Raport: Implementare Q-Learning

-utilizând environment Flappy Bird-

Echipa: Hurmuzache Maria-Ioana, Nistor Cristiana-Elena

1. Introducere

Acest proiect implementează un agent de Reinforcement Learning capabil să joace jocul *Flappy Bird* folosind exclusiv informația dată de pixelii de pe ecran. Soluția utilizează o arhitectură de tip **Dueling Double Deep Q-Network**, implementată în Python cu biblioteca PyTorch și mediul de simulare flappy-bird-gymnasium.

Obiectivul agentului este maximizarea scorului (numărul de țevi depășite) prin învățarea unei politici optime de acțiune (Sari / Nu face nimic) pe baza stărilor vizuale consecutive.

2. Procesarea Datelor de Intrare

Datele de intrare constau în seturi de 4 frame-uri consecutive preprocesate după cum urmează:

1. **Captură:** Se preia frame-ul brut RGB din joc.
2. **Conversie Grayscale:** Se elimină informația de culoare, reducând dimensiunea datelor de 3 ori prin eliminarea canalelor corespunzătoare culorilor din RGB.
3. **Decupare:** Se elimină zona de sus (scorul) și zona de jos (pământul), concentrând atenția agentului strict pe obstacole.
4. **Redimensionare:** Imaginea este redusă la dimensiunea **84x84 pixeli** folosind interpolare biliniară, format standard pentru rețelele convoluționale (inspirat din lucrarea DeepMind Atari).
5. **Normalizare:** Valorile pixelilor [0, 255] sunt împărțite la 255.0 pentru a obține valori float între [0, 1].

Acest lucru permite rețelei să deducă *viteza* și *direcția* de deplasare a păsării pe lângă poziționarea elementelor din imagini (pasăre și țevi).

3. Arhitectura Rețelei Neuronale

Agentul utilizează o arhitectură **Dueling DQN**, care îmbunătățește stabilitatea învățării față de DQN-ul clasic prin separarea estimării valorii stării de necesitatea acțiunii.

3.1. Blocul Convoluțional

Acesta are rolul de a extrage trăsături vizuale din stiva de 4 imagini (4 x 84 x 84).

- **Conv2d 1:** 32 filtre, kernel 8x8, stride 4. (Detectează trăsături brute).
    - Dimensiune Rezultat: 32x20x20

- **Conv2d 2:** 64 filtre, kernel 4x4, stride 2. (Detectează forme mai complexe).
    - Dimensiune Rezultat: 64x9x9
- **Conv2d 3:** 64 filtre, kernel 3x3, stride 1. (Detectează detalii fine).
    - Dimensiune Rezultat: 64x7x7
- **Flatten:** Transformă hărțile de trăsături 3D într-un vector 1D de dimensiune **3136**.

Formula utilizată la calculul dimensiunilor:

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

3.2. Fluxurile Dueling

După extragerea trăsăturilor, rețeaua se bifurcă:

1. **Value Stream V(s):** Estimează cât de bună este starea curentă în sine (ex: "Sunt într-o poziție sigură?").
    - Structura:
        i. Linear(3136 -> 512)
        ii. ReLU() (funcție de activare)
        iii. Linear(512 -> 1).
2. **Advantage Stream A(s, a):** Estimează cât de bună este fiecare acțiune comparativ cu celelalte.
    - Structura:
        i. Linear(3136 -> 512)
        ii. ReLU() (funcție de activare)
        iii. Linear(512 -> 2) (Sari/Nimic).

3.3. Agregarea

Cele două fluxuri sunt combinate în stratul final folosind formula care asigură stabilitatea (scăzând media avantajelor):

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a') \right)$$

4. Implementarea Algoritmului Q-Learning

Algoritmul folosit este **Double DQN**, care rezolvă problema supraestimării valorilor Q din DQN-ul simplu.

4.1. Componente Cheie

- **Replay Memory (Experience Replay):** Un buffer (deque de mărime 15.000) care stochează tranziții de forma: (stare, acțiune, recompensă, stare_următoare, done). Antrenarea se face pe un *batch* aleatoriu (64 mostre), rupând corelația temporală dintre frame-uri și stabilizând gradienții.
- **Două Rețele (Policy & Target):**
  - policy_network: Alege acțiunile și se antrenează la fiecare pas.
  - target_network: O copie "înghețată" a rețelei principale, folosită pentru calculul țintei. Se actualizează la fiecare 1000 de pași (load_state_dict).
- **Double DQN Logic:**
  - **Selecția:** policy_network decide *care* este cea mai bună acțiune viitoare (argmax).
  - **Evaluarea:** target_network calculează *valoarea* acelei acțiuni.
  - Formula Țintei:

$$Y = R + \gamma \cdot Q_{target}(s', argmax\, Q_{policy}(s', a))$$

4.2. Funcția de Loss și Optimizare

Antrenarea rețelei neuronale se bazează pe minimizarea erorii dintre ceea ce agentul *crede* că se va întâmpla (Predicția) și ceea ce *calculează* că se va întâmpla pe baza ecuației Bellman (Ținta).

4.2.1. Funcția de Pierdere: SmoothL1Loss (Huber Loss)

În acest proiect, am utilizat funcția nn.SmoothL1Loss, cunoscută și ca **Huber Loss**. Aceasta măsoară discrepanța dintre current_q_values (predicția rețelei policy) și target_q_values (ținta calculată cu rețeaua target).

Formula Matematică:

Fie x diferența dintre țintă și predicție: $x = y_{target} - y_{pred}$. Funcția de pierdere este definită astfel:

$$L(x) = \begin{cases} 0.5(x-y)^2, & \text{if } |x-y| < 1 \\ |x-y| - 0.5, & \text{otherwise} \end{cases}$$

Justificarea alegerii în Reinforcement Learning:

În jocurile de tip Atari sau Flappy Bird, datele de antrenament sunt instabile. Recompensele pot apărea brusc, generând erori mari ("outliers").

1. **Protecție împotriva "Exploding Gradients" (Cazul |x| > 1):**
   - Dacă am folosi MSE (Mean Squared Error - x^2), o eroare mare ar rezulta într-un gradient imens. Acest lucru ar modifica greutățile rețelei prea drastic, destabilizând tot ce a învățat anterior.

    - SmoothL1Loss se comportă **liniar** pentru erori mari (derivata este constantă). Aceasta funcționează ca o formă automată de *Gradient Clipping*, asigurând o învățare stabilă chiar și când agentul face greșeli majore.
2. **Convergență Fină (Cazul |x| < 1):**
    - Dacă am folosi MAE (Mean Absolute Error - |x|), gradientul ar fi constant chiar și când suntem foarte aproape de 0. Acest lucru face ca rețeaua să "oscileze" în jurul soluției optime fără să se stabilizeze.
    - SmoothL1Loss se comportă **pătratic** pentru erori mici, micșorând pașii de învățare pe măsură ce eroarea scade spre zero, permițând o convergență precisă.

4.2.2. Optimizatorul: Adam (Adaptive Moment Estimation)

Pentru actualizarea greutăților rețelei pe baza gradienților calculați de funcția de loss, am utilizat algoritmul **Adam**.

Mecanismul de funcționare:

Adam nu folosește o rată de învățare fixă pentru toți parametrii, ci o adaptează individual pentru fiecare neuron (greutate) din rețea, combinând două idei:

1. **Momentum:** Păstrează o medie a gradienților anteriori. Dacă gradientul a tot indicat "sari" în ultimii pași, Adam va continua să împingă "sari" cu viteză, ajutând algoritmul să treacă peste minimele locale și să navigheze zonele plate ale funcției de loss.
2. **RMSProp (Scalarea):** Împarte rata de învățare la o medie a pătratelor gradienților recenți. Aceasta înseamnă că parametrii care se schimbă rar vor primi actualizări mai mari, iar cei care fluctuează des vor primi actualizări mai mici (stabilizare).

Hiperparametrul Learning Rate:

Am ales o rată de învățare de 0.0001.

- O valoare mai mare ar fi făcut ca rețeaua să uite prea repede experiențele vechi ("Catastrophic Forgetting") în favoarea celor noi.
- O valoare mai mică ar fi necesitat un timp de antrenare prohibitiv de lung.
- Valoarea aleasă reprezintă compromisul ideal între viteză și stabilitate pentru arhitectura Dueling DQN.

4.2.3. Fluxul de Backpropagation

Procesul de optimizare la fiecare pas de antrenare urmează secvența standard PyTorch:

1. **optimizer.zero_grad()**: Se șterg gradienții acumulați anterior pentru a nu influența pasul curent.
2. **loss.backward()**: Se calculează gradientul erorii în raport cu fiecare parametru al rețelei (regula lanțului), propagând eroarea de la ieșire spre intrare.
3. **optimizer.step()**: Adam actualizează greutățile folosind gradienții calculați și stările interne (momentum).

4.3. Strategia de Explorare

Se folosește o strategie **Epsilon-Greedy** cu descreștere exponențială:

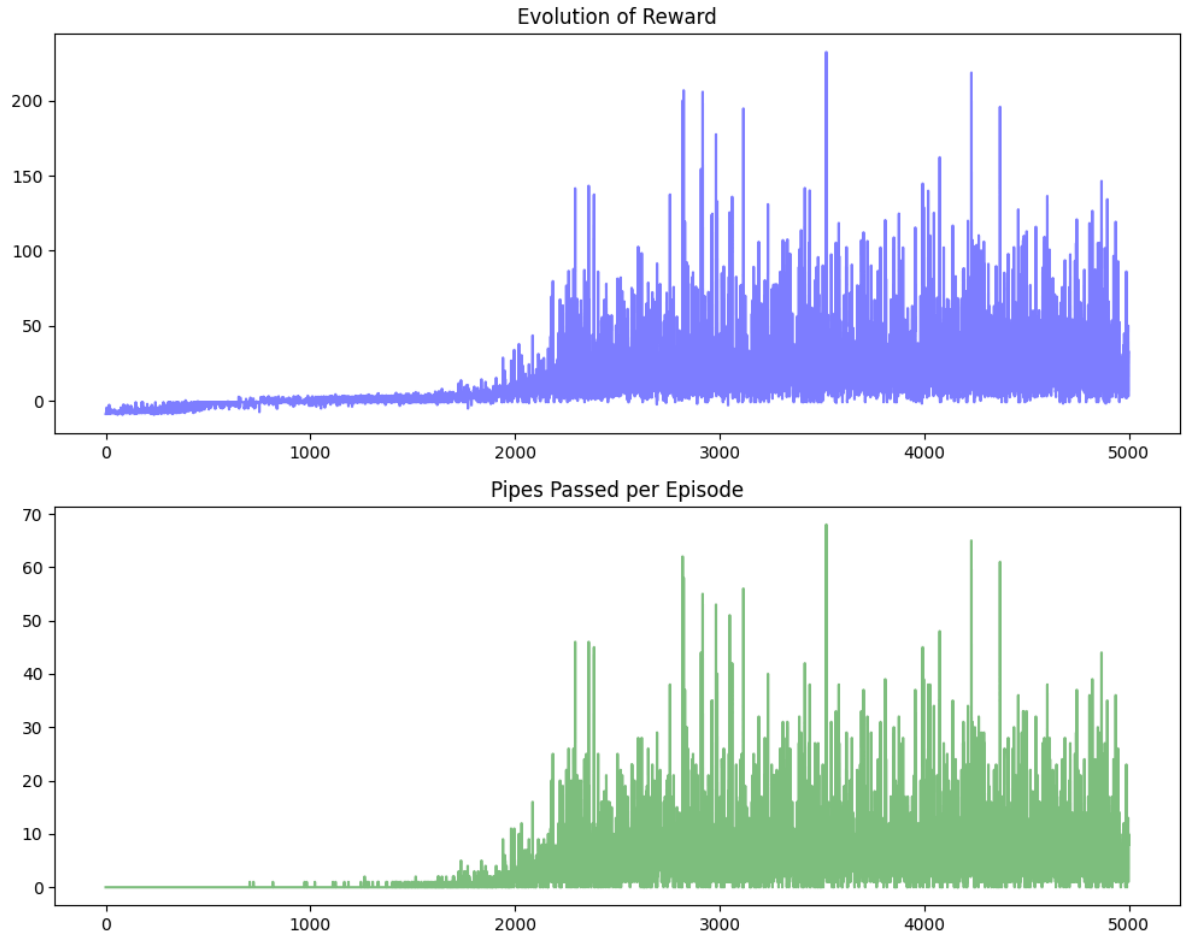
- La început, epsilon = 1.0 (Agentul acționează complet aleatoriu pentru a explora).
- La fiecare episod, epsilon scade cu un factor de 0.998.
- Se plafonează la epsilon = 0.01 (Exploatare 99%, Explorare 1%).

5. Hiperparametrii Utilizați

Parametru	**Valoare**	**Descriere**
MINI_BATCH_SIZE	64	Numărul de exemple folosite la un pas de antrenare.
LEARNING_RATE	0.0001	Viteza de ajustare a greutăților.
GAMMA	0.99	Factorul de discount (cât contează viitorul).
REPLAY_MEMORY	15000	Capacitatea memoriei de experiență.
EPSILON_DECAY	0.998	Viteza de trecere de la explorare la exploatare.
SYNC_TARGET_FREQ	1000	Frecvența actualizării rețelei țintă (pași).

6. Rezultate Experimentale și Performanță

Antrenamentul a fost rulat pe parcursul a 5000 de episoade.



7. Concluzii

Arhitectura propusă demonstrează eficiența Deep Reinforcement Learning în medii vizuale. Utilizarea **Dueling DQN** combinată cu **Double Q-Learning** a permis agentului să învețe jocul Flappy Bird direct din pixeli, atingând performanțe supraumane în evitarea obstacolelor.

Provocările principale au fost ajustarea hiperparametrilor (în special rata de descreștere a lui epsilon și dimensiunea memoriei) pentru a balansa explorarea cu stabilitatea antrenamentului.