Universitatea POLITEHNICA din București
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
060042 București, Splaiul Independenței, nr. 313, sector 6

LUCRARE DE DISERTAȚIE

RAPORT DE CERCETARE – SEMESTRUL II

LOAD BALANCING NEURAL NETWORK: O ABORDARE BAZATĂ PE DEEP REINFORCEMENT LEARNING PENTRU OPTIMIZAREA APLICAȚIILOR WEB

Autor: Ștefan PATRICHE

Conducători științifici:

Profesor Dr. Ing. Mihai DASCĂLU

Conf. Dr. Ing. Ștefan RUȘETI

București

Iunie 2025

CUPRINS

1 INTRODUCERE ȘI OBIECTIVE

1.1 Contextul continuării cercetării

În primul semestru al cercetării am realizat o analiză a domeniului de load balancing-ului și a aplicațiilor de Deep Reinforcement Learning în sisteme distribuite. Această analiză a evidențiat o lacună semnificativă în literatura de specialitate: în timp ce există numeroase implementări de load balancing bazate pe DRL pentru centrele de date, aplicarea acestor tehnici pentru aplicațiile web rămâne relativ neexplorată.

Pe baza concluziilor din primul semestru, am identificat oportunitatea de a adapta și simplifica tehnicile existente din domeniul centrelor de date pentru contextul specific al aplicațiilor web. Această adaptare necesită nu doar o implementare tehnică, ci și o reconsiderare fundamentală a modului în care un agent de Deep Reinforcement Learning poate funcționa ca load balancer direct, nu doar ca un controler al unor sisteme tradiționale de distribuire a sarcinii.

1.2 Obiectivele semestrului II

Obiectivul principal al acestui semestru a fost prototiparea unei infrastructuri complete și funcționale pentru sistemul "Load Balancing Neural Network" (LBNN), cu accent pe următoarele componente cheie.

Obiective tehnice:

- Implementarea unei arhitecturi de simulare bazată pe containere Docker, care să permită testarea și antrenarea algoritmilor de load balancing.
- Dezvoltarea unui sistem de simulare pentru încărcarea serverelor bazat pe tick-uri discrete.
- Crearea unui mecanism de sincronizare între toate componentele sistemului pentru a asigura consistența datelor.
- Implementarea unui pipeline automat de colectare și analiză a datelor pentru antrenarea viitoare a modelului DRL.

Obiective de cercetare:

- Designul și testarea unei funcții de recompensă care să fie invariantă la nivelul general de încărcare al sistemului.

- Explorarea și soluționarea provocărilor practice legate de integrarea unui agent DRL în rolul direct de load balancer.
- Validarea arhitecturii propuse prin implementarea unui algoritm baseline (Round Robin) și măsurarea performanței acestuia.

Rezultate obținute: Infrastructura dezvoltată a fost validată cu succes, demonstrând funcționalitatea completă a tuturor componentelor. Testarea cu algoritmul Round Robin a stabilit un baseline de performanță de 40% decizii optime din totalul de 20 de cereri testate, cu recompense variind între -28.0 și +38.595. Sistemul de recompense bazat pe gradient s-a dovedit eficient în oferirea de semnale clare pentru învățare, fiind invariant la nivelul de încărcare al sistemului. Aceste rezultate creează fundamentul solid pentru implementarea și antrenarea algoritmilor de Deep Reinforcement Learning în fazele următoare ale proiectului.

1.3 Metodologia de implementare

Abordarea adoptată în acest semestru a fost una iterativă și experimentală, bazată pe principiul “build-measure-learn”. Am început cu componentele fundamentale ale sistemului și am construit gradual complexitatea, testând și validând fiecare etapă înainte de a continua.

Metodologia tehnică:

1. **Prototipare incrementală** - Fiecare componentă a fost dezvoltată și testată independent înainte de integrare.
2. **Containerizare completă** - Utilizarea Docker pentru a asigura reproductibilitatea și izolarea componentelor.
3. **Testare continuă** - Validarea funcționalității după fiecare modificare majoră.
4. **Documentare detaliată** - Înregistrarea tuturor deciziilor tehnice și a motivațiilor din spatele acestora.

Metodologia de cercetare:

1. **Experimentare practică** - Testarea ideilor teoretice prin implementare și măsurare.
2. **Analiza erorilor** - Documentarea detaliată a problemelor întâmpinate și a soluțiilor găsite.
3. **Iterație bazată pe feedback** - Modificarea designului pe baza rezultatelor observate în testare.

Această metodologie s-a dovedit deosebit de valoroasă în identificarea și rezolvarea unor probleme fundamentale în designul sistemului de recompense, așa cum va fi detaliat în capitolele următoare.

2 ARHITECTURA SISTEMULUI IMPLEMENTAT

2.1 Conceptul fundamental - Agentul ca Load Balancer

Una dintre deciziile arhitecturale cele mai importante ale acestui proiect a fost alegerea ca agentul de Deep Reinforcement Learning să funcționeze direct ca load balancer, nu doar ca un controler al unui sistem tradițional de distribuire a sarcinii precum HAProxy sau NGINX (sisteme care decid dinamic algoritmul care este utilizat la acel moment pentru distribuirea saricinii).

Motivația acestei abordări:

Soluțiile tradiționale de load balancing bazate pe DRL din literatura de specialitate operează de obicei prin controlul parametrilor unor sisteme existente - de exemplu, ajustarea ponderilor în algoritmi WCMP (Weighted Cost Multi-Path) sau modificarea configurărilor de rutare. Această abordare, deși validă, introduce un nivel suplimentar de abstracție între decizia agentului și rezultatul final.

În schimb, prin poziționarea agentului direct în calea cererilor HTTP, obținem:

- **Control direct** asupra fiecărei decizii de rutare.
- **Feedback imediat** asupra consecințelor deciziilor luate.
- **Flexibilitate completă** în algoritmul de luare a deciziilor.
- **Simplitate arhitecturală** prin eliminarea intermediarilor.

Această abordare permite testarea pură a capacităților de învățare ale algoritmului DRL, fără interferența sau limitările unui sistem de load balancing preexistent.

2.2 Componente principale și integrarea lor

Sistemul implementat constă în cinci componente principale, fiecare containerizate pentru izolare și reproductibilitate.

2.2.1 Agentul de Load Balancing

Rol: Componenta centrală care primește toate cererile HTTP și decide către care server să le direcționeze.

Implementare: Serviciu Flask care expune endpoint-ul `/route_request` pentru primirea cererilor de la k6. În faza actuală, implementează Round Robin ca algoritm de bază, dar arhitectura este pregătită pentru înlocuirea cu un model DRL.

Funcționalități cheie:

- Menținerea stării curente a tuturor serverelor.
- Luarea deciziilor de rutare.
- Coordonarea episoadelor de antrenare.
- Comunicarea sincronizată cu toate serverele.

2.2.2 Serverele Web (3 instanțe)

Rol: Simularea unei infrastructuri eterogene de servere cu capacități diferite.

Capacități configurate:

- Server-1: 1500 unități CPU, 2000 unități memorie
- Server-2: 3500 unități CPU, 3200 unități memorie
- Server-3: 5000 unități CPU, 3000 unități memorie

Funcționalități:

- Procesarea cererilor reale și a cererilor de sincronizare.
- Simularea consumului de resurse pe baza metadatelor din cereri.
- Calcularea automată a stării curente (procentaj utilizare CPU/memorie).
- Expirarea automată a cererilor după durata specificată în tick-uri.

2.2.3 Generatorul de Trafic (k6)

Rol: Generarea de cereri variate pentru testarea sistemului.

Tipuri de cereri generate:

- **Light:** 50 unități CPU, 30 unități memorie, durată 2 tick-uri.
- **Medium:** 150 unități CPU, 100 unități memorie, durată 4 tick-uri.
- **Heavy:** 300 unități CPU, 200 unități memorie, durată 6 tick-uri.

2.2.4 Componenta de Antrenare (Trainer)

Rol: Analiza episoadelor completate și calculul recompenselor pentru antrenarea viitoare.

Funcționalități:

- Colectarea istoricului complet din toate serverele.
- Calcularea recompenselor.
- Generarea de analize detaliate ale performanței.
- Salvarea rezultatelor pentru procesare ulterioară.

2.2.5 Sistemul de Monitorizare

Fiecare componentă expune endpoint-uri de monitoring (`/health`, `/metrics`) pentru observabilitatea sistemului.

2.3 Fluxul de date și sincronizarea

Una dintre provocările majore ale implementării a fost asigurarea sincronizării perfecte între toate componentele. Am dezvoltat un sistem bazat pe "tick-uri" care garantează consistența stării sistemului la fiecare decizie.

2.3.1 Protocolul de sincronizare

Fluxul pentru fiecare cerere:

1. **k6** trimite o cerere către **Agent.**
2. **Agentul** analizează starea curentă a serverelor și ia o decizie de rutare.
3. **Agentul** trimite cererea reală către serverul ales și cereri goale către celelalte servere (pentru sincronizare).
4. **Toate serverele** procesează tick-ul și returnează starea lor actualizată în răspuns.
5. **Agentul** actualizează starea internă pe baza răspunsurilor primite.
6. **Agentul** răspunde la k6 confirmând procesarea cererii.

2.3.2 Sistemul de tick-uri

Fiecare interacțiune din sistem este asociată cu un **tick_id** unic și secvențial. Acest mecanism oferă următoarele avantaje:

- **Sincronizare perfectă:** Toate serverele înregistrează aceeași stare pentru același moment de decizie.
- **Trasabilitate completă:** Fiecare decizie poate fi reconstruită exact cu contextul în care a fost luată.
- **Analiza contrafactuală:** Putem determina care ar fi fost decizia optimă pentru fiecare tick.
- **Eliminarea race condition-urilor:** Ordinea evenimentelor este strict controlată.

2.3.3 Managementul episoadelor

Sistemul operează în **episoade** de lungime fixă (configurabilă, testat cu 5-20 cereri). La sfârșitul fiecărui episod:

1. **Agentul** detectează automat completarea episodului.

2. **Agentul** declanșează **Trainer-ul** pentru analiza datelor.
3. **Trainer-ul** colectează istoricul complet de la toate serverele.
4. **Trainer-ul** calculează recompensele și salvează analiza.
5. **Agentul** resetează contoarele pentru episodul următor.

Această arhitectură asigură un pipeline complet automatizat pentru colectarea datelor de antrenare, eliminând intervenția manuală și reducând riscul erorilor.

3 IMPLEMENTAREA INFRASTRUCTURII DE SIMULARE

3.1 Containerizarea componentelor

Alegerea containerizării cu Docker pentru toate componentele sistemului s-a dovedit a fi una dintre deciziile tehnice cele mai înțelepte ale implementării. Această abordare a oferit multiple beneficii care s-au materializat pe parcursul dezvoltării proiectului.

3.1.1 Structura arhitecturală

Sistemul este organizat ca o aplicație multi-container orchestrată prin Docker Compose, cu următoarea structură:

```
LBNN/
├── Agent/              # Agentul de load balancing (port 8080)
├── Containers/
│   ├── web-server-1/   # Server cu capacitate mică (port 8081)
│   ├── web-server-2/   # Server cu capacitate medie (port 8082)
│   └── web-server-3/   # Server cu capacitate mare (port 8083)
├── Trainer/            # Componenta de antrenare (port 8084)
├── K6/                 # Generatorul de trafic
├── models/             # Volum partajat pentru rezultate
└── docker-compose.yml  # Orchestrarea sistemului
```

Tabel 1

3.1.2 3.1.2 Beneficiile containerizării

Izolarea componentelor: Fiecare serviciu rulează în propriul mediu izolat, eliminând conflictele de dependențe și asigurând reproductibilitatea.

Rețeaua internă: Docker Compose creează automat o rețea privată (`lbnn-network`) care permite comunicarea între containere folosind nume simbolice (ex: `http://lbnn-server-1:8081`) în loc de adrese IP hard-codate.

Scalabilitatea dezvoltării: Modificările într-o componentă nu afectează celelalte componente, permițând dezvoltarea paralelă și testarea incrementală.

Reproductibilitatea: Sistemul poate fi pornit identic pe orice mașină cu Docker, eliminând problemele de “works on my machine”.

3.1.3 Provocări întâmpinate

Ordinea de pornire: Inițial, am întâmpinat probleme cu ordinea în care containerele deveneau disponibile. Am rezolvat prin implementarea unui mecanism de retry în agentul de load balancing pentru inițializarea stărilor serverelor.

Sincronizarea volumelor: Partajarea directorului `models/` între agent și trainer pentru salvarea rezultatelor a necesitat configurarea corectă a permisiunilor și path-urilor.

3.2 Sistemul de simulare a încărcării bazat pe tick-uri

Una dintre provocările majore a fost crearea unui sistem de simulare realist care să nu depindă de resursele hardware reale ale containerelor Docker.

3.2.1 Problema inițială

Prima abordare considerată folosea funcția `time.sleep()` pentru simularea timpului de procesare și monitorizarea reală a CPU-ului și memoriei containerelor Docker prin `docker stats`. Această abordare s-a dovedit problematică din mai multe motive:

- **Scalele de timp inadecvate:** Simularea cu 1000 de cereri în 2-3 secunde făcea ca cererile să se acumuleze artificial, toate având durate de 500-1000ms
- **Resurse hardware irelevante:** Operațiunile `sleep()` nu consumă CPU real, astfel metricile Docker nu reflectau încărcarea simulată
- **Lipsa controlului:** Nu aveam control precis asupra pattern-urilor de încărcare

3.2.2 Soluția bazată pe tick-uri

Soluția a constat în dezvoltarea unui sistem complet nou bazat pe conceptul de “tick-uri discrete”.

Principiul fundamental: În loc de timp real, sistemul operează în “tick-uri” numerotate secvențial (1, 2, 3, ...). Fiecare cerere specifică durata în numărul de tick-uri pentru care va consuma resurse.

Exemplu practic:

- **Cerere light:** durată 2 tick-uri.
- **Cerere medium:** durată 4 tick-uri.
- **Cerere heavy:** durată 6 tick-uri.

Implementarea:

```
python
```

```
# În loc de:
time.sleep(duration_ms / 1000)

# Folosim:
server_state["active_requests"].append({
    "tick_id": current_tick,
    "expires_at_tick": current_tick + duration_ticks,
    "cpu_cost": cpu_cost,
    "memory_cost": memory_cost
})
```

3.2.3 Avantajele sistemului cu tick-uri

Predictibilitate completă: Știm exact când se va termina fiecare cerere și cum va evolua starea sistemului.

Control granular: Putem genera orice pattern de încărcare dorit prin simpla modificare a secvenței de cereri.

Scalabilitate temporală: Simularea poate rula la orice viteză, independent de puterea hardware.

Reproductibilitate perfectă: Aceeași secvență de cereri va genera întotdeauna aceleași rezultate.

3.3 Provocări tehnice și soluții adoptate

3.3.1 Sincronizarea stărilor

Problema: Asigurarea că toate serverele înregistrează starea sistemului la același moment de decizie.

Soluția: Implementarea protocolului de "tick sincron" în care agentul trimite cereri către toate serverele simultan, inclusiv cereri goale pentru sincronizare către serverele care nu primesc cererea reală.

3.3.2 Calculul încărcării eterogene

Problema: Crearea unor servere cu capacități diferite pentru a testa inteligența algoritmului de load balancing.

Soluția: Definirea capacităților diferite pentru fiecare server și calcularea procentajului de utilizare relativ la capacitatea individuală:

```
python
server_configs = {
    “server-1”: {“max_cpu”: 1500, “max_memory”: 2000},    # Server mic
    “server-2”: {“max_cpu”: 3500, “max_memory”: 3200},    # Server mediu
    “server-3”: {“max_cpu”: 5000, “max_memory”: 3000}     # Server mare
}


cpu_usage = (total_cpu_load / max_cpu) * 100
```

3.3.3 Persistența datelor

Problema: Salvarea rezultatelor analizei pentru procesare ulterioară și inspecție manuală.

Soluția: Utilizarea unui volum Docker partajat (`./models:/app/models`) între agentul și trainer-ul, cu generarea automată de fișiere JSON timestamp-ate pentru fiecare episod.

3.3.4 Handling-ul erorilor și resilența

Problema: Comportamentul sistemului în cazul eșecului temporar al unei componente.

Soluția: Implementarea de mecanisme de retry cu timeout-uri și fallback-uri:

```
python
try:
    response = requests.post(url, json=payload, timeout=10)
    if response.status_code == 200:
        # proces normal
    else:
        print(f”Error from {server}: {response.status_code}”)
except Exception as e:
    print(f”Failed to contact {server}: {e}”)
    # continuare fără să blocheze întregul sistem
```

4 DESIGNUL SISTEMULUI DE RECOMPENSE – LECȚII ÎNVĂȚATE

4.1 Prima abordare – limitările recompenselor bazate pe capacitate

Dezvoltarea unui sistem de recompense eficient pentru antrenarea agentului de Deep Reinforcement Learning s-a dovedit a fi una dintre provocările cele mai complexe și educative ale întregului proiect. Procesul de design m-a condus prin mai multe iterații, fiecare aducând învățăminte valoroase despre natura învățării automate în contextul load balancing-ului.

Abordarea inițială părea logică și intuitivă: să calculăm recompensele pe baza capacității disponibile a serverelor la momentul luării deciziei. Ideea de bază era să recompensăm agentul pentru alegerea serverelor cu cea mai mare capacitate liberă și să îl penalizăm pentru alegerea celor supraîncărcați.

Această metodă implica calcularea capacității disponibile pentru fiecare server ca diferența între capacitatea maximă (100%) și utilizarea curentă. Pentru un server cu 60% utilizare CPU și 55% utilizare memorie, am fi calculat capacitatea disponibilă ca fiind minimul dintre 40% CPU disponibil și 45% memorie disponibilă. Alegerea de a folosi minimul părea rezonabilă, considerând că serverul este limitat de resursa cea mai solicitată.

Implementarea acestei logici a fost relativ directă, iar primele teste păreau să funcționeze. Agentul primea recompense pozitive pentru alegerea serverelor cu capacitate mare și penalizări pentru alegerea celor congestionați.

4.2 Descoperirea problemei dependenței de încărcarea sistemului

Momentul crucial în evoluția proiectului a venit când am realizat o problemă fundamentală cu abordarea bazată pe capacitate: recompensele nu măsurau calitatea reală a deciziei, ci erau influențate semnificativ de nivelul general de încărcare al sistemului.

Pentru a ilustra această problemă, să considerăm două scenarii identice din perspectiva calității deciziei, dar diferite din perspectiva încărcării generale:

În primul scenariu, sistemul operează sub încărcare ușoară, cu serverele având utilizări de 10%, 20% și 15% respectiv. Serverul optimal avea 90% capacitate disponibilă, iar o decizia optimă primea o recompensă mare în valoare absolută.

În al doilea scenariu, același sistem opera sub încărcare intensă, cu utilizări de 70%, 80% și 75%. Serverul optimal avea doar 30% capacitate disponibilă, iar aceeași calitate de decizie optimă primea o recompensă mult mai mică în valoare absolută, deși calitatea relativă a deciziei era identică deoarece ne raportam la diferența dintre încărcătura curentă a serverelor și capacitatea lor maximă (de 100%).

Această observație m-a condus la realizarea că sistemul de recompense introduce un bias sistemic legat de încărcarea generală. Agentul ar fi învățat implicit că "deciziile bune în condiții de încărcare ușoară sunt mai valoroase decât deciziile bune în condiții de încărcare intensă", ceea ce reprezintă o distorsiune fundamentală a semnalului de învățare.

Mai mult, acest bias ar fi făcut antrenarea instabilă și imprevizibilă, deoarece aceeași calitate de decizie ar fi fost recompensată diferit în funcție de contextul extern al încărcării sistemului, un factor care nu ar trebui să influențeze evaluarea competenței agentului.

4.3 Dezvoltarea sistemului de recompense cu ideea de gradient în gând

Realizând limitările abordării bazate pe capacitate, am dezvoltat o metodologie complet nouă, pe care am numit-o "sistemul de recompense cu gradient". Această abordare se bazează pe o intuiție simplă dar puternică: calitatea unei decizii ar trebui măsurată prin comparația cu alternativele disponibile la acel moment, nu prin valori absolute.

Noul sistem operează pe principiul comparației directe între serverele disponibile. Pentru fiecare decizie, identificăm mai întâi serverul cu încărcarea totală cea mai mică (suma utilizării CPU și memorie), care devine alegerea "optimă" pentru acel tick. Această alegere se bazează pe premisa că, în general, serverul cu încărcarea cea mai mică este cel mai potrivit pentru primirea unei noi cereri.

Calculul recompenselor urmează două scenarii distincte. Când agentul face alegerea optimă, recompensa se calculează ca diferența dintre încărcarea medie a celorlalte servere și încărcarea serverului ales. Această formulă asigură că alegerea optimă primește întotdeauna o recompensă pozitivă, proporțională cu cât de mult mai bun este serverul ales față de alternativele disponibile.

Când agentul face o alegere suboptimală, recompensa devine diferența dintre încărcarea serverului optimal și încărcarea serverului ales efectiv. Această diferență este întotdeauna negativă, reprezentând o penalizare proporțională cu gravitatea "greșelii" comise.

Frumusețea acestei abordări constă în faptul că recompensele devin complet independente de nivelul general de încărcare al sistemului. O decizie optimă într-un sistem puternic încărcat va primi aceeași recompensă ca o decizie de calitate similară într-un sistem ușor încărcat, eliminând bias-ul sistemic identificat anterior.

4.4 Validarea prin testare

Implementarea noului sistem de recompense a fost urmată de o fază intensă de testare și validare. Am rulat multiple episoade cu algoritmul Round Robin ca baseline, analizând în detaliu distribuția recompenselor și calitatea deciziilor.

Rezultatele au confirmat eficiența abordării cu gradient. Într-un episod de test cu 20 de cereri, sistemul Round Robin a demonstrat comportamentul așteptat al unui algoritm care nu ia în considerare starea serverelor. Analiza detaliată a arătat că algoritmul a obținut un mix de decizii optime și suboptime, rezultând în recompense care variază de la +38.595 până la -28.0.

Distribuția recompenselor a validat funcționarea corectă a sistemului de gradient:

Pentru decizii optime: Când Round Robin a ales întâmplător serverul cu cea mai mică încărcare, sistemul a acordat recompense pozitive proporționale cu diferența față de alternativele disponibile. De exemplu, în tick-ul 3, alegerea server-ului 3 (0% încărcare) față de servere cu 30% și 14.82% încărcare a rezultat într-o recompensă de +22.41.

Pentru decizii suboptime: Când algoritmul a ales servere mai încărcate, penalizările au reflectat fidel gravitatea greșelii. În tick-ul 4, alegerea server-ului 1 (30% încărcare totală) când server-ul 3 avea doar 2% încărcare a rezultat într-o penalizare de -28.0.

Invarianța la încărcare: Cel mai important, sistemul a demonstrat că magnitudinea recompenselor depinde doar de diferențele relative între servere, nu de nivelul absolut de încărcare. Decizii similare în contexte de încărcare diferite au primit recompense comparabile, confirmând eliminarea bias-ului sistemic identificat în abordarea inițială.

Această distribuție clară a recompenselor oferă semnale de învățare ideale pentru antrenarea unui agent de Deep Reinforcement Learning: feedback pozitiv consistent pentru alegeri bune, penalizări proporționale pentru erori, și invarianță la contextul global al sistemului.

5 SISTEMUL DE ANTRENARE AUTOMAT

5.1 Coordonarea episoadelor

Una dintre realizările cele mai importante ale acestui semestru a fost dezvoltarea unui sistem complet automat pentru coordonarea episoadelor de antrenare. Această funcționalitate elimină nevoia de intervenție manuală în procesul de colectare și analiză a datelor, creând un pipeline fluid de la generarea traficului până la calculul recompenselor.

Inițial, m-am confruntat cu o provocare conceptuală importantă: cum să coordonăm sfârșitul unui episod astfel încât toate componentele să știe când să colecteze datele și să înceapă analiza? Prima abordare considerată implica ca generatorul de trafic k6 să semnaleze direct trainer-ul la sfârșitul fiecărui episod. Însă această metodă ridica probleme de sincronizare, deoarece k6 nu avea vizibilitate completă asupra momentului în care toate serverele terminau de procesat ultima cerere.

Soluția adoptată a fost elegantă în simplitatea ei: agentul de load balancing, fiind componenta centrală care orchestrează toate cererile, s-a dovedit a fi candidatul ideal pentru coordonarea episoadelor. Agentul contorizează intern fiecare cerere procesată și poate detecta automat când se atinge lungimea configurată a episodului.

Implementarea acestui mecanism a implicat extinderea agentului cu funcționalități de management al episoadelor. Variabila `current_episode_requests` se incrementează după fiecare cerere procesată, iar când atinge valoarea `episode_length` configurată, agentul declanșează automat secvența de finalizare a episodului. Această lungime poate fi ajustată dinamic prin endpoint-ul `/set_episode_length`, oferind flexibilitate în experimentare.

5.2 Colectarea și analiza datelor

Procesul de colectare a datelor la sfârșitul fiecărui episod reprezintă inima sistemului de antrenare automat. Trainer-ul implementează o strategie comprehensivă pentru agregarea informațiilor din toate sursele disponibile, reconstituind complet istoria deciziilor luate pe parcursul episodului.

Primul pas în acest proces este colectarea istoricului individual de la fiecare server prin endpoint-ul `/get_episode_history`. Fiecare server menține o înregistrare detaliată a tuturor tick-urilor procesate, incluzând starea sa înainte și după procesare, tipul cererii primite și metadatele asociate. Această abordare distribuită asigură că nicio informație valoroasă nu se pierde și că reconstituirea stării sistemului la orice moment dat este posibilă.

Următorul pas implică sincronizarea datelor din multiple surse într-o viziune coerentă a întregului episod. Trainer-ul construiește o mapare completă a tuturor tick-urilor, asociind pentru fiecare moment de decizie stările tuturor serverelor și identificând care server a primit cererea reală. Această informație este crucială pentru calculul corect al recompenselor, deoarece permite evaluarea calității fiecărei decizii în contextul complet al alternativelor disponibile.

Calculul recompenselor folosește sistemul de gradient dezvoltat și validat în capitolul anterior. Pentru fiecare tick, trainer-ul identifică serverul cu încărcarea minimă (considerată alegerea optimă) și compară această alegere cu decizia efectiv luată de agent. Rezultatul acestei comparații, ajustat conform formulelor descrise anterior, devine recompensa asociată acelei decizii.

Analiza finală include nu doar calculul recompenselor individuale, ci și o evaluare comprehensivă a performanței episodului:

- **Metrici de calitate:** numărul de decizii optime versus suboptime, distribuția recompenselor, variația performanței.
- **Analiza pe servere:** frecvența alegerii fiecărui server, recompensele medii asociate fiecărei alegeri.
- **Tendințe temporale:** evoluția calității deciziilor pe parcursul episodului, identificarea de pattern-uri.

5.3 Pipeline-ul de procesare automat

Integrarea tuturor acestor componente într-un pipeline fluid și automat a necesitat atenție deosebită la detaliile de sincronizare și la handling-ul erorilor. Secvența completă a procesării unui episod urmează un protocol strict pentru a asigura consistența datelor.

Când agentul detectează sfârșitul unui episod, acesta inițiază imediat comunicarea cu trainer-ul prin endpoint-ul `/start_training`. Această comunicare include metadate despre episodul completat, cum ar fi numărul total de cereri procesate și identificatorul ultimului tick. Trainer-ul folosește aceste informații pentru validarea consistenței datelor colectate.

Trainer-ul implementează mecanisme robuste de protecție împotriva procesării concurente prin utilizarea de lock-uri thread-safe. Aceasta previne situațiile în care multiple episoade ar putea să se suprapună din cauza unor întârzieri în rețea sau erori temporare. Variabila `training_in_progress` asigură că doar un singur episod este procesat la un moment dat.

Persistența rezultatelor se realizează prin generarea automată de fișiere JSON timestamp-ate în directorul partajat `/app/models`. Fiecare episod generează două

fișiere principale: unul conținând detaliile complete ale tuturor deciziilor și recompenselor (`episode_[timestamp]_rewards.json`) și unul cu o sinteză a metricilor de performanță (`episode_[timestamp]_summary.json`). Această structură permite atât analiza detaliată a comportamentului agentului, cât și tracking-ul evoluției performanței de-a lungul timpului.

După salvarea rezultatelor, trainer-ul returnează un răspuns detaliat către agent, confirmând succesul operațiunii și incluzând un rezumat al analizei efectuate. Agentul folosește această confirmare pentru a reseta contoarele interne și a pregăti sistemul pentru episodul următor, închizând astfel ciclul automat de procesare.

Această arhitectură automatizată elimină complet nevoia de intervenție manuală în procesul de antrenare, creând fundamentul pentru rularea de experimente extinse și pentru colectarea sistematică a datelor necesare antrenării algoritmilor de Deep Reinforcement Learning în etapele viitoare ale proiectului.

6 REZULTATE PRELIMINARE ȘI VALIDARE

6.1 Testarea infrastructurii

Validarea infrastructurii dezvoltate a constituit o etapă critică în demonstrarea viabilității conceptului LBNN. Această fază de testare a avut un dublu scop: pe de o parte, să verifice funcționarea corectă a tuturor componentelor integrate, iar pe de alta, să obțină o măsurătoare de bază (baseline) pentru evaluarea performanței viitorului agent de Deep Reinforcement Learning.

Primul set de teste a vizat validarea comunicării între componente și a sincronizării sistemului de tick-uri. Am verificat că fiecare server răspunde corect la cererile de procesare, că stările sunt calculate și raportate fidel, și că mecanismul de expirare automată a cererilor funcționează conform specificațiilor. Testele au demonstrat că sistemul menține consistența datelor chiar și sub condiții de încărcare variabilă, cu toate serverele raportând același tick la fiecare moment de decizie.

Testarea funcționalității de coordonare a episoadelor a necesitat validarea mai multor scenarii. Am verificat comportamentul sistemului pentru episoade de lungimi diferite (5, 10, 20 de cereri), asigurându-ne că agentul detectează corect sfârșitul fiecărui episod și că trainer-ul este activat automat. De asemenea, am testat robustețea sistemului în fața unor condiții de eroare, cum ar fi întârzieri în comunicarea cu serverele sau eșecuri temporare ale unei componente.

Un aspect crucial al testării a fost validarea sistemului de simulare a încărcării. Am verificat că serverele cu capacități diferite raportează procentaje de utilizare corecte pentru aceleași cereri, confirmând că heterogenitatea configurată se reflectă fidel în comportamentul sistemului. Server-ul cu capacitatea cea mai mică (server-1) prezenta procentaje de utilizare semnificativ mai mari decât serverele cu capacități superioare pentru aceleași tipuri de cereri.

6.2 Performanța baseline (Round Robin)

Implementarea algoritmului Round Robin ca strategie baseline s-a dovedit o decizie înțeleaptă pentru evaluarea ulterioară a performanței agentului de Deep Reinforcement Learning. Round Robin oferă o referință predictibilă și ușor de înțeles, distribuind cererile în mod ciclic între servere fără a lua în considerare starea actuală a acestora.

Rezultatele testării Round Robin au fost revelatorii în demonstrarea limitărilor abordărilor naive de load balancing. Într-un episod reprezentativ de 20 de cereri, analiza detaliată a performanței a evidențiat următoarele:

Distribuția calității deciziilor:

- Decizii optime: 8 din 20 (40%)
- Decizii suboptime: 12 din 20 (60%)
- Variație mare în calitatea deciziilor, cu recompense între -28.0 și +38.595

Această performanță demonstrează clar natura aleatorie a succesului Round Robin. Algoritmul obține decizii optime doar când ciclul său fix coincide întâmplător cu serverul cel mai puțin încărcat.

Pattern-uri problematice observate:

Analiza tick-by-tick a evidențiat problemele structurale ale acestei abordări. De exemplu, în secvența tick-urilor 1-4:

- **Tick 1:** Server-1 ales pentru cerere heavy (toate serverele goale - decizie neutră)
- **Tick 2:** Server-2 ales corect (0% vs 30% la server-1) - recompensă +30.0
- **Tick 3:** Server-3 ales corect (0% vs alte încărcări) - recompensă +22.41
- **Tick 4**: Server-1 ales greșit (30% încărcare când server-3 avea 2%) - penalizare -28.0

Acest pattern arată cum Round Robin poate face alegeri consecutive bune prin coincidență, apoi poate distruge această performanță prin revenirea mecanică la un server deja încărcat.

Impactul asupra serverelor:

Distribuția uniformă a cererilor între servere (aproximativ 6-7 cereri per server) ignoră complet diferențele de capacitate:

- Server-1 (1500 CPU): suprasolicitat frecvent.
- Server-2 (3500 CPU): utilizare moderată.
- Server-3 (5000 CPU): subutilizat.

Această distribuție ineficientă confirmă necesitatea unei abordări inteligente care să considere atât starea curentă, cât și capacitățile individuale ale serverelor.

Metrică	Valoare
Total cereri procesate	20
Decizii optime	8 (40%)
Decizii suboptime	12 (60%)

Recompensă medie	+4.28
Recompensă minimă	-28.0
Recompensă maximă	+38.595
Deviație standard	17.85
Distribuție pe servere	S1: 7, S2: 7, S3: 6
Tabel 2	

6.3 Analiza calității deciziilor

Examinarea în detaliu a fiecărei decizii din episoadele de test a oferit perspective valoroase asupra comportamentului sistemului și a eficacității sistemului de recompense dezvoltat. Această analiză granulară a demonstrat că noul sistem de calcul al recompenselor captează fidel nuanțele calității deciziilor în contexte diferite.

Analiza deciziilor optime:

Deciziile optime au primit consistent recompense pozitive proporționale cu avantajul față de alternative. Câteva exemple semnificative:

- **Tick 2:** Alegerea server-ului 2 (0% încărcare) când server-1 avea 30% încărcare a generat o recompensă de +30.0, reflectând diferența clară de încărcare.
- **Tick 6:** Alegerea server-ului 3 (0% încărcare) când alternativele aveau 60% și 17.19% încărcare a rezultat în cea mai mare recompensă din episod: +38.595.
- **Tick 9:** Chiar și pentru o cerere light, alegerea optimă a server-ului 3 (12.67% încărcare) față de servere cu 60% și 14.82% a generat +24.74 recompensă.

Analiza deciziilor suboptime:

Penalizările pentru decizii greșite au variat proporțional cu gravitatea erorii:

- **Erori minore:** În tick 11, alegerea server-ului 2 (14.82%) când server-3 avea 14.67% a rezultat doar în -0.15 penalizare, reflectând diferența minimă.
- **Erori moderate:** În tick 5, alegerea server-ului 2 (14.82%) când server-3 avea doar 2% a generat -12.82 penalizare.
- **Erori grave:** În tick 4, alegerea server-ului 1 (30%) când existau alternative mult mai bune a rezultat în -28.0 penalizare.

Validarea componentelor CPU și memorie:

Sistemul calculează corect contribuțiile separate ale CPU și memoriei la recompensa finală. Analiza a arătat că:

- Pentru majoritatea cererilor, componenta CPU domină recompensa (raport tipic 60-70% CPU vs 30-40% memorie).
- Acest lucru reflectă corect caracteristicile cererilor de test, care au raporturi CPU/memorie mai mari.
- Sistemul poate distinge între bottleneck-uri CPU vs memorie.

Confirmarea invarianței la încărcare:

Cel mai important, analiza a confirmat că sistemul de recompense este într-adevăr invariant la nivelul general de încărcare:

- Decizii similare în tick-uri cu încărcare sistemică diferită au primit recompense comparabile.
- Magnitudinea recompenselor depinde exclusiv de diferențele relative între servere.
- Nu există bias sistemic legat de momentul în care sunt luate deciziile.

6.4 Structura datelor și comenzi de control

6.4.1 Exemplu de date colectate pentru analiză

Pentru transparență completă asupra sistemului implementat, prezentăm primele 9 decizii dintr-un episod de test cu structura completă a datelor:


```
[
  {
    "tick_id": 1,
    "chosen_server": "server-1",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": 0.0,
    "cpu_reward": 0.0,
    "memory_reward": 0.0,
    "server_states": {
```

```
      "server-1": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 1},
      "server-2": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 1},
      "server-3": {"connections": 0, "cpu": 0, "memory": 0, "tick": 0}
    }
  },
  {
    "tick_id": 2,
    "chosen_server": "server-2",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": 30.0,
    "cpu_reward": 20.0,
    "memory_reward": 10.0,
    "server_states": {
      "server-1": {"connections": 1, "cpu": 20.0, "memory": 10.0, "tick": 2},
      "server-2": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 2},
      "server-3": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 1}
    }
  },
  {
    "tick_id": 3,
    "chosen_server": "server-3",
    "request": {
      "cpu_cost": 50,
      "duration": 2,
      "memory_cost": 30,
      "type": "light"
    },
    "final_reward": 22.41,
    "cpu_reward": 14.285,
    "memory_reward": 8.125,
    "server_states": {
```

```
      "server-1": {"connections": 1, "cpu": 20.0, "memory": 10.0, "tick": 3},
      "server-2": {"connections": 1, "cpu": 8.57, "memory": 6.25, "tick": 3},
      "server-3": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 2}
    }
  },
  {
    "tick_id": 4,
    "chosen_server": "server-1",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": -28.0,
    "cpu_reward": -19.0,
    "memory_reward": -9.0,
    "server_states": {
      "server-1": {"connections": 1, "cpu": 20.0, "memory": 10.0, "tick": 4},
      "server-2": {"connections": 1, "cpu": 8.57, "memory": 6.25, "tick": 4},
      "server-3": {"connections": 1, "cpu": 1.0, "memory": 1.0, "tick": 3}
    }
  },
  {
    "tick_id": 5,
    "chosen_server": "server-2",
    "request": {
      "cpu_cost": 50,
      "duration": 2,
      "memory_cost": 30,
      "type": "light"
    },
    "final_reward": -12.82,
    "cpu_reward": -7.57,
    "memory_reward": -5.25,
    "server_states": {
```

```
      "server-1": {"connections": 2, "cpu": 40.0, "memory": 20.0, "tick": 5},
      "server-2": {"connections": 1, "cpu": 8.57, "memory": 6.25, "tick": 5},
      "server-3": {"connections": 1, "cpu": 1.0, "memory": 1.0, "tick": 4}
    }
  },
  {
    "tick_id": 6,
    "chosen_server": "server-3",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": 38.595,
    "cpu_reward": 25.0,
    "memory_reward": 13.595,
    "server_states": {
      "server-1": {"connections": 2, "cpu": 40.0, "memory": 20.0, "tick": 6},
      "server-2": {"connections": 2, "cpu": 10.0, "memory": 7.19, "tick": 6},
      "server-3": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 5}
    }
  },
  {
    "tick_id": 7,
    "chosen_server": "server-1",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": -17.33,
    "cpu_reward": -14.0,
    "memory_reward": -3.33,
    "server_states": {
```

```
      "server-1": {"connections": 1, "cpu": 20.0, "memory": 10.0, "tick": 7},
      "server-2": {"connections": 1, "cpu": 8.57, "memory": 6.25, "tick": 7},
      "server-3": {"connections": 1, "cpu": 6.0, "memory": 6.67, "tick": 6}
    }
  },
  {
    "tick_id": 8,
    "chosen_server": "server-2",
    "request": {
      "cpu_cost": 300,
      "duration": 6,
      "memory_cost": 200,
      "type": "heavy"
    },
    "final_reward": 36.335,
    "cpu_reward": 23.0,
    "memory_reward": 13.335,
    "server_states": {
      "server-1": {"connections": 2, "cpu": 40.0, "memory": 20.0, "tick": 8},
      "server-2": {"connections": 0, "cpu": 0.0, "memory": 0.0, "tick": 8},
      "server-3": {"connections": 1, "cpu": 6.0, "memory": 6.67, "tick": 7}
    }
  },
  {
    "tick_id": 9,
    "chosen_server": "server-3",
    "request": {
      "cpu_cost": 50,
      "duration": 2,
      "memory_cost": 30,
      "type": "light"
    },
    "final_reward": 24.740000000000002,
    "cpu_reward": 18.285,
    "memory_reward": 6.455,
    "server_states": {
```

```
      "server-1": {"connections": 2, "cpu": 40.0, "memory": 20.0, "tick": 9},
      "server-2": {"connections": 1, "cpu": 8.57, "memory": 6.25, "tick": 9},
      "server-3": {"connections": 1, "cpu": 6.0, "memory": 6.67, "tick": 8}
    }
  }
]
```

Această structură captează pentru fiecare decizie:

- Identificatorul tick-ului și serverul ales
- Detaliile complete ale cererii procesate
- Recompensele calculate (totală, CPU, memorie)
- Starea completă a tuturor serverelor după procesare

6.4.2 Comenzi de control și monitorizare

Sistemul expune multiple endpoint-uri pentru control și observabilitate:

Configurarea lungimii episoadelor:

```
bash
POST /set_episode_length
{
  "episode_length": 20
}

Response:
{
  "status": "updated",
  "new_episode_length": 20
}
```

Verificarea progresului episodului curent:

```
bash
GET /episode_status

Response:
{
```

```
  "current_requests": 12,
  "episode_length": 20,
  "progress_percentage": 60.0,
  "current_tick": 12
}
```


Resetarea manuală a serverelor (pentru debugging):

```
bash
GET /reset_episode

Response:
```


```
{
  "status": "episode_reset",
  "server_resets": {
    "server-1": "success",
    "server-2": "success",
    "server-3": "success"
  }
}
```

7 PROVOCĂRI ÎNTÂMPINATE ȘI SOLUȚII

7.1 Sincronizarea componentelor

Una dintre provocările tehnice cele mai complexe a fost asigurarea sincronizării perfecte între toate componentele sistemului distribuit. Natura asincronă a comunicării prin HTTP între containere Docker a creat multiple oportunități pentru apariția de inconsistențe în starea globală a sistemului.

Problema inițială s-a manifestat prin situații în care serverele raportau timpi diferite pentru același moment de decizie, rezultând în imposibilitatea reconstituirii fidele a contextului în care au fost luate deciziile. Această problemă era amplificată de latența variabilă a rețelei Docker și de diferențele în timpul de procesare între servere.

Prima abordare încercată implica transmiterea de mesaje de sincronizare separate, dar această metodă s-a dovedit ineficientă și predispusă la erori. Soluția finală adoptată a fost integrarea informațiilor de sincronizare direct în protocolul de comunicare existent. Fiecare cerere trimisă de agent către servere include un `tick_id` unic și secvențial, iar serverele sunt obligate să proceseze cererile în ordinea acestor identificatori.

Aspectul cel mai ingenios al soluției a fost decizia de a trimite cereri către toate serverele la fiecare decizie, nu doar către serverul ales. Serverele care nu primesc cererea reală procesează o cerere goală care servește exclusiv sincronizării. Această abordare garantează că toate componentele înregistrează același moment de decizie și că starea globală a sistemului poate fi reconstituită cu precizie perfectă.

Implementarea acestei soluții a necesitat modificări substanțiale în logica de procesare a tuturor serverelor, dar rezultatul a fost un sistem complet deterministic. Testele ulterioare au demonstrat că diferențele de timing între componente nu afectează consistența datelor, iar sistemul funcționează identic indiferent de variațiile în latența rețelei.

7.2 Designul funcției de recompensă

Evoluția sistemului de recompense de-a lungul proiectului ilustrează perfect natura iterativă a cercetării în Machine Learning și importanța testării riguroase a ipotezelor teoretice prin implementare practică.

Sistemul inițial bazat pe capacitatea disponibilă părea teoretic solid și a fost implementat cu încredere că va oferi semnalele corecte pentru învățare. Primele teste au confirmat funcționarea tehnică a implementării, iar recompensele calculate

păreau să reflecte intuițiile despre calitatea deciziilor. Însă aprofundarea analizei a relevat un defect fundamental care ar fi putut compromite întregul proces de antrenare.

Momentul revelator a fost când am simulat scenarii cu încărcări diferite ale sistemului și am observat că aceeași calitate relativă de decizie primea recompense foarte diferite în funcție de contextul global. Un agent antrenat cu un astfel de sistem ar fi dezvoltat preferințe irațională legate de nivelul de încărcare al sistemului, învățând implicit că anumite momente sunt “mai bune” pentru luarea de decizii decât altele.

Procesul de reproiectare a necesitat o reconceptualizare fundamentală a ceea ce înseamnă “calitatea unei decizii” în contextul load balancing-ului. Am realizat că o decizie ar trebui evaluată exclusiv prin prisma alternativelor disponibile la acel moment, nu prin comparație cu standarde absolute. Această intuiție a condus la dezvoltarea sistemului de gradient, care măsoară performanța relativă în loc de performanța absolută.

Implementarea noului sistem a fost mai complexă din punct de vedere computațional, necesitând calcularea relativă a încărcărilor pentru toate serverele și identificarea alegerii optime pentru fiecare tick. Însă această complexitate suplimentară a fost justificată de stabilitatea și predictibilitatea semnalelor de învățare rezultate.

Lecții învățate din această evoluție:

1. **Importanța gândirii relative:** În sisteme distribuite, performanța absolută este mai puțin relevantă decât performanța relativă față de alternative.
2. **Validarea prin implementare:** Doar prin testarea practică am putut identifica problema subtilă dar critică a bias-ului de încărcare.
3. **Simplicitatea conceptuală vs complexitatea implementării:** Sistemul de gradient, deși mai complex de implementat, oferă o semantică mai clară și mai intuitivă.
4. **Robustețea semnalelor de învățare:** Un sistem de recompense bun trebuie să ofere feedback consistent indiferent de contextul extern.

Rezultatele finale au confirmat superioritatea abordării cu gradient. Testele cu Round Robin au demonstrat că sistemul discriminează corect între decizii de calități diferite, oferind recompense pozitive pentru alegeri optime (până la +38.595) și penalizări proporționale pentru erori (până la -28.0), toate fiind invariante la nivelul general de încărcare al sistemului.

7.3 Coordonarea episoadelor de antrenare

Dezvoltarea unui mecanism robust pentru coordonarea automată a episoadelor a trecut prin mai multe iterații înainte de atingerea soluției finale elegante și eficiente.

Prima abordare considerată plasa responsabilitatea coordonării la nivelul generatorului de trafic k6. Ideea era ca k6 să notifice direct trainer-ul la sfârșitul fiecărui batch de cereri, declanșând procesul de analiză. Această abordare părea logică din perspectiva separării responsabilităților, dar a ridicat rapid probleme practice de implementare.

Principala dificultate era că k6 nu avea vizibilitate asupra momentului real de finalizare a procesării cererii. Deși k6 primea confirmarea că cererea a fost acceptată de agent, procesarea efectivă și actualizarea stărilor serverelor continuau asincron. Încercările de a implementa mecanisme de așteptare în k6 au complicat semnificativ logica de generare a traficului și au introdus dependențe tight-coupled între componente.

A doua abordare a mutat responsabilitatea la nivelul trainer-ului, care monitoriza periodic starea serverelor pentru a detecta finalizarea episoadelor. Această metodă elimina dependențele directe între k6 și trainer, dar introducea complexitate în detectarea momentului precis de sfârșit al episodului și risca să rateze cereri din cauza timing-ului incorect al polling-ului.

Soluția finală a emersat din observația că agentul este componenta care are cea mai completă vizibilitate asupra stării sistemului. Agentul procesează fiecare cerere, comunică cu toate serverele și primește confirmări despre finalizarea procesării. Această poziție centrală îl face candidatul ideal pentru detectarea momentului exact de sfârșit al episodului.

Implementarea acestei soluții a fost surprinzător de simplă și elegantă. Agentul menține un contor intern al cererilor procesate și compară această valoare cu lungimea configurată a episodului după fiecare operațiune. Când pragul este atins, agentul inițiază imediat secvența de finalizare, garantând că toate datele necesare sunt disponibile și că nu există întârzieri în procesare.

Această abordare a eliminat complet race condition-urile legate de timp și a creat un sistem complet deterministic pentru managementul episoadelor. Testele extensive au demonstrat că sistemul funcționează consistent indiferent de variațiile în încărcarea serverelor sau de întârzierile ocazionale în comunicarea prin rețea.

8 CONTRIBUȚII ȘI INOVAȚII

8.1 Sistemul de recompense invariant la încărcare

Una dintre contribuțiile cele mai semnificative ale acestui proiect este dezvoltarea unui sistem de recompense care elimină bias-ul legat de încărcarea globală a sistemului, o problemă care nu a fost identificată în mod explicit în literatura de specialitate consultată pentru primul semestru.

Majoritatea implementărilor de Deep Reinforcement Learning pentru load balancing din literatură se concentrează pe optimizarea metricilor absolute, cum ar fi utilizarea resurselor sau timpii de răspuns. Aceste abordări, deși intuitive, introduc subtile dependențe între calitatea percepută a deciziilor și contextul operațional global. Sistemul de recompense cu gradient rezolvă această problemă prin focalizarea exclusivă pe performanța relativă în cadrul setului de alternative disponibile.

Principiul fundamental al inovației constă în recunoașterea că o decizie optimă într-un sistem puternic încărcat merită aceeași recompensă ca o decizie de calitate similară într-un sistem ușor încărcat. Această invarianță la încărcare creează un mediu de învățare stabil, unde agentul nu dezvoltă preferințe irațională legate de momentul în care sunt luate deciziile.

Implementarea practică a acestui concept a necesitat dezvoltarea unei metodologii de comparație dinamică între servere. Pentru fiecare decizie, sistemul identifică alegerea optimă prin calcularea încărcării totale pentru toate serverele disponibile, apoi calculează recompensa ca diferența între performanța alegerii efective și performanța alternativelor. Această abordare asigură că magnitudinea recompenselor reflectă fidel calitatea relativă a deciziilor, independent de contextul absolut.

Validarea acestei inovații prin testarea cu Round Robin a demonstrat capacitatea sistemului de a discrimina corect între decizii de calități diferite. Recompensele pozitive pentru alegeri optime și penalizările proporționale pentru alegeri suboptime creează un gradient clar de învățare, esențial pentru convergența eficientă a algoritmilor de Deep Reinforcement Learning.

8.2 Arhitectura de sincronizare bazată pe tick-uri

O altă contribuție majoră a proiectului este dezvoltarea unei arhitecturi de simulare bazată pe tick-uri discrete, care elimină dependențele de timpul real și creează un mediu de antrenare complet reproductibil pentru algoritmii de load balancing.

Abordările tradiționale de simulare în acest domeniu se bazează adesea pe timing real sau pe simularea aproximativă a comportamentului sistemelor. Aceste metode introduc variabilitate nedorită în procesul de antrenare și fac dificilă reproducerea exactă a experimentelor. Sistemul cu tick-uri elimină aceste probleme prin discretizarea completă a timpului și a stărilor sistemului.

Inovația cheie constă în înlocuirea conceptului de durată în timp real cu durata în tick-uri. Fiecare cerere specifică numărul de tick-uri pentru care va consuma resurse, iar sistemul avansează deterministic în funcție de aceste specificații. Această abordare permite simularea rapidă a unor scenarii complexe fără dependențe de performanța hardware sau de variațiile în latența rețelei.

Sincronizarea perfectă între toate componentele sistemului este asigurată prin trimiterea de cereri către toate serverele la fiecare tick, incluzând cereri goale pentru serverele care nu primesc sarcina reală. Această strategie garantează că toate componentele înregistrează aceeași stare pentru același moment de decizie, asigurând consistența datelor.

Implementarea acestei arhitecturi a necesitat reconsiderarea fundamentală a modului în care sunt simulate resursele serverelor. În loc să depindem de metrici hardware reale, am dezvoltat un sistem de contabilizare artificial care calculează utilizarea bazată pe cererile active și pe durata acestora. Această abordare oferă control granular asupra pattern-urilor de încărcare și permite generarea de scenarii de test reproductibile.

8.3 Pipeline-ul automat de antrenare

Dezvoltarea unui pipeline complet automatizat pentru coordonarea episoadelor de antrenare reprezintă o contribuție practică semnificativă care facilitează experimentarea intensivă și reduce barierele din calea cercetării iterative.

Arhitectura pipeline-ului se bazează pe principiul responsabilității distribuite, unde fiecare componentă are roluri clare și bine delimitate. Agentul detectează automat sfârșitul episoadelor și declanșează procesul de analiză, trainer-ul colectează și procesează datele din toate sursele, iar rezultatele sunt salvate automat pentru analiză ulterioară. Această separare a responsabilităților asigură modularitatea și facilitează extinderea sistemului.

Aspectul cel mai valoros al automatizării este eliminarea erorilor umane din procesul de colectare a datelor. Sistemul manual ar fi fost predispus la inconsistențe temporale, la omiterea unor date sau la erori în calculul recompenselor. Pipeline-ul automat asigură că fiecare decizie este înregistrată fidel și că analiza este efectuată consistent conform metodologiei definite.

Flexibilitatea sistemului este asigurată prin parametrizarea lungimii episoadelor și prin expunerea de endpoint-uri pentru configurarea dinamică. Această caracteristică permite experimentarea rapidă cu diferite setup-uri fără necesitatea modificărilor de cod, accelerând semnificativ ciclul de dezvoltare și testare.

Impactul acestei automatizări se va manifesta complet în fazele următoare ale proiectului, când antrenarea intensivă a modelelor de Deep Reinforcement Learning va necesita rularea a sute sau mii de episoade. Eliminarea intervenției manuale din acest proces va permite focalizarea resurselor pe optimizarea algoritmilor și pe analiza rezultatelor, accelerând progresul cercetării.

Toate aceste contribuții și inovații creează împreună o platformă solidă și bine fundamentată pentru implementarea și evaluarea algoritmilor de Deep Reinforcement Learning în contextul load balancing-ului pentru aplicații web. Fiecare component adresează provocări specifice identificate în procesul de dezvoltare și oferă soluții care facilitează cercetarea viitoare în acest domeniu.

9 ETAPELE URMĂTOARE

9.1 Implementarea modelului de Deep Reinforcement Learning

Dezvoltarea mediului Gymnasium:

- Crearea unei interfețe standardizate pentru integrarea cu Stable Baselines3.
- Definirea spațiului de stări: metrici servere (CPU, memorie, conexiuni) + caracteristici cerere.
- Configurarea spațiului de acțiuni: alegere discretă între cele 3 servere (0, 1, 2).
- Integrarea sistemului de recompense cu gradient dezvoltat și validat.

Selectarea și configurarea algoritmului:

- Testarea algoritmilor PPO și DQN pentru determinarea celui mai eficient.
- Optimizarea hiperparametrilor: learning rate, batch size, exploration strategy.
- Configurarea arhitecturii rețelei neuronale pentru procesarea datelor de stare.
- Implementarea de mecanisme pentru salvarea și încărcarea modelelor antrenate.

9.2 Antrenarea și evaluarea performanței

Procesul de antrenare:

- Rularea de episoade intensive (1000 – 10000) pentru convergența modelului.
- Monitorizarea metricilor de antrenare: recompensă medie, rata de convergență.
- Implementarea de early stopping pentru evitarea overfitting-ului.
- Validarea periodică pe episoade de test separate.

Evaluarea comparativă:

- Măsurarea performanței față de baseline-ul Round Robin (ținta: > 80% decizii optime vs 40% actual).
- Implementarea algoritmului Least Connections pentru o comparație suplimentară.
- Analiza comportamentului în scenarii variate: trafic uniform, vârfuri de încărcare, distribuții diferite de cereri.
- Măsurarea stabilității deciziilor și a capacității de generalizare.

9.3 Comparația cu algoritmi tradiționali

Algoritmi de referință:

- **Round Robin:** distribuție ciclică.
- **Least Connections:** alegere bazată pe numărul de conexiuni active.
- **Weighted Round Robin:** distribuție bazată pe capacitățile serverelor.
- **Random:** alegere aleatoare pentru validarea inferiorității extreme.

Metrici de evaluare:

- **Rata deciziilor optime:** procentul alegărilor care coincid cu serverul cu încărcarea minimă.
- **Recompensa medie pe episod:** folosind sistemul de gradient dezvoltat.
- **Distribuția încărcării:** măsurarea echilibrului utilizării între servere.
- **Stabilitatea performanței:** variația calității deciziilor între episoade diferite.

10 10. CONCLUZII

10.1 Obiective realizate în semestrul II

Infrastructura completă și funcțională:

- Sistem distribuit bazat pe Docker cu 5 componente integrate fără fricțiune.
- Arhitectură de simulare bazată pe tick-uri, reproductibilă și deterministică.
- Pipeline automat de antrenare care elimină complet intervenția manuală.
- Mecanisme robuste de sincronizare și handling al erorilor.

Sistemul de recompense:

- Dezvoltarea conceptului de recompense invariante la încărcarea sistemului.
- Validarea prin testare a superiorității față de abordările tradiționale bazate pe capacitate.
- Implementarea calculului de gradient care oferă semnale clare pentru învățarea RL.
- Sistem funcțional care discriminează corect între decizii optime (+38.595 max) și suboptime (-28.0 min).

Rezultate de bază solide:

- Măsurarea performanței Round Robin: 40% decizii optime din 20 de cereri testate.
- Demonstrarea variabilității mari în calitatea deciziilor (recompense între -28.0 și +38.595).
- Identificarea clară a spațiului pentru îmbunătățire prin abordări inteligente.
- Validarea că sistemul discriminează corect între decizii de calități diferite.

Infrastructura de testare și validare:

- Pipeline complet automatizat pentru rularea episoadelor și colectarea datelor.
- Sistem de analiză care calculează automat metricile de performanță.
- Capacitatea de a rula experimente reproductibile cu diferite configurații.
- Generare automată de rapoarte detaliate pentru fiecare episod.

Pregătirea pentru faza de Deep Learning:

- Toate componentele necesare pentru implementarea agentului DRL sunt funcționale.
- Sistemul de recompense oferă semnale de învățare optime și stabile.

- Baseline-ul Round Robin stabilește o țintă clară de îmbunătățire (de la 40% la > 80% decizii optime).
- Infrastructura permite experimentarea rapidă cu diferiți algoritmi și hiperparametri.

10.2 Învățăminte din procesul de implementare

Importanța testării iterative:

- Prima abordare la funcția de recompensă părea corectă teoretic, dar avea defecte fundamentale.
- Implementarea și testarea practică au evidențiat probleme ascunse în design.
- Ciclul build-measure-learn s-a dovedit esențial pentru atingerea unei soluții robuste.

Valoarea automatizării:

- Eliminarea intervenției manuale a accelerat ciclul de experimentare.
- Reducerea erorilor umane a îmbunătățit calitatea și consistența datelor colectate.
- Investiția în automatizare se va amortiza exponențial în fazele de antrenare intensivă.

Sincronizarea în sisteme distribuite:

- Coordonarea componentelor asincrone necesită design atent și mecanisme explicite
- Soluțiile simple și elegante sunt adesea superioare celor complexe.
- Testarea în condiții variate este curială pentru validarea robusteții.

10.3 Fundamentul pentru continuarea lucrării

Baza tehnică solidă:

- Infrastructura dezvoltată este pregătită pentru implementarea oricărui algoritm de RL.
- Sistemul de recompense oferă semnale de învățare optime pentru convergența modelelor.
- Pipeline-ul automat permite experimentarea rapidă cu diferite abordări și configurații.

Contribuții originale validate:

- Sistemul de recompense cu gradient elimină bias-urile comune.
- Arhitectura bazată pe tick-uri oferă reproductibilitate perfectă pentru cercetare.
- Designul modular facilitează extinderea și adaptarea pentru scenarii noi.

Perspectiva clară pentru semestrul următor:

- Obiective concrete și măsurabile: îmbunătățirea de la 40% la > 80% decizii optime.
- Metodologie bine definită pentru antrenare, evaluare și comparație.
- Instrumentele necesare pentru demonstrarea superiorității abordărilor DRL sunt pregătite.