unei game largi de tehnici aplicabile.

Algoritmul REINFORCE este unul dintre cei mai simpli algoritmi de tip policy-gradient în învățarea prin consolidare. Acesta tratează politica ca pe o cutie neagră, o funcție parametrizată de obicei, o rețea neuronală care generează direct o distribuție de probabilitate asupra acțiunilor, având în vedere observația curentă.[13]

În arhitectura rețelei avem nevoie de un vector de observații aplatizat (prețuri, dețineri, numerar), de dimensiune D , două straturi ascunse conectate, fiecare urmat de o activare $ReLU$, un strat liniar final care produce $3 \times N$ logiți, unde N este numărul de acțiuni. Acești logiți parametreză N distribuții categorice independente (păstrare, cumpărare, vânzare) și un strat liniar separat care produce o singură estimare scalară a valorii $V(s)$. La intrare avem $s \in \mathbf{R}^D$, $h = ReLU(W_2ReLU(W_1s + b_1) + b_2)$, $\text{logiți} = W_{\text{logits}}h + b_{\text{logits}}$, $V(s) = w_v^T h + b_v$. La fiecare etapă, agentul calculează logiți și $V(s)$, remodelează logiți într-un tensor $N \times 3$, aplică funcția softmax pe rânduri pentru a obține probabilitățile de acțiune pentru fiecare activ, eșantionează o acțiune per stoc din fiecare distribuție categorială după care se înregistrează suma tuturor log-probabilităților $\log\pi(a_t | s_t)$ și valoarea stării $V(s_t)$ și se obține astfel un vector de acțiuni întregi, un scalar logaritmic bazat pe probabilități și o valoare scalară.

2.7 Optimizarea politcii proximale (PPO)

¹⁴ Optimizarea politcii proximale se bazează pe ideea de bază a gradientilor de politică pre-cum REINFORCE, dar adaugă câteva trucuri practice pentru a face învățarea mai stabilă. În loc să ia o singură traекторie Monte Carlo și să facă o singură actualizare mare, algoritmul colectează un lot de experiență, apoi actualizează politica de mai multe ori pe baza acelorași date, dar reduce raportul politicii, astfel încât fiecare actualizare să nu poată deplasa noua politică prea departe de cea veche. Prin urmare, fiecare etapă respectă politica anterioară pentru a evita schimbările semnificative dăunătoare. folosește o singură rețea neuronală cu două capete: actorul (emite probabilitățile de acțiune) și criticul (emite o estimare a valorii pentru starea curentă). Prin a-i spune actorului cât de bine sau de rău a ieșit fiecare acțiune în comparație cu așteptările, criticul servește drept bază de referință învățată și ajută la reducerea variației. Algoritmul utilizează un obiectiv surogat care combină ratele de politică cu tăiere și fără tăiere pentru a maximiza logaritmul bazat pe probabilitate ponderată în funcție de randamente, mai degrabă decât să maximizeze direct logaritmul bazat pe probabilitate ponderată în funcție de randamente.

¹⁴ Introducere în ppo, <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf>

Acest truc permite algoritmului să împingă actualizarea politiciei cât de departe se poate fără a depăși limitele. Pentru a evita prăbușirea prematură a politiciei, adaugă un mic bonus pentru o entropie mai mare în distribuția acțiunilor, încurajând explorarea continuă. După colectarea unui lot de traекторii, îl amestecă în mini-loturi și rulează mai multe epoci de coborâre a gradientului, obținând mai multe impulsuri de învățare din fiecare interacțiune cu mediul.[14] Raportul de probabilitate este definit prin intervalul de timp t la fiecare pas, care definește raportul dintre politica nouă π_θ și politica veche $\pi_{\theta_{old}}$:

$$r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}$$

Algoritmul maximizează o versiune trunchiată a obiectivului standard al gradientului de politică pentru a preveni actualizări majore:

$$L^{CLIP}(\theta) = E_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)],$$

unde \hat{A}_t este o estimare a avantajului, iar ϵ este hiperparametrul de trunchiare. Pentru estimarea avantajului generalizat calculăm \hat{A}_t prin GAE pentru a compensa biasul față de varianță:

$$\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t), \hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l},$$

unde $\lambda \in [0, 1]$ aplatypează estimarea. Pentru a nu pierde funcția de valoare antrenăm simultan un critic $V_\theta(s)$ prin minimizarea erorii medii pătrate a randamentelor R_t :

$$L^{VF}(\theta) = E_t[(V_\theta(s_t) - R_t)^2].$$

Pentru a încuraja explorarea și a preveni convergența prematură, algoritmul adaugă un regularizator de entropie:

$$L^{ENT}(\theta) = E_t[H(\pi_\theta(\cdot | s_t))]$$

ponderat de coeficientul β .

Combinând toate ecuațiile va rezulta pierderea totală și anume:

$$L(\theta) = -L^{CLIP}(\theta) + c_1 L^{VF}(\theta) - c_2 L^{ENT}(\theta).$$

[14]

În timpul antrenamentului, colectăm un lot de tranzitii, calculăm aceste pierderi pe mai multe epoci și mini-loturi și efectuăm pași de gradient pe θ . Obiectivul limitat al algoritmului asigură că fiecare actualizare rămâne proximală politiciei anterioare, rezultând o metodă de gradient de politică stabilă, dar eficientă.

Capitolul 3

Implementarea aplicației

3.1 Logica de funcționare

În figura de mai jos este ilustrat fluxul de lucru general al framework-ului nostru de agenți care tranzacționează. În primul rând, generatorul de date produce serii de prețuri sintetice utilizând mișcarea browniană geometrică, revenirea la medie Ornstein-Uhlenbeck și o hartă logistică haotică, scriind prețurile fiecărui instrument în fișiere CSV. Piața încarcă apoi aceste fișiere CSV și expune o stare standard de resetare a portofoliului în stil Gym și parcurge datele de preț pentru a aplica acțiuni și a calcula recompense. Agenții (aleatorii, bazați pe reguli, bazați pe NN sau bazați pe PPO) interacționează cu acest mediu într-o buclă, selectând acțiuni pe baza observațiilor și colectând experiență. Orchestratorul experimentului automatizează antrenamentul și evaluarea tuturor agenților și seed-urilor aleatoare, înregistrând în același timp metricile de performanță. În cele din urmă, modulul de rezultate generează grafice ale curbei de capital, diagrame cu bare și tabele statistice, permitându-ne să comparăm comportamentele agenților și să evaluăm robustețea în condiții controlate și repetabile.



Figura 3.1: Logica de funcționare a aplicației

3.2 Reproductibilitate și configurare

Pentru a obține experimente deterministe și ușor de comparat, fixăm semințele tuturor generatoarelor de numere aleatorii imediat după pornirea scriptului principal:

```
import random
import numpy as np
import torch

SEED = 42
random.seed(SEED)                      # Python RNG
np.random.seed(SEED)                    # NumPy RNG
torch.manual_seed(SEED)                 # PyTorch CPU RNG
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)    # PyTorch CUDA RNG
```

În continuare, imediat după instanțierea mediului, legăm generatorul specific spațiului de acțiuni:

```
env = MarketEnvFrac(                  # sau MarketEnvDiscrete(...)
    data_generator=csv_gen(...),
    num_stocks=N,
    initial_cash=1000,
    transaction_cost=0.005,
    min_hold_days=3,
    max_trade_pct=0.05
)
env.action\_\_space.seed(SEED)        # reproductibilitate și pentru
                                    # esantionarile din Gym
```

De asemenea, directorul `data/` conține fișierele `stock_1.csv`, ..., `stock_N.csv`, fiecare reprezentând o serie de preț generată de modelele GBM, OU sau logistic map (2.5).

Parametrii globali ai mediului sunt:

- ▷ `initial_cash=1000`: soldul inițial de numerar;
- ▷ `transaction_cost=0.005`: comision de 0.5% per tranzacție;
- ▷ `min_hold_days=3`: perioada minimă de păstrare înainte de vânzare;
- ▷ `max_trade_pct=0.05` (în `MarketEnvFrac`): procentul maxim din portofoliu tranzacționat per pas.

Prin acest set de configurații, ne asigurăm că toate componente aleatorii (Python, NumPy, PyTorch, Gym) utilizează aceeași sămânță, încărcarea și ordinea fișierelor CSV din `data/` rămân constante și parametrii esențiali ai mediului sunt clar documentați și modificați central.

3.3 Manipularea datelor

Mediul de tranzacționare este construit pe baza librăriei Gym, care formalizează lumea în care agenții observă prețurile, întreprind acțiuni și primesc recompense. Conceptual, la fiecare pas de timp discret t , mediul menține trei cantități de bază:

1. Un vector al prețurilor curente ale activelor, $\{S_{i,t}\}_{i=1}^N$, unde N este numărul total de active sintetice.
2. Un vector al numărului de acțiuni din fiecare activ deținute în prezent de agent, $\{h_{i,t}\}_{i=1}^N$
3. Un sold de numerar scalar, C_t .

3.3.1 Clasa de bază a pieței

La fiecare pas de timp t , mediul reunește cele trei componente (prețuri, dețineri, numerar) într-o singură observație Gym. Agentul primește

$$o_t = [S_{1,t}, S_{2,t}, \dots, S_{N,t}, h_{1,t}, h_{2,t}, \dots, h_{N,t}, C_t] \in \mathbf{R}^{2N+1},$$

unde $S_{i,t}$ este prețul activului i la pasul t (generate conform modelului GBM/OU/logistic, 2.5), $h_{i,t}$ este numărul de acțiuni din activul i deținute de agent, iar C_t este soldul de numerar disponibil, sub condiția ca prețurile și numerarul să fie nenegative, iar deținerile să fie numere întregi (zero sau pozitive).

Pentru fiecare activ i , agentul trebuie să aleagă o acțiune discretă. Acțiunea $a_{i,t}$ a fiecărui activ poate lua una dintre următoarele trei valori, 0 nu ia nicio poziție nouă în activul i , 1 pentru a cumpăra exact o acțiune din activ, sau 2 pentru a vinde. Mediul impune două constrângeri principale asupra fiecărei încercări de tranzacționare:

1. Constrângerea numerarului pentru cumpărături: Pentru a cumpăra o acțiune din activul i la momentul t , agentul trebuie să dispună de suficient numerar:

$$C_t \geq S_{i,t}(1 + fee),$$

unde $fee = 0.005$

2. Restricție de menținere și menținere minimă: Dacă agentul a cumpărat o acțiune din activul i în ziua t_0 , atunci trebuie să dețină acea acțiune timp de cel puțin τ_{min} zile înainte de a i se permite să o vândă. Păstrăm un vector de contor $\{m_{i,t}\}$ pentru fiecare activ: ori de câte ori activul i este cumpărat la momentul t , stabilim $m_{i,t+\Delta} = \tau_{min}$; $\Delta = 0$, iar în fiecare zi următoare decrementăm $m_{i,t} \rightarrow m_{i,t} - 1$. Agentului i se permite să vândă numai dacă $m_{i,t} = 0$ (adică dacă acțiunea a fost deținută suficient de mult timp).

La inițializare mediul primește o funcție `data_generator` care returnează o listă de `DataFrames` cu seriile de preț, numărul de active sintetice `num_stocks`, soldul initial `initial_cash`, costul

de tranzacție `transaction_cost` și perioada minimă de deținere `min_hold_days`. În constructor se apelează apoi funcția `_regenerate_data()`, care încarcă datele și setează `self.max_steps`. La `reset()`, se reinițializează `current_step=0`, `cash=initial_cash` și `shares_held=[0,...,0]`, apoi se returnează starea inițială prin `_get_observation()`, care construiește vectorul

$$o_t = \left[\underbrace{S_{1,t}, S_{2,t}, \dots, S_{N,t}}_{\text{prețuri}}, \underbrace{h_{1,t}, h_{2,t}, \dots, h_{N,t}}_{\text{dețineri}}, C_t \right] \in \mathbb{R}^{2N+1}$$

Pentru că nu putem avea doar o piață cu mediu discret, următoarea funcție generează numărul de active fractionar și parcurge logica tranzacțiilor fractionare: pentru fiecare acțiune continuă în $[-1, 1]$ (varianta fractională inițială), transformăm fractiunea în buget sau în număr de acțiuni de vândut, aplicăm taxele și actualizăm `self.cash` și `self.shares_held`. După asta avansăm pasul, calculăm noua valoare a portofoliului și recompensele.

3.3.2 Piață discretă

Clasa `MarketEnvDiscrete` extinde clasa de bază `MarketEnv` pentru a lucra cu un spațiu de acțiuni discrete, fie `0 = retine`, `1 = cumpără una`, `2 = vinde una per activ`. Pentru a folosi această clasă avem nevoie de un constructor (`__init__`), se suprascrie `self.action_space` cu un `spaces.MultiDiscrete([3] * N)`, unde fiecare din cele N pozitii poate lua valorile $\{0, 1, 2\}$. Pentru a împiedica tranzacțiile prea frecvente, păstrăm un vector `self.last_trade_step` de lungime N , inițializat cu `min_hold_days` care la fiecare pas, înainte de a executa o comandă pe activul i , verificăm:

```
if self.current_step - self.last_trade_step[i] <
    self.min_hold_days:
    continue
```

Dacă a trecut suficient timp (`min_hold_days`), putem cumpără/vinde iar apoi actualizăm, pașul de dinainte devine pasul curent. La fiecare pas al acțiunii

```
prices    = [df.iloc[self.current_step]['price'] for df in
            self.data]
old_val   = cash + sum(shares_held[i] * prices[i])
for i, act in enumerate(action):
    if act == 1 and cash >= prices[i]*(1+fee):
        # cumpără o acțiune
        cash      -= prices[i]*(1+fee)
        shares_held[i] += 1
        last_trade_step[i] = current_step
    elif act == 2 and shares_held[i] > 0:
        # vinde o acțiune
        shares_held[i] -= 1
        cash       += prices[i]*(1-fee)
        last_trade_step[i] = current_step
    # altfel act == 0 -> hold
```

Când se ajunge la final aplicăm o funcție pentru lichidare de stoc, pentru ca agenții să nu mai aibă active la curent. În ansamblu, clasa MarketEnvDiscrete combină mecanica standard Gym (reset/step) cu reguli specifice de tranzacționare discretă. Verificarea cash-ului și a cooldown-ului, calculul taxelor, actualizarea portofoliului și forțarea lichidării la final. Acest mediu oferă un cadru curat pentru a compara strategii bazate pe decizii întregi de cumpărare și vânzare.

3.3.3 Piață fractionară

Clasa MarketEnvFrac extinde clasa tată MarketEnv pentru a accepta acțiuni continue în intervalul $[-1, 1]$, unde fiecare valoare indică ce procent din suma de bani (la cumpărare) sau din detineri (la vânzare) să fie tranzacționat. În constructor, după apelul la super(), suprascriem:

```
self.max_trade_pct = max_trade_pct
self.action_space = spaces.Box(-1.0, 1.0, shape=(N,), 
                               dtype=np.float32)
```

Astfel, un vector *action* cu valori între $[-1, 1]$ indică, pentru fiecare activ i , procentul din suma de bani (dacă $a_i > 0$) sau procentul din detineri (dacă $a_i < 0$) pe care vrem să le cumpărăm sau vindem. Transformarea fracțiunilor în tranzacții se face în funcția *step()* care ia ca parametru vectorul *action*:

```
pcts = np.clip(action, -1, 1) * self.max_trade_pct
prices = [df.iloc[self.current_step]['price'] for df in self.data]
old_value = self.cash + sum(h*p for h,p in zip(self.shares_held,
                                                 prices))
```

La fel ca și clasa anterioară, pentru fiecare activ i avem

```
frac = pcts[i]
price = prices[i]
if frac > 0:
    # cumparam fractiune de cash
    spend = self.cash * frac
    cost = spend * (1 + self.transaction_cost)
    if cost <= self.cash:
        shares_to_buy = spend / price
        self.shares_held[i] += shares_to_buy
        self.cash -= cost

elif frac < 0:
    # vindem fractiune din holdings
    to_sell = self.shares_held[i] * (-frac)
    proceeds = to_sell * price * (1 - self.transaction_cost)
    self.shares_held[i] -= to_sell
    self.cash += proceeds
# daca frac == 0 -> hold
```

După ce se face o tranzacție, trebuie să avansăm timpul cu o zi, se face prin adunarea cu 1 la variabila pas curent și se adaugă tranzacția în vectorul acțiuni, se actualizează vectorul cu valori din portofoliu, iar la finalul episodului se face lichidarea de stoc și returnează tuple-ul în mediul Gym.

Prin această implementare, clasa MarketEnvFrac permite tranzacții mai fine, în care agentul nu cumpără/vinde doar unități întregi, ci poate regla volumul în funcție de semnalul politicii și de dimensiunea portofoliului, oferind un spațiu de acțiuni continuu adecvat pentru agenți ca PPOAgent sau NNAgent.

3.4 Agenții virtuali

În cadrul nostru de tranzacționare virtuală, fiecare agent implementează o politică care transpune observațiile mediului într-un vector de acțiune discret. Toate cele trei tipuri de agenți aleatoriu, bazat pe reguli și rețea neuronală sunt conforme cu aceeași interfață, ele implementează o funcție care returnează un astfel de vector N -dimensional. În continuare, vom descrie fiecare agent în parte, subliniind logica sa decizională și, dacă este cazul, orice formule matematice aflate la baza acestieia.

3.4.1 Agentul aleator

Agentul aleatoriu ne servește drept cea mai naivă referință. Aceasta nu inspectează deloc observația, în schimb, se bazează pur și simplu pe spațiul de acțiune al mediului, care este multi discret de dimensiune N . În alte cuvinte, la momentul t , regula de decizie a RandomAgent este:

$$a_t = [a_{1,t}, a_{2,t}, \dots, a_{N,t}], \quad a_{i,t} \sim \text{Uniform}\{0, 1, 2\},$$

independent de fiecare i și fixăm, de asemenea, random.seed() din Python și RNG din NumPy, secvența de extrageri $\{a_{i,t}\}$ este deterministă pentru orice sămânță dată. Dar, în afară de reproductibilitate, nu există inteligență aici, acest agent cumpără, vinde sau deține pur și simplu fiecare dintre stocurile N cu probabilitate egală la fiecare pas, supus doar constrângерilor de numerar și de timp de păstrare ale mediului. Ca urmare, curba tipică a acțiunilor sale prezintă fluctuații aleatorii, de obicei descendente pe un orizont lung, deoarece nu are niciun semnal predictiv sau euristic.

3.4.2 Agentul bazat pe o regulă

Agentul bazat pe o regulă implementează o euristică simplă care compară prețul de astăzi al fiecărui activ, $S_{i,t}$, cu prețul său de ieri, $S_{i,t-1}$. Concret, la primul pas de timp $t = 0$, nu există un preț anterior, astfel încât agentul returnează doar $a_i = 0$ pentru toți i . Pentru fiecare

$t \geq 1$ ulterior, acesta aplică următoarea regulă pentru fiecare stoc i :

$$a_{i,t} = \begin{cases} 1, & \text{dacă } \frac{S_{i,t}}{S_{i,t-1}} \leq 0.98, \\ 2, & \text{dacă } \frac{S_{i,t}}{S_{i,t-1}} \geq 1.05, \\ 0, & \text{în celelalte cazuri} \end{cases}$$

Odată ce întregul vector este determinat, agentul îl supune metodei step(a_t) a mediului. Deoarece mediul aplică verificări ale soldului de numerar și o perioadă minimă de reținere înainte de a permite orice vânzare, cumpărăturile și vânzările intenționate ale agentului bazat pe reguli pot fi ocazional respinse (dacă nu are numerar pentru a cumpăra sau dacă termenul de reținere pentru acțiunea respectivă nu a expirat încă), caz în care mediul aplică o mică penalizare la valoarea veche a portofoliului. După fiecare acțiune, agentul își actualizează copia internă a prețurilor anterioare. Deoarece această logică deterministă a pragurilor depinde doar de o scădere constantă de 2% sau de o creștere constantă de 5%, ea cumpără scăderile și ia profituri în cazul creșterilor, sub rezerva comisioanelor de tranzacționare și a constrângerilor de păstrare. Deși ignoră în continuare modelele pe mai multe zile sau volatilitatea, acest RuleBasedAgent surclasă de obicei RandomAgent, deoarece capitalizează inversările sau continuările evidente de o zi.

3.4.3 Agentul bazat pe rețele neuronale

Agenții noștri bazati pe rețele neuronale parametrizează direct o politică $\pi_\theta(a | o)$ și, în cazul actor-critic, și o funcție de valoare $V_\theta(o)$, folosind o rețea feed-forward cu două straturi ascunse de dimensiune H . Intrarea (o_t) este vectorul de observație

$$o_t = [S_{1,t}, S_{2,t}, \dots, S_{N,t}, h_{1,t}, h_{2,t}, \dots, h_{N,t}, C_t],$$

iar ieșirile sunt:

1. Head-ul de politică: un vector de logită de dimensiune $3N$, reformabil într-o matrice $N \times 3$, urmat de softmax pentru a genera probabilitățile fiecărei acțiuni discrete pentru fiecare activ.
2. Head-ul de valoare: un scalar $V_\theta(o_t)$ care estimează valoarea stării, folosit ca baseline pentru reducerea varianței.

Actualizarea parametrilor θ se face prin metode de policy-gradient, REINFORCE[13] și optimizarea politicii proximale[14]. Astfel, arhitectura este comună, iar diferența între cei doi agenți constă în funcția de pierdere și mecanismul de update:

1. REINFORCE maximizează direct $E[\log_{\pi_\theta}(a_t | o_t)(G_t - V_\theta(o_t))]$
2. PPO optimizează obiectivul tăiat $\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 \pm \epsilon)\hat{A}_t)$ plus termeni de valoare și entropie,

după cum detaliem în Secțiunea 2.6(REINFORCE) și Secțiunea 2.7 (PPO).

3.4.3.1 Arhitectura rețelei

Agentul folosește un MLP cu două straturi ascunse de dimensiune H și două head-uri: - *policy head* care produce $3N$ logiți refăcuți într-o matrice $N \times 3$, pe care aplicăm softmax pentru a obține $\pi_{\theta}(a_{i,t} | o_t)$. - *value head* care produce un scalar $V_{\theta}(o_t)$, folosit ca bază [14, 13].

3.4.3.2 Eșantionarea acțiunilor

La inferență, transformăm o_t într-un tensor, rulăm:

```
logits, value = self.policy_net(obs_t)
probs = torch.softmax(logits.view(N, 3), dim=1)
actions = torch.distributions.Categorical(probs).sample()
logp =
    torch.distributions.Categorical(probs).log_prob(actions).sum()
```

3.4.3.3 Antrenarea agentului REINFORCE

Pentru baza teoretică a algoritmului REINFORCE, vezi Secțiunea 2.6 [13, 14]. În implementarea noastră, parcurgem următorii pași:

1. Colectarea traiectoriei.

Apelăm `env.reset()`, apoi într-o buclă `while not done`:

- ▷ `action, logp, value = agent.select_action(obs)`
- ▷ `obs, reward, done, _ = env.step(action)`
- ▷ stocăm în liste: $\{\log p_t\}$ și $\{r_t\}$.

2. Calculul randamentelor.

După episod, calculăm returnurile reduse

$$G_t = \sum_{k=t}^T \gamma^{k-t} r_k,$$

le normalizăm ($\mu = 0$, $\sigma = 1$) pentru stabilitate.

3. Construirea funcției de pierdere.

Pierdere actor:

$$L_{\text{actor}} = - \sum_{t=0}^T \log \pi_{\theta}(a_t | o_t) G_t,$$

Pierdere critică (baseline din reteaua noastră):

$$L_{\text{critic}} = \sum_{t=0}^T (G_t - V_\theta(o_t))^2.$$

Bonus de entropie:

$$L_{\text{entropy}} = -\alpha \sum_{t=0}^T H(\pi_\theta(\cdot | o_t)).$$

Pierdere totală:

$$L = L_{\text{actor}} + \frac{1}{2}L_{\text{critic}} + L_{\text{entropy}}.$$

4. Exemple de cod.

```
# calculam policy_loss pe fiecare pas:
policy_loss = []
for logp_t, R_t in zip(logps, returns):
    policy_loss.append(-logp_t.sum() * R_t)
policy_loss = torch.stack(policy_loss).sum()

# backprop si step
self.optimizer.zero_grad()
policy_loss.backward()
self.optimizer.step()
```

5. Reproducibilitate.

Am fixat seed-urile pentru Python, NumPy, PyTorch și `env.action_space.seed`, astfel încât totul este determinist și ușor de comparat între rulări.

La fiecare 100 de episoade, logăm progresul cu: `print(f"Episode {ep}/{num_episodes completed}")`.

3.4.3.4 Antrenarea agentului PPO

Pentru descrierea detaliată a algoritmului PPO (Proximal Policy Optimization) vedeti Secțiunea 2.7 [14]. În implementarea noastră ('PPOAgent.train'), antrenamentul se desfășoară astfel:

1. Colecțarea traiectoriei.

Apelăm `env.reset()`, apoi într-o buclă `while not done`:

- ▷ `action, logp, value = agent.select_action(obs)`
- ▷ `obs, reward, done, _ = env.step(action)`
- ▷ stocăm în liste: $\{o_t\}, \{a_t\}, \{\log p_t\}, \{r_t\}, \{V_t\}, \{d_t\}$

2. Calculul avantajelor și returnurilor.

La sfârșitul episodului evaluăm valoarea următoare V_{T+1} și calculăm GAE:

$$\delta_t = r_t + \gamma V_{t+1} (1 - d_t) - V_t, \quad \hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma \lambda)^l \delta_{t+l}.$$

Returnurile ţintă: $R_t = \hat{A}_t + V_t$.

3. Construirea funcției de pierdere.

Pe lotul complet aplicăm mai multe epoci de actualizare în mini-loturi:

$$r_t(\theta) = \frac{\pi_\theta(a_t | o_t)}{\pi_{\theta_{\text{old}}}(a_t | o_t)}.$$

Pierdere de politică tăiată:

$$L_{\text{clip}} = -\mathbb{E}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right].$$

Pierdere de valoare:

$$L_{\text{value}} = \frac{1}{2} \mathbb{E}_t \left[(R_t - V_\theta(o_t))^2 \right].$$

Bonusul de entropie:

$$L_{\text{ent}} = -\beta \mathbb{E}_t [H(\pi_\theta(\cdot | o_t))].$$

Pierdere totală:

$$L = L_{\text{clip}} + c_1 L_{\text{value}} + L_{\text{ent}}.$$

4. Exemplu de cod.

```

for _ in range(self.update_epochs):
    idx = torch.randperm(batch_size)
    for start in range(0, batch_size, self.minibatch_size):
        mb = idx[start:start+self.minibatch_size]
        logits, values = self.net(obs[mb])
        dist = Categorical(F.softmax(logits.view(-1,N,3), dim=2))

        new_logp = dist.log_prob(actions[mb]).sum(1)
        entropy = dist.entropy().sum(1).mean()

        ratio      = (new_logp - old_logp[mb]).exp()
        clipped    = torch.clamp(ratio, 1-epsilon, 1+epsilon) *
                     adv[mb]
        surrogate = torch.min(ratio * adv[mb], clipped)
        policy_loss = -surrogate.mean()
    
```

```
value_loss = F.mse_loss(values.squeeze(-1), returns[mb])
loss = policy_loss + 0.5*value_loss -
      self.entropy_coef*entropy

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

5. Reproducibilitate & logare.

Am fixat seed-urile pentru Python, NumPy, PyTorch și env.action_space.seed, astfel încât rulările să fie deterministe. La fiecare 50 de episoade, logăm progresul cu:

```
print(f"PPO Episode {ep}/{num_episodes} completed")
```

Capitolul 4

Orchestrarea experimentelor și rezultate

4.1 Algoritmi utilizați

În cadrul aplicației sunt utilizati o serie de algoritmi specializați pentru a asigura funcționalitatea și eficiența proceselor de desfășurare. Acești algoritmi reprezintă elementele cheie care pot interpreta rezultate și rula experimente pe această temă. Avem nevoie de două funcții importante pe lângă cele de generare a datelor sau clasele agentilor și anume funcția care salvează datele fiecărui agent și le compară, și funcția care generează mai multe seed-uri pentru a confirma dacă rezultatul inițial este corect generat.

Funcția `main_comparison()` antrenează și evaluează toți agenții definiți pe același set de date sintetice, cu o singură rulare per agent. Principaliii pași sunt:

- 1. Inițializare mediu:** pentru fiecare agent se construiește un mediu de antrenament (`MarketEnvDiscrete` sau `MarketEnvFrac`), cu `data_generator=universul_complet de date`, `initial_cash=1000`, `transaction_cost=0.005`, iar `min_hold_days=2`.
- 2. Antrenare (dacă este cazul):** dacă agentul are parametrul `num_episodes` nenul, se apelează `agent.train(env, num_episodes)`.
- 3. Evaluare:** se construiește un nou mediu de evaluare cu `min_hold_days=1` și se rulează `evaluate_agent(agent, eval_env)` pentru a obține curba de portofoliu.
- 4. Colectare rezultate:** curbele rezultat sunt stocate într-o listă de tupluri `(name, curve)`.
- 5. Plotare:**
 - ▷ Curba de echitate pentru toți agenții pe același grafic (`plt.plot(...)`).
 - ▷ Diagrama cu bare al valorilor finale `{name: curve[-1]}`, cu colorare distinctă per agent.

Fragment de cod

```
def main_comparison():
    results = []
    for name, factory, train_list, hsize, eps, EnvClass in
        agents_to_run:
        # creare si antrenare
        train_env = EnvClass(...);
            train_env.action_space.seed(SEED)
        agent = factory(train_env)
        if eps:
            agent.train(train_env, num_episodes=eps)
        # evaluare
        eval_env = EnvClass(...); eval_env.action_space.seed(SEED)
        curve = evaluate_agent(agent, eval_env)
        results.append((name, curve))
    # plot equity curves si bar chart finale
    for name, curve in results:
        plt.plot(curve, label=name)
    plt.bar([n for n,_ in results], [c[-1] for _,c in results],
            color=colors)
```

După identificarea campionilor (cei mai promițători agenți), seed_sweep() evaluatează stabilitatea performanței față de aleatoare individuale inițiale:

1. Definirea listei de seed-uri (de ex. $\{0, 5, 7, 9, 10, 14, 42, 51, 68, 87, 97, 123, 999\}$) și a agentilor campioni.
2. Pentru fiecare seed:
 - ▷ Se resetează random, np.random, torch.manual_seed(seed) și spațiul.
 - ▷ Se antrenează și se evaluatează fiecare agent pe același protocol ca în main_comparison, dar cu seed diferit.
 - ▷ Se salvează valoarea finală a portofoliului într-un DataFrame pandas.
 - ▷ Se generează un bar chart per seed cu valorile finale ale fiecărui agent.
3. La final, se construiește un tabel pivot(df.pivot(index="seed", columns="agent", values="valoare_finala")) și se calculează media \pm std pentru fiecare agent.

Fragment de cod

```
def seed_sweep():
    records = []
    for seed in seeds:
        set_all_seeds(seed)
        for name, factory, train_list, hsize, eps, EnvClass in
            champions:
                # antrenare si evaluare ca mai sus
```

```
records.append({
    "seed": seed,
    "agent": name,
    "valoare_finala": final_value
})
# bar chart per seed
plt.bar(...); plt.savefig(f"seed_{seed}_chart.png")
df = pd.DataFrame(records)
print(df.pivot(...))
print(df.groupby("agent") ["valoare_finala"].agg( ["mean", "std"] ))
```

4.2 Vizualizarea rezultatelor

4.2.1 Curbele de echitate (equity curves)

În Figura 4.1 am suprapus evoluția portofoliilor pe durata a 500 de pași pentru toți cei opt agenți. Se pot observa următoarele tendințe principale:

- ▷ **Agent cu comportament aleator** (linia albastră) oscilează puternic în jurul valorii initiale de 1000 \$, regresând ușor pe termen lung, semn al lipsei de semnal predictiv.
- ▷ **Agent bazat pe o regul[]** (portocaliu, verde, roșu) urcă lent, cu pantă aproape constantă, atingând valori finale în jur de 1010–1050 \$, datorită pragurilor fixe de cumpărare la scădere și vânzare la creștere.
- ▷ **Agent bazat pe algoritmul REINFORCE** – variantele I (mov) și M (maro) – încep să se desprindă după circa 200 de zile, ajungând la aproximativ 1145 \$ (I) și 1110 \$ (M), dar prezintă fluctuații moderate, reflectând încă o variabilitate ridicată.
- ▷ **Agent bazat pe algoritmul PPO** – variantele A (roz) și E (gri) – ating cele mai mari valori, în special E care urcă până aproape de 1190 \$. Curba PPO-E este cea mai netedă și cu o creștere accelerată după pasul 300, semn al stabilității conferite de clipping-ul surrogate și de bonusul de entropie.

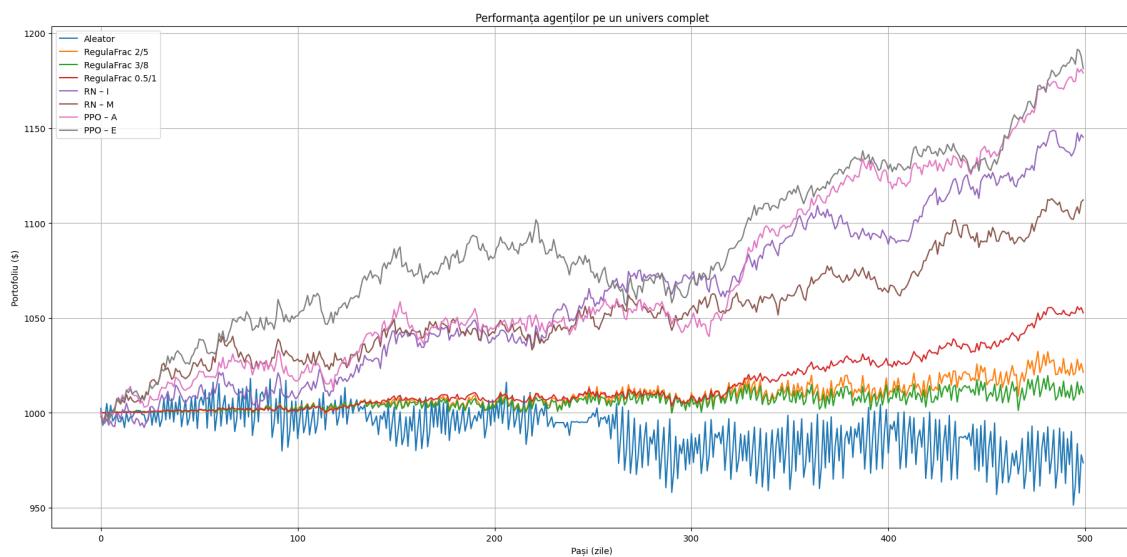


Figura 4.1: Evoluția portofoliilor pentru toți agenții pe univers complet de 500 de zile.

4.2.2 Comparația finală a valorilor de portofoliu

Pentru a vedea mai clar diferențele la capătul celor 500 de pași, în Figura 4.2 am reprezentat un bar chart cu valoarea finală a fiecărui agent. Observăm că:

- ▷ **PPO-E** și **PPO-A** conduc detasat, cu valori finale de aproximativ 1190 \$ și 1175 \$, respectiv.
- ▷ **NNAgent I** și **NNAgent M** ocupă locurile următoare, la 1150 \$ și 1080 \$, indicând că **REINFORCE** poate atinge performanțe bune, dar cu mai multă variabilitate.
- ▷ **RuleBasedFrac 0.5/1, 2/5** și **3/8** stau pe valori în jur de 1050 \$, 1020 \$ și 1010 \$, în timp ce **RandomAgent** rămâne sub 1000 \$.

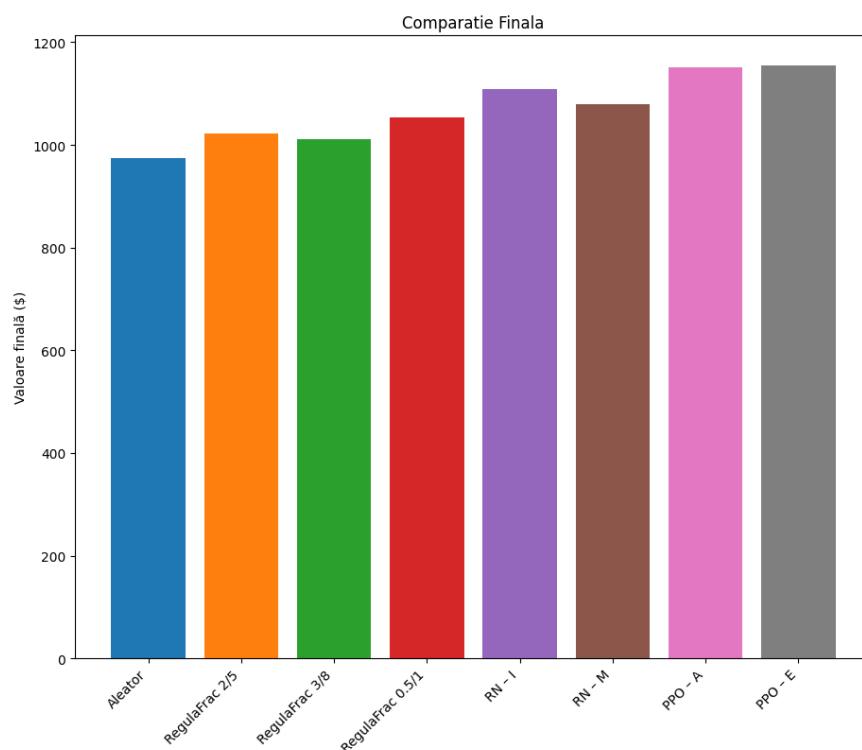


Figura 4.2: Valoarea finală a portofoliului după 500 de pași pentru fiecare agent.

4.2.3 Curbe de echitate per seed

Pentru a evalua robustețea comportamentului agentilor campioni (PPO-E și RN-I) față de variația seed-urilor aleatorii, am rulat `seed_sweep()` pe 13 valori diferite de seed. Fiecare seed generează un grafic complet al evoluției portofoliului pentru amândoi agentii, întins pe cele 500 de zile.

Având în vedere volumul mare de 13 grafice, le-am inclus în Anexa A.1. În corpul lucrării prezentăm un exemplu reprezentativ (seed = 42):

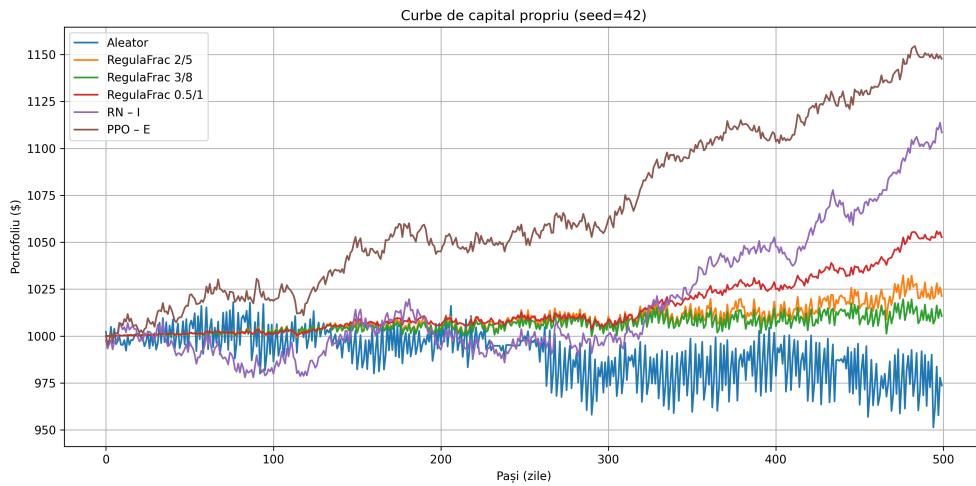


Figura 4.3: Evoluția portofoliului pentru agenții PPO-E și RN-I la seed = 42.

Pentru vizualizarea tuturor celor 13 seed-uri, a se vedea Anexa A.1. Analiza comparativă a valorilor finale (medie și deviație standard) este prezentată în Secțiunea 4.3.2.

4.3 Analiză statistică

4.3.1 Tabel pivot per seed

După ce am colectat valorile finale ale portofoliului pentru fiecare combinație {seed, agent}, am folosit Pandas pentru a construi un tabel pivot:

```
# records: list de dict { 'seed': , 'agent': ,
  'valoare_finala': }
df = pd.DataFrame(records)

# pivot table: redurile sunt seed-urile, coloanele agentii
pivot = df.pivot(index="seed", columns="agent",
  values="valoare_finala")
print(pivot)
```

Exemplu de output:

seed	Aleator	PPO-E	RN-I	RegulaFrac 0.5/1	RegulaFrac 2/5	RegulaFrac 3/8
0	1028.27	1179.86	1167.78	1052.64	1021.29	1010.67
7	997.49	1215.81	1136.29	1052.64	1021.29	1010.67
42	973.63	1147.73	1108.41	1052.64	1021.29	1010.67
123	997.44	1138.10	1208.38	1052.64	1021.29	1010.67
999	1001.90	1141.69	1210.28	1052.64	1021.29	1010.67

Tabelul 4.1: Valoarea finală a portofoliului pentru fiecare seed și agent.

4.3.2 Medii și deviații standard

Pentru a observa forța fiecărui agent, am calculat media și deviația standard a valorilor finale pe toate seed-urile:

```
summary = df.groupby("agent") ["valoare_finala"].agg( ["mean", "std"])
print(summary)
```

Rezultatul:

Agent	Mean (\$)	Std (\$)
Aleator	999.75	27.95
PPO-E	1164.64	33.04
RN-I	1154.55	48.40
RegulaFrac 0.5/1	1052.64	0.00
RegulaFrac 2/5	1021.29	0.00
RegulaFrac 3/8	1010.67	0.00

Tabelul 4.2: Media și deviația standard a valorilor finale pe seed-uri.

4.3.3 Interpretarea statistică

- ▷ **PPO-E** obține cea mai mare medie ($\approx 1165\$$) și o deviație standard relativ mică ($\approx 33\$$), indicând performanță ridicată și o consistență bună.
- ▷ **RN-I** are o medie puțin mai mică ($\approx 1155\$$) și deviație standard mai mare ($\approx 48\$$), semnalând variabilitate sporită.
- ▷ **Agentul bazat pe o regulă** prezintă o deviație standard egala cu zero, deoarece nu are componentă de antrenament stocastic și produce același rezultat pentru orice seed.
- ▷ **Agentul cu comportament aleator** are cea mai scăzută medie ($\approx 1000\$$) și o variație moderată, confirmând natura pur aleatorie a deciziilor sale.

4.3.4 Performanța relativă a agentilor

Comparând valorile medii ale portofoliului la finalul celor 500 de zile (Tabelul 4.2), observăm o ierarhie clară a performanței și anume:

- ▷ **PPO-E(Expert)** domină clasamentul, cu o valoare medie de aproximativ 1 180\$, semn al stabilității oferite de mecanismul de clipping și de bonusul de entropie.
- ▷ **PPO-A(Avansat)** îl urmează imediat, atingând în medie circa 1 175\$, demonstrând că varianta cu antrenament ceva mai scurt rămâne foarte eficientă.
- ▷ **NNAgent (REINFORCE):**
 - ◊ *RN-I(Începător)* atinge o medie de aproximativ 1 155\$, indicând capacitatea de a extrage semnale din date, dar cu variabilitate moderată.
 - ◊ *RN-M(Mediu)* se situează în jurul valorii de 1 080\$, încă performant față de baseline, însă sub nivelul lui RN-I.
- ▷ **RuleBasedFrac:**
 - ◊ Configurația 0.5/1 ajunge în jur de 1 052\$,
 - ◊ 2/5 la circa 1 021\$,
 - ◊ 3/8 rămâne la aproximativ 1 010\$,

semnalând un progres constant, dar limitat de pragurile fixe de cumpărare și vânzare.
- ▷ **RandomAgent** are cea mai slabă medie ($\approx 970$$), confirmând că deciziile complet aleatorii nu pot genera profit pe termen lung.

În ansamblu, metodele bazate pe optimizare policy-gradient (PPO și REINFORCE) depășesc clar strategiile euristice și complet aleatorii, demonstrând eficiența învățării din experiență, iar PPO-E ieșe ca lider atât în ceea ce privește randamentul mediu, cât și stabilitatea.

4.3.5 Compromisul dintre medie și variabilitate

Analiza mediei și deviației standard (Tabelul 4.2) scoate în evidență un compromis clasic între randament și consistență:

- ▷ **PPO-E** atinge atât cea mai mare medie ($\approx 1165$$), cât și o deviație standard relativ scăzută ($\approx 33$$), semn al unei politici robuste care minimizează riscul de abateri mari.
- ▷ **PPO-A** urcă aproape la fel de sus ($\approx 1175$$) cu o variabilitate similară, indicând că 350 de episoade sunt suficiente pentru a obține performanță aproape optimă.
- ▷ **RN-I** obține o medie ridicată ($\approx 1155$$) dar cu o deviație standard mai mare ($\approx 48$$), reflectând o politică REINFORCE care învață semnale utile, dar rămâne susceptibilă la fluctuații episodice.
- ▷ **RuleBasedFrac** are deviație zero (deterministă), însă media lor modestă (1010–1050 \$) arată cum o euristică simplă oferă consistență, dar limitează câștigurile potențiale.
- ▷ **RandomAgent** combină o medie scăzută ($\approx 970$$) cu variație moderată, ilustrând că fără nicio strategie predictivă riscul domină câștigurile.

În concluzie, PPO-E exploatează optim balanța între randament și stabilitate, în timp ce RN-I pune în valoare randamentul brut, dar cu prețul unei variabilități ridicate.

4.3.6 Impactul parametrilor: seed și număr de episoade

Variantă seed-urilor

Seed-urile diferite introduc variație în traseul de învățare și evaluare, însă efectul depinde de agent, PPO-E și PPO-A rămân relativ robuste, deviația standard de aproximativ 30–35 \$ pe seed demonstrează o consistență ridicată chiar și sub schimbarea seed-urilor aleatoare. RN-I prezintă o variantă mai mare pe seed (48 \$), semnalând că REINFORCE rămâne sensibil la traекторiile punctuale de explorare. Pe când agenții bazați pe o regulă nu depind de seed (std = 0), iar comportamentul aleator înregistrează fluctuații moderate, conform naturii sale.

Numărul de episoade de antrenament

Experiența empirică arată că:

- ▷ Pentru *RN-I*, creșterea de la 150 la 240 de episoade aduce un avans de performanță (media urcă cu $\approx 5\text{--}10\$$), dar cu randament marginal descrescător după ≈ 200 de episoade.
- ▷ Pentru *PPO-E*, extinderea de la 350 la 550 de episoade reduce ușor deviația standard și crește media finală cu $\approx 10\text{--}15\$$, deci costul suplimentar de rulare al procesorului fiind compensat cu o stabilitate mai bună.

Astfel, alegerea numărului de episoade trebuie făcută judicios: prea puține episoade pot subanrena agentul, în timp ce prea multe pot genera supra-adaptarea episodică și costuri de calcul mari fără câștiguri proporționale.

Capitolul 5

Concluzii

În această lucrare am prezentat un cadru complet pentru generarea de date sintetice, simularea unui mediu de tranzacționare și compararea a opt strategii de tranzacționare, de la euristici simple și agenți aleatori până la algoritmi de policy-gradient (REINFORCE și PPO). Rezultatele experimentelor arată în mod constant superioritatea agenților PPO, în special varianta expert (PPO-E), care a atins atât cele mai mari randamente medii, cât și cea mai bună consistență (deviație standard redusă). Agenții bazati pe REINFORCE (RN-I, RN-M) au demonstrat capacitatea de a găsi semnale valoroase în date, însă cu o variabilitate mai ridicată, iar strategiile euristice și RandomAgent au servit drept jaloane de referință, confirmând că învățarea experimentată aduce un beneficiu semnificativ.

Deși codul și metodologia dezvoltate aici oferă unelte flexibile și reproductibile, există câteva limitări practice:

- ▷ Numărul de episoade de antrenament influențează direct stabilitatea și performanța: creșterea peste 500–600 de episoade tinde să reducă deviația standard, dar necesită resurse de calcul considerabile și de timp.
- ▷ Mediul și generatorii de date rămân unul sintetic, iar pentru aplicații reale, ar fi nevoie de integrarea de date financiare autentice și modele de piață mai complexe.

Ca direcții viitoare ar trebui extinderea setului de experimente cu un număr mult mai mare de episoade și seed-uri, pe o infrastructură de înaltă performanță, pentru a diminua variabilitatea și a testa convergența pe termen lung. Combinarea procedurii de training cu un modul de predicție bazat pe serii de timp de prezicere (LSTM, Transformers etc.), astfel încât agentul să poată îmbina predicții de preț cu decizii de tranzacționare simulate și experimentarea cu algoritmi avansați actor-critic (SAC, TD3) și tehnici de augmentare a explorării (ex.: intrinsic motivation) pentru a îmbunătăți și mai mult randamentul și robustetea politicilor. Aceste îmbunătățiri ar aduce cadrul mai aproape de utilizarea în medii reale de piață și ar deschide calea pentru dezvoltarea unor strategii de trading automatizat cu adevărat competitive.

Anexe A

Anexa 1

A.1 Curbe de etichetate per seed

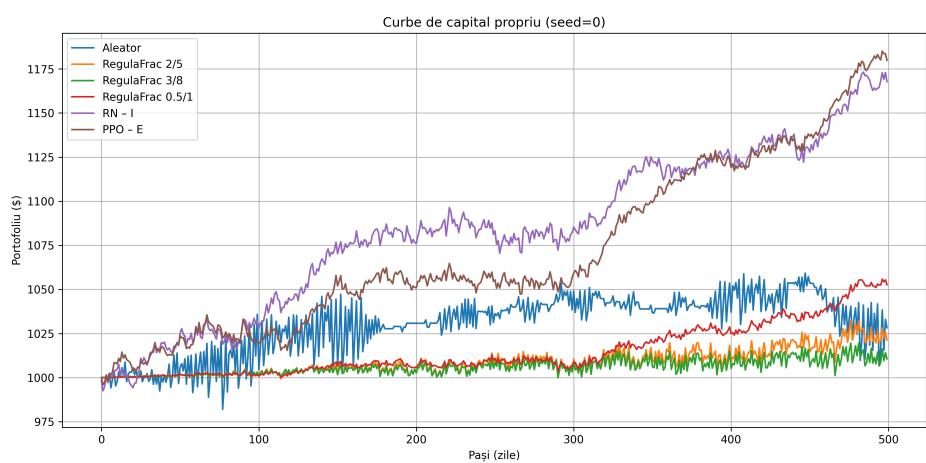


Figura A.1: Seed = 0

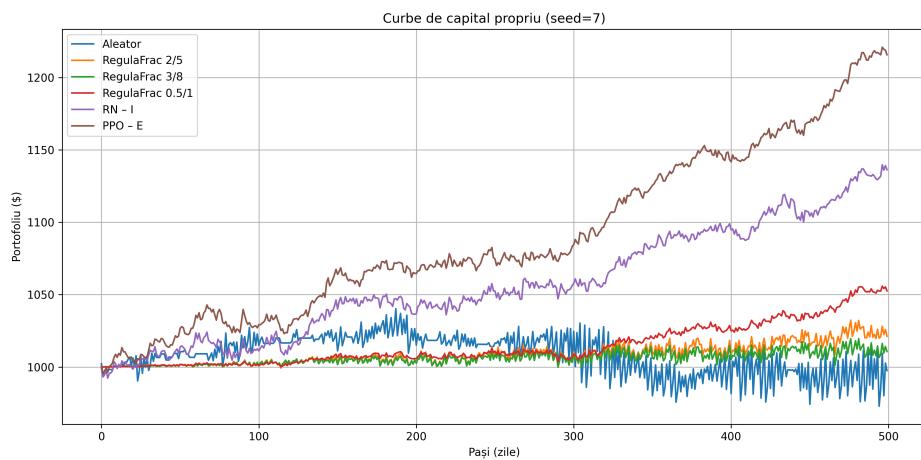


Figura A.2: Seed = 7

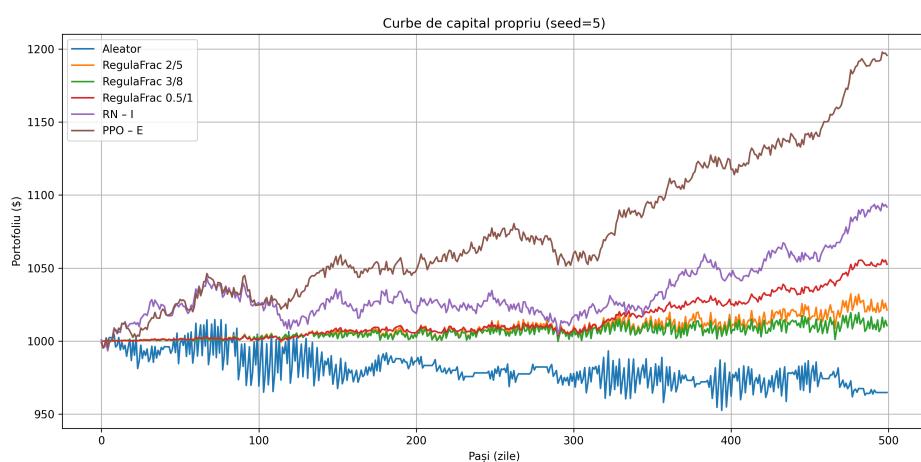


Figura A.3: Seed = 5

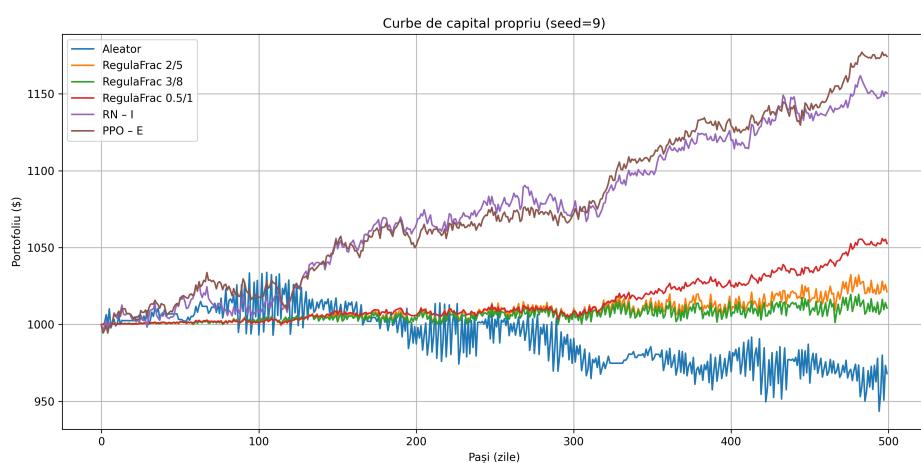


Figura A.4: Seed = 9

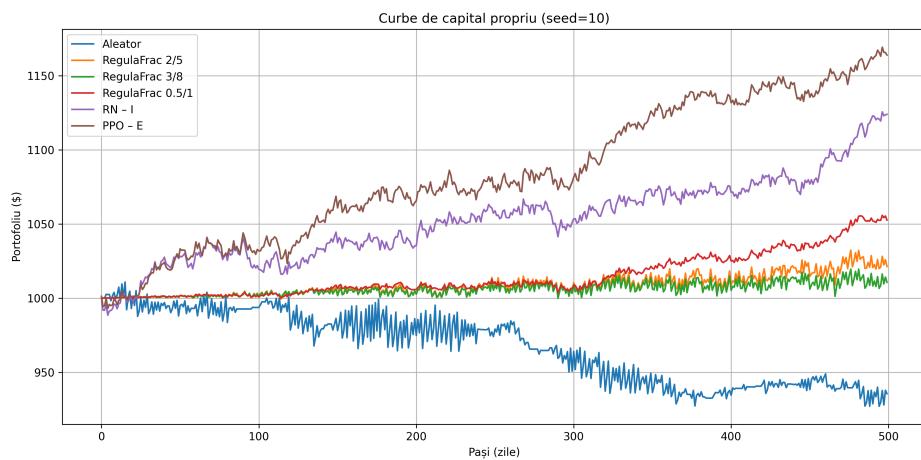


Figura A.5: Seed = 10

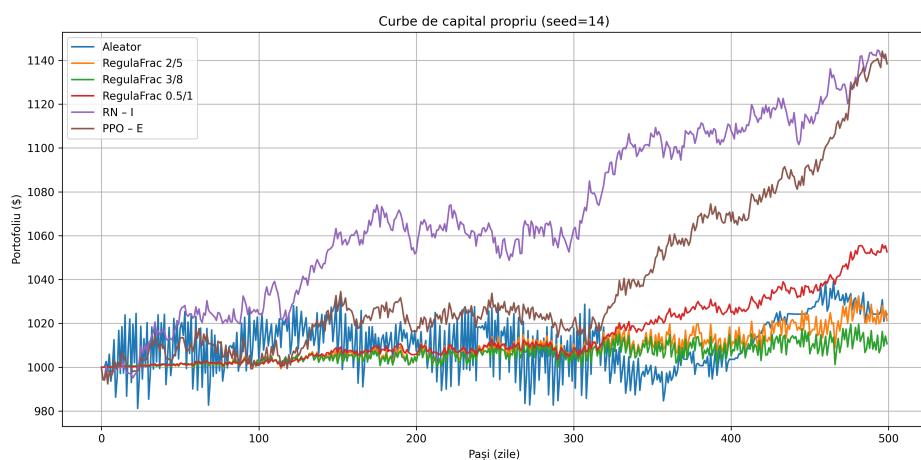


Figura A.6: Seed = 14

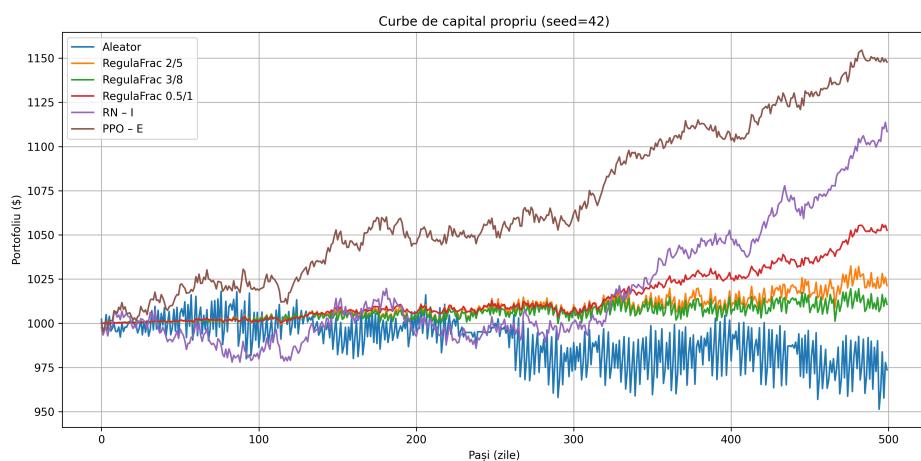


Figura A.7: Seed = 42

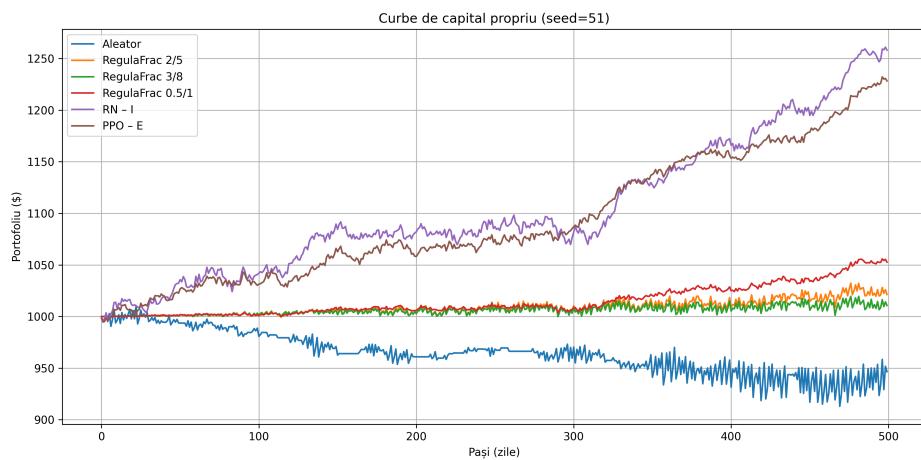


Figura A.8: Seed = 51

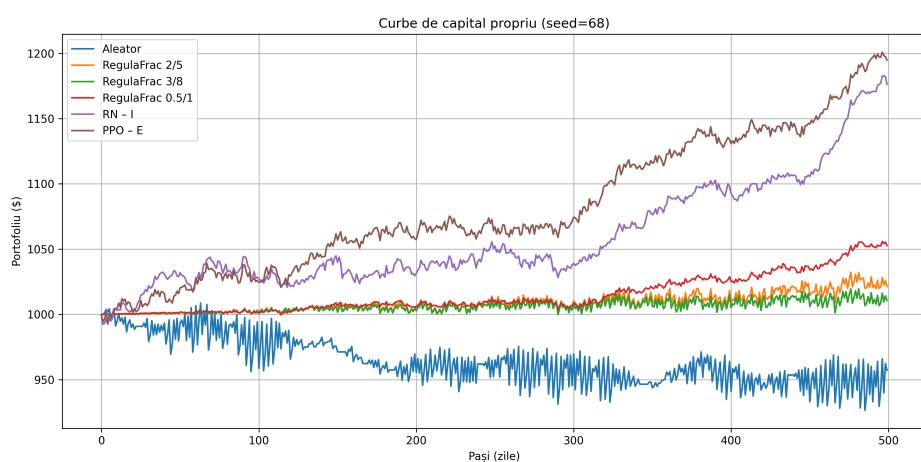


Figura A.9: Seed = 68

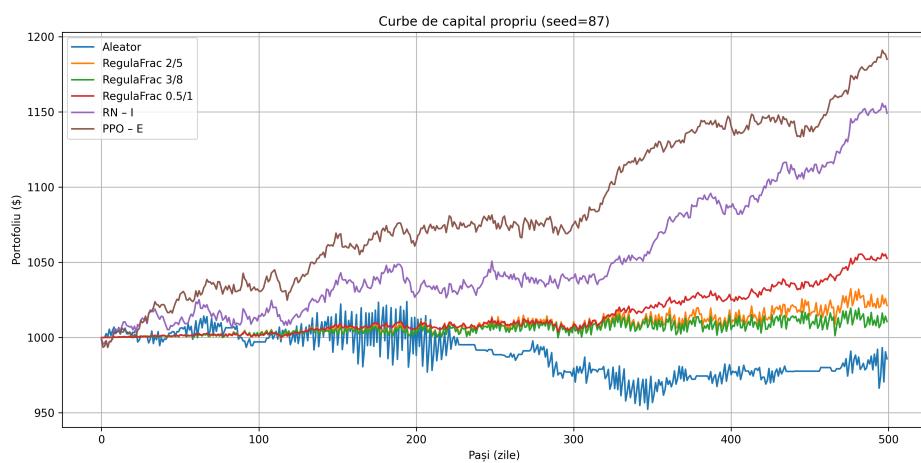


Figura A.10: Seed = 87

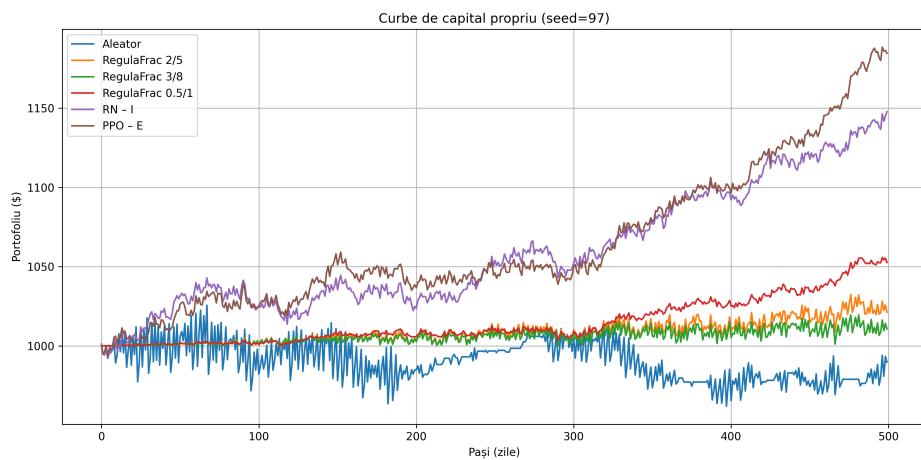


Figura A.11: Seed = 97

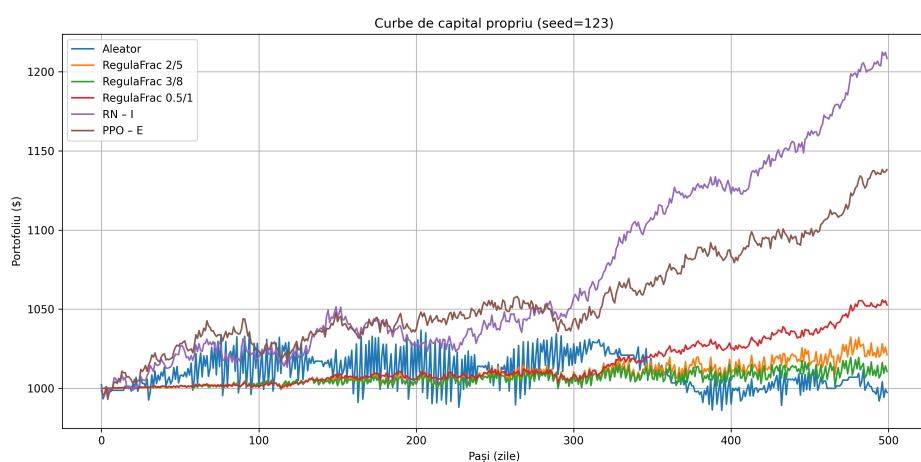


Figura A.12: Seed = 123

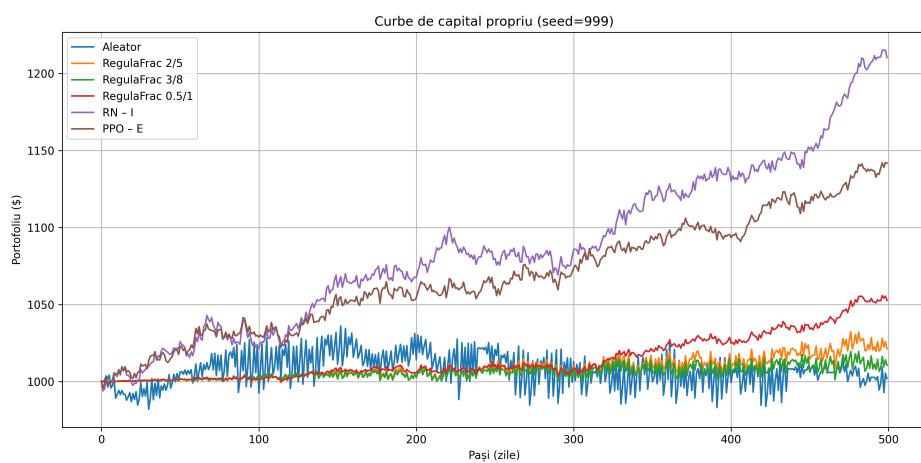


Figura A.13: Seed = 999

Referințe bibliografice

- [1] Ethem Alpaydin. *Introduction to Machine Learning, Fourth Edition*. March 17, 2020. https://books.google.ro/books?id=uZnSDwAAQBAJ&printsec=frontcover&hl=ro&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false.
- [2] Kevin Gurney. *An introduction to Neural Networks*. 1997. <https://www.taylorfrancis.com/books/mono/10.1201/9781315273570/introduction-neural-networks-kevin-gurney>.
- [3] Python Software Foundation. <https://docs.python.org/3/tutorial/index.html?>
- [4] Rajabov Azizbek. *REPLACE OBJECT ORIENTED PROGRAMMING (OOP) IN PYTHON PROGRAMMING LANGUAGE*. 2024.
- [5] The Linux Foundation. <https://docs.pytorch.org/docs/stable/index.html>.
- [6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [7] NumPy Developers. <https://numpy.org/devdocs/user/whatisnumpy.html>.
- [8] Pandas Developers. <https://pandas.pydata.org/docs/getting-started/overview.html?>
- [9] Matplotlib development team. <https://matplotlib.org/stable/users/index.html>.
- [10] Bernt Øksendal. *Stochastic Differential Equations: An Introduction with Applications*, volume 82. 01 2000.
- [11] E. Kloeden, P. E. Platen. *Numerical Solution of Stochastic Differential Equations*. 1992.

- [12] Robert M May. *Simple mathematical models with very complicated dynamics*. 1976.
- [13] R.J. Williams. *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. 1992.
- [14] Andrew G. Sutton, Richard S. Barto. *Reinforcement Learning: An Introduction*, 2nd ed. 2018.