BrekoutAI

Progetto Intelligenza Artificiale

1.Descrizione Generale

Questo progetto implementa il famoso gioco Atari Breakout e costruisce un agente che sia in grado di vincere il gioco (ottenendo il punteggio massimo). Il gioco si compone di due modalità:

- PLAYER : l'utente può giocare a Breakout nel modo classico, muovendo il paddle a destra e a sinistra con i tasti A e D.
- CPU : l'agente gioca ad Atari e cerca di vincere il gioco, distruggendo tutti i blocchi senza far toccare alla pallina il bordo inferiore (game over).

Se si arriva alla vittoria o alla perdita della partita esce una schermata mostrand i punti ottenuti. L'orientamento della direzione destra o sinistra della pallina viene generato randomicamente.

2.Descrizione dell'agente e dell'algoritmo di IA

L'agente modellizza l'ambiente (il mondo di gioco) con un Markov Decision Process. Per cominciare, bisongna innanzitutto fare alcune considerazioni.
Il mondo di gioco è un mondo completamente deterministico, di conseguenza abbiamo a disposizione tutte le informazioni sul mondo di gioco (velocità della pallina, velocità del paddle, posizione...) perchè l'ambiente è completamente osservabile.
In questo senso ho deciso di applicare le seguenti restrizioni: l'agente non conosce tutte le informazioni sullo stato del gioco (come posizioni e velocità degli oggetti) ma solo una versione "riassuntiva" di queste informazioni, in modo da creare stati discreti finiti e in numero ridotto per implementare un MDP le cui dimensioni non fossero eccessive.
Queste informazioni, o stati dell'agente, sono le seguenti:

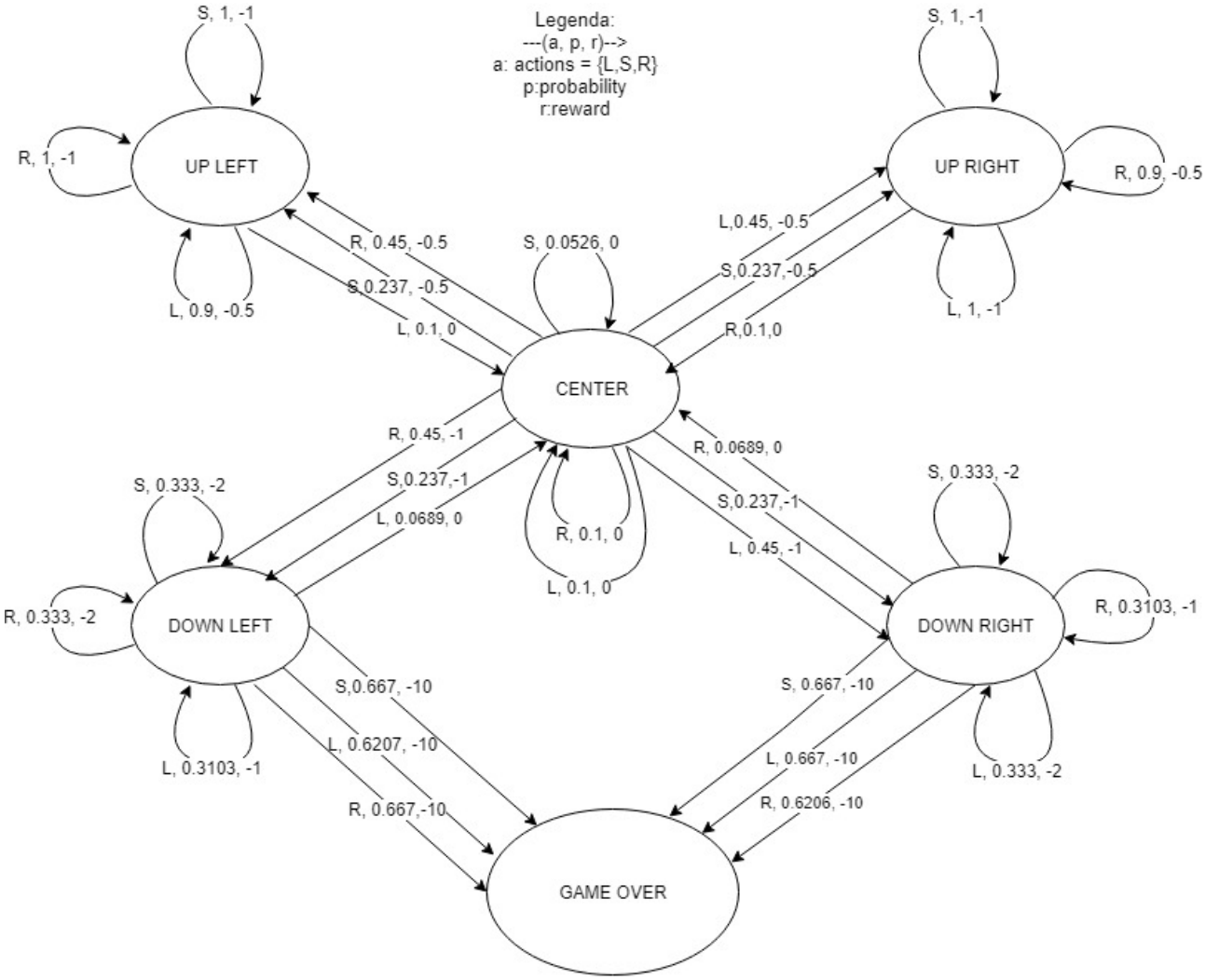
- CENTER : la distanza relativa tra il centro della pallina e il centro del paddle (lungo l'asse x) è minore della metà della dimensione del paddle (in altre parole, la pallina si trova "sopra" il paddle); successivamente è stato impostato un epsilon di sicurezza in modo tale che l'area centrale non corrispondesse alla dimensione del

paddle, ma fosse "un po' meno" di questa dimensione.
- UP_LEFT : la pallina si trova a sinistra (distanza relativa dei centri maggiore della metà della dimensione del paddle)rispetto al paddle e la pallina va verso l'alto.
- UP_RIGHT : a pallina si trova a destra (distanza relativa dei centri maggiore della metà della dimensione del paddle) rispetto al paddle e la pallina va verso l'alto.
- DOWN-LEFT : la pallina si trova a sinistra rispetto al paddle e la pallina va verso il basso.
- DOWN_RIGHT : a pallina si trova a destra rispetto al paddle e la pallina va verso il basso.
- GAME_OVER: fine del gioco (per come è stato implementato, non è entrerai mai in questo stato, in quanto appena la pallina tocca il bordo esterno il gioco termina, ma servirà questa informazione per per il MDP) .

Non sono state considerati stati riguardanti la distruzione dei blocchi in quanto non influenti sulla dinamica del gioco: infatti, lo scopo dell'agente è vincere la partita, o meglio dire non permettere il game over e perdere. Perciò saranno ininfluenti i blocchi e le relative reward.
Le azioni sono i tre possibili movimenti del paddle e sono LEFT (L), STOP (S) e RIGHT (R).

Per essere un MDP, bisogna che gli stati siano indipendenti dal passato, ovvero lo stato corrente non dipenda dallo stato passato. Chiaramente nel caso reale ogni stato presente non è indipendente dal passato: ad esempio, se il gioco si trova nello stato DOWN_RIGHT id un determinato frame del gioco, è ovviamente molto probabile che nel frame precedente lo stato sia ancora DOWN_RIGHT (infatti lo spostamento percorso dalla pallina o dal paddle tra un frame e l'altro è molto piccolo). Per questo bisonga teorizzare che gli stati di gioco cambiano in modo totalmente casuale, o perlomeno, visto che non rappresenta la situazione reale, come può essere osservato e percepito dall'agente. Ovviamente questo rappresenta un limite per l'agente, un'approssimazione del comportamento dell'ambiente. Questo, come spiegato sotto, influenzerà le probabilità impostate nel MDP.

Mostriamo qui di eseguito il MDP creato:



Comesono state calcolate le probabilità delle transizioni?

$p(s' = s'_i|s,a) = E[dist \mid s'_i] / \sum_j E[dist \mid s'_j] = (E[dist \mid s'_i] / Risoluzione.x)/ \sum_j (E[dist \mid s'_j] / Risoluzione.x)$

dove s' è lo stato successivo, s lo stato corrente, a l'azione del paddle, p la probabilità di transizione e $dist \mid s'_i$ è una variabile aleatoria che rappresenta la lunghezza della zona dove si può trovare la pallina dato lo stato in cui si troverà essendo in s e applicando a, cioè s'. Esempio, s' = X_RIGHT allora la pallina si troverà a sinistra del paddle, dist | X_RIGHT è la lunghezza che va dal paddle e fino al bordo destro (cioè la parte a destra del paddle). Questa variabile aleatoria è funzione della posizione della pallina e del paddle che, essendo per l'agente variabili causali con probabilità uniforme, anche $dist \mid s'_i$ sarà uniforme. La lunghezza di questa zona dipende dallo stato successivo di arrivo dato lo stato corrente e l'azione. Ad esempio, se dato un s e a, ho s' = CENTER, allora pr(dist =

paddle.size.x |CENTER) = 1, altrimenti zero per altri valori, cioè l'area dove si troverà la pallina corrisponde alla dimensione lungo x del paddle. Idem per S' = GAME_OVER, dove pr(dist = Risoluzione.x - paddle.size.x |GAME_OVER) = 1 per , altrimenti zero.
Se s' = UP_RIGHT (o DOWN_RIGHT), ovvero la pallina si trova a sinistra del paddle, la distanza assoluta che si trova sulla destra del paddle ha pr(Area |UP_RIGHT) = 1 / (Risoluzione.x – (paddle.size.x – 2*epsilon)), con una porbabilià uniforme dove dist assume un range di valori con minimo ball.radius – epsilon, dove il paddle può trovarsi a destra del bordo (non completamente a destra, teoricamente lascia una distanza pari a ball.radius – epsilon tra il bordo destro e il paddle) e massimo Risulzione.x – paddle.size.x dove il paddle si trova completamente a sinistra. (nel calcolo considero ball.radius – epsilon = 0 in quanto imposto epsilon ≈ ball.radius ma sempre epsilon < ball.radius altrimenti la configurazione degli stati cambierebbe).
Identico ragionamento per UP_RIGHT, DOWN_RIGHT e GAME_OVER.
Di questa varaibile aleatoria prendiamo il valore atteso : ragionando sul valore atteso, il valore atteso di, ad esmpio, dist | UP_LEFT sarà 1/2 * (Risoluzione.x – paddle.size.x) perchè il range di dist | UP_LEFT è (0, Risoluzione.x – paddle.x), ovvero E[dist | UP_LEFT] = Risoluzione.x/2 - paddle.size.x/2. Questo caso si verifica coincide quando il paddle si trova nella posizione centrale. Stesso ragionamento vale per lo stato DOWN_LEFT, UP_RIGHT, DOWN_RIGHT . Per CENTER o GAME_OVER il problema non si pone visto che dist | CENTER o dist | GAME_OVER diventano deterministici. Quinid generalizzando E[dist | s'] vanno calcolati ragionando sul fatto che il valore si otterrà se il paddle si troverà nella posione centrale , posizione "media" di qualsiasi posizione che può assumere il paddle.
Quindi ragionando su questo discorso si calcola p(s' | s,a) per ogni valore s' dato s e a (divido per la sommatoria di tutti i valori E[dist|s'] per avere la probabolità tra 0 e 1).
Alcuni esempi:
 p(s' = DOWN_LEFT | s = CENTER, a = RIGHT) = 0.45 / (0.45 + 0.45 + 0.1) = 0.45
p(s' = CENTER | s = DOWN_LEFT, a = LEFT) = 0.1 / (0.1 + 0.45 + 0.9) = 0.068965

Nel codice i valori vengono tutti precalcolati.

Per l'assegnazione delle reward, l'idea consiste nel penalizzare tutti quesgli stati quelle che portano al game over.
Di base quando s' = GAME_OVER ha reward -10 , s' = CENTER ha reward 0.
Per UP_LEFT e UP_RIGHT dipende dallo stato s e dall'azione a: se da s=X_LEFT mi sposto con a = LEFT, e raggingo s' = UP_LEFT, la reward sarà impostata a -0.5 (poco penality, perchè il paddle si avvicinerà alla pallina) mentre se raggiungo s' = CENTER sarà 0 (niente penalty, il paddle raggiunge la pallina). Se invece s = UP_LEFT e a = RIGHT, per s' = UP_LEFT arà una penalty maggiore (-1) perchè il paddle si allontana dalla pallina. Analogamente vale per UP_RIGHT. Per DOWN_RIGHT e DOWN_LEFT, bisogna prendere le corrispondenti penalty UP_RIGHT e UP_LEFT e raddoppiarle, visto che quando la pallina è in discesa è in uno stato più pericoloso rispetto a quando va verso l'alto. Per s' = GAME_OVER, indipendentemente da a e s, si avrà penalty -10 e, ugualmente, per s' = CENTER, la penalty vale 0.

Infine, per calcolare i valori della value function V(s) è stato implementato l'algoritmo value iteration (nel suo caso deterministico) presentato qui di seguito:

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

L'algoritmo viene eseguito all'inizio del gioco, di conseguenza una volta appresa la policy $\pi(s)$ l'agente prosegue con quella policy fino alla fine del gioco. Siccome la reward cumulata è finita, γ viene impostato a 1.

4.Considerazioni finali

Sotto tutte le ipotesi presentate l'agente è in grado di agire correttamente.

di conseguenza è possibile risolvere il gioco semplicemente utilizzando le informaioni molto semplificate sulla direzione del moto della pallina (UP o DOWN) e la posizione relativa tra il paddle e la pallina (LEFT, RIGHT, CENTER), senza esplicitamente conosce le velocità e le posizioni. Questo progetto rappresenta un esperimento che cercava di implementare un agente in grado di riuscire a vincere la partita usando stati finiti che rappresentassero le informazioni minime sufficienti allo scopo.