Intelligent robotic systems

Catch it

Gustavo Mazzanti

Giugno 2023

Indice

Capitolo 1

Caso di studio

1.1 Obiettivo

Per questo progetto ho implementato il gioco "preso-ripreso" nel simulatore Argos.
Il simulatore é stato impostato con una semplice arena con 4 pareti laterali e il pavimento, insieme a due robot che giocheranno sfidandosi.

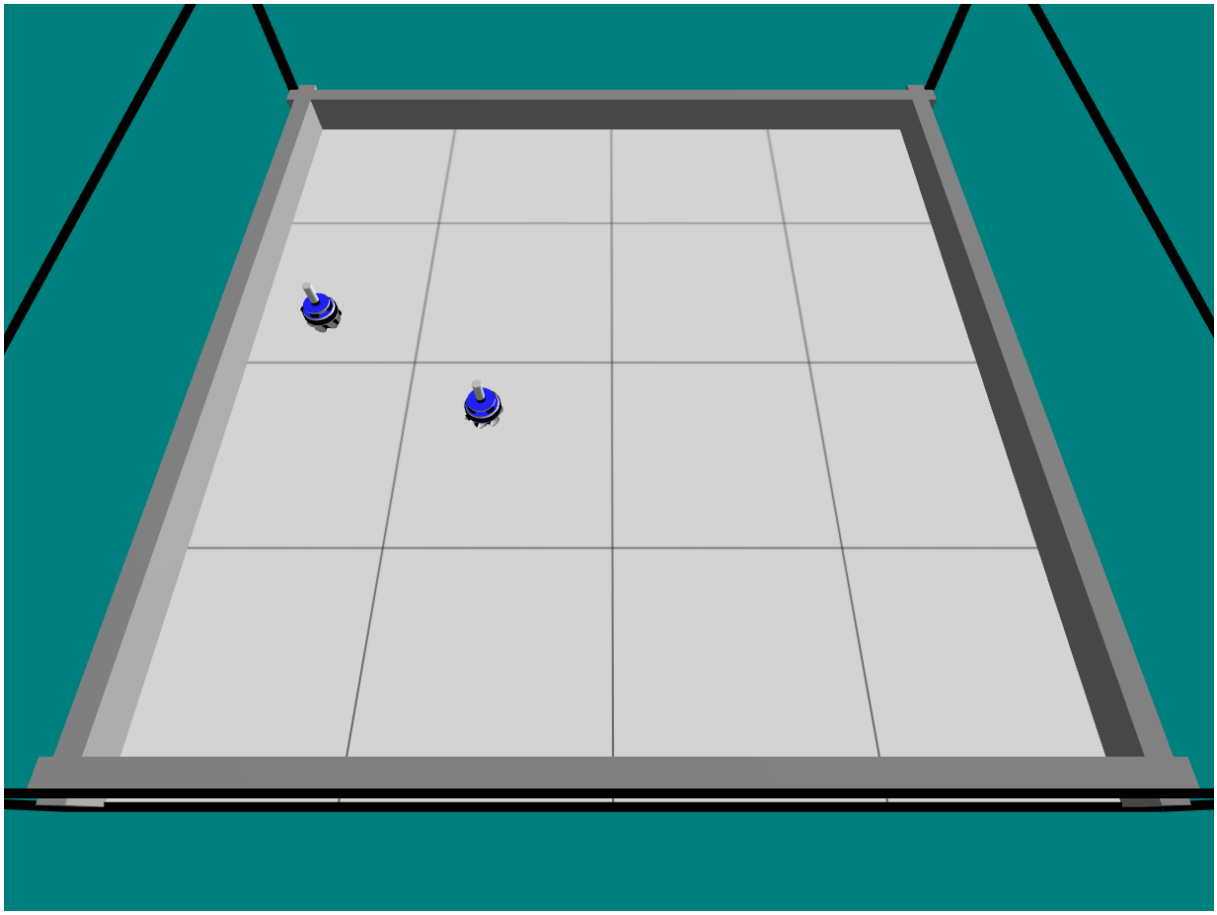

Figura 1.1: Arena simulazione robot.

Il gioco consiste nel cercare di toccare l'avversario per potergli "rubare" il ruolo, quindi ogni volta che il robot inseguitore riuscirà a toccare l'avversario questi si scambieranno di ruolo, a quel punto chi inseguiva inizierà a scappare e viceversa.

1.2 Organizzazione del lavoro

Prima di iniziare il lavoro è stato deciso di suddividere il progetto in due step:

1. implementare manualmente due controller che potessero fare rispettivamente target tracking e l'altro che potesse scappare dal target, entrambi evitando gli ostacoli;
2. sostituire uno dei due controller e utilizzare l'insegnamento tramite reinforcement learning. In questo caso abbiamo deciso di sostituire il controller del robot inseguitore.

Per il secondo step è stato scelto di utilizzare la libreria Qlearning che sfrutta una tabella di stati azioni e reward per poter prendere decisioni.

Capitolo 2

Motor schema controller

2.1 Ambiente

Come descritto in precedenza ho utilizzato una arena simulata nell'ambiente argos, questa arena ha 4 pareti lunghe 4 mt. che non consentono al robot di uscire dall'ambiente, abbiamo posizionato i robot in partenza sempre nello stesso punto e aggiunto all'ambiente 10 ostacoli di dimensioni random,i quali sono 5 cilindri e 5 box.

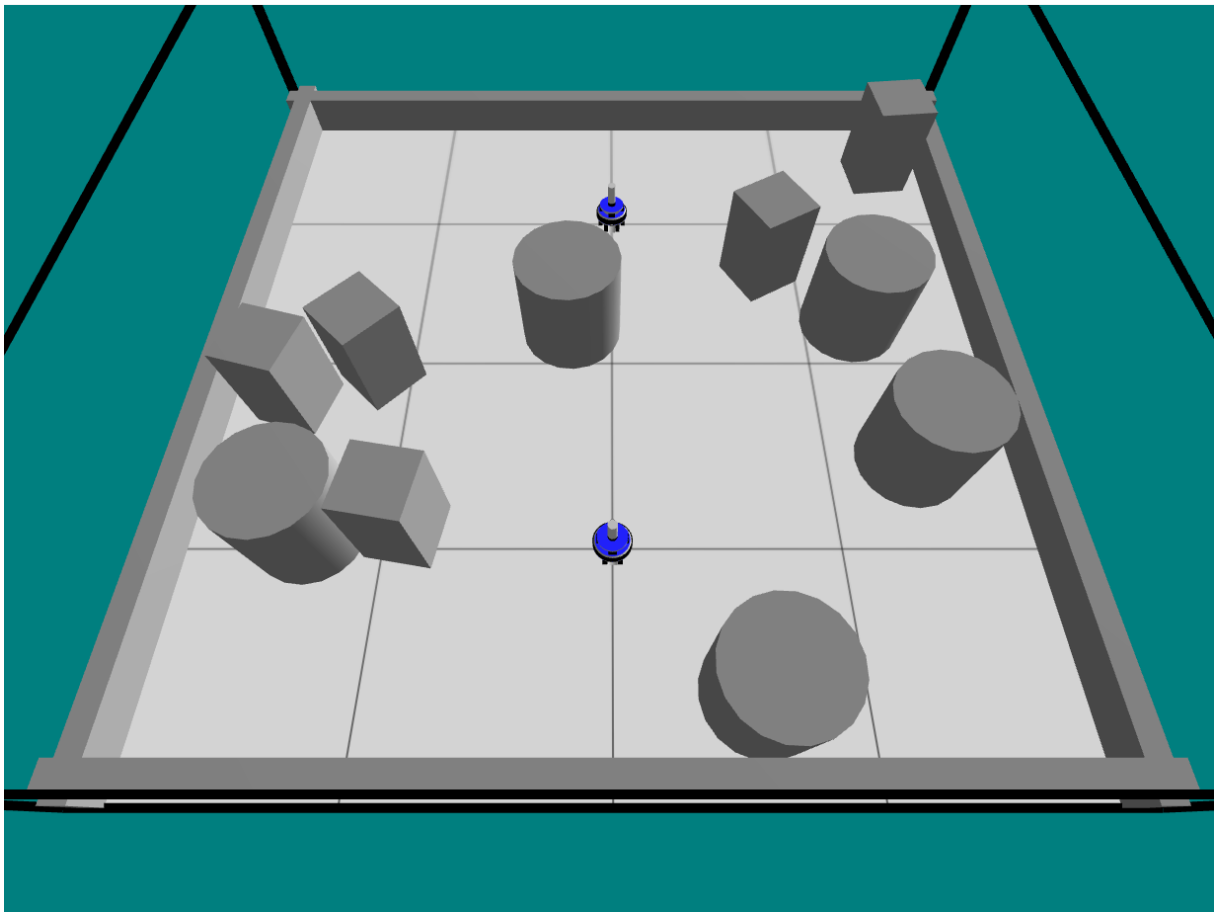

Figura 2.1: Arena con ostacoli.

2.2 Motor schema

Dopo aver visto diversi metodi per implementare un controller, ho deciso di utilizzare Motor Schema come pattern.
Questo metodo consiste nel sommare un insieme di vettori che rappresentano delle possibili buone direzione per il robot, in modo tale da ottenere un unico vettore rappresentante la direzione ottimale che il robot possa scegliere.

2.2.1 Motor schema per avoid obstacle

Per questo task abbiamo utilizzato tutti i sensori di prossimità presenti nel robot, per ogni sensore riceviamo come valori la distanza a cui si trova un oggetto (max 10cm) e da che angolo sta arrivando il segnale rispetto al fronte del robot.
Grazie alla libreria Vector è stato possibile convertire questi dati in vettori partenti dal centro del robot, in seguito sarà necessario modificarne la direzione rendendoli opposti rispetto a quella da cui arriva il segnale.
Infatti sarà sufficiente per ogni sensore che sta rilevando ostacoli, applicare un vettore opposto al segnale in modo tale da consigliare al robot di allontanarsi dall'ostacolo. Una volta recuperati tutti i vettori del robot, sempre tramite la libreria Vector sarà possibile sommarli in modo tale da ottenere un unico vettore rappresentante la direzione migliore che il robot possa prendere.

```
143 function vector_avoid_ostacles_exclude_target()
144     vec = {length = 0, angle = 0}
145     target = vector_catch_it()
146     target.length = 0.5
147     for i=1,#robot.proximity do
148         ang = robot.proximity[i].angle
149         if ang > 0 then
150             ang = ang - math.pi
151         else
152             ang = ang + math.pi
153         end
154         vec = vector.vec2_polar_sum(vec, {length = robot.proximity[i].value, angle = ang})
155     end
156 
157     vec = vector.vec2_polar_sum(vec, target)
158     return vec
159 end
```

Figura 2.2: Funzione per calcolare il vettore dell'avoid obstacles.

In questo metodo vediamo come sia sufficiente sommare i vettori prelevati dai sensori di prossimità, insieme al vettore corrispondente al target (vettore restituito dal metodo *vector_catch_it()*).
Il metodo per scappare dal nemico risulta uguale, con la differenza che in questo caso viene richiamato il metodo *vector_get_out()*, metodo che restituisce un vettore in direzione opposta rispetto al segnale.

2.2.2 Motor schema per target tracking e target avoiding

Per questo task sarà sufficiente utilizzare il sensore range and bearing, questo sensore è in grado di capire distanza (massimo 3 mt.) dell'altro robot e l'angolo da cui arriva il segnale, sempre rispetto al centro.
Utilizzando sempre la libreria Vector sarà possibile sfruttare questi dati per calcolare il vettore ottimale per il task da eseguire:

- vettore target tracking: il vettore sarà orientato nella direzione da cui arriva il segnale;
- vettore target avoiding: il vettore sarà orientato nella direzione opposta rispetto al segnale.

```lua
function vector_catch_it()
    vec = {length = 0, angle = 0}
    if robot.range_and_bearing[1] ~= nil then
        vec.length = robot.range_and_bearing[1].range
        vec.angle = robot.range_and_bearing[1].horizontal_bearing
    end
    return vec
end

function vector_get_out()
    vec = {length = 0, angle = 0}
    if robot.range_and_bearing[1] ~= nil then
        vec.length = robot.range_and_bearing[1].range
        vec.angle = robot.range_and_bearing[1].horizontal_bearing
        if vec.angle > 0 then
            vec.angle = vec.angle - math.pi
        else
            vec.angle = vec.angle + math.pi
        end
    end
    return vec
end
```

Figura 2.3: Funzioni che calcolano il vettore del nemico.

Qui possiamo vedere i due metodi *vector_catch_it()* e *vector_get_out()*, implementati rispettivamente come vettore target tracking e vettore target avoiding.

Una volta completati entrambi i task è stato sufficiente sommare il vettore avoid obstacle con il vettore target tracking per il robot che insegue e il vettore avoid osbtacle con il vettore target avoiding per il robot che scappa come visto in precedenza (fig. 2.2).

Capitolo 3

Reinforcement learning controller

In questo caso abbiamo deciso di istruire solo il robot che deve inseguire, per il robot che scappa è stato utilizzato il controller implementato in precedenza.
In questo task abbiamo deciso di escludere la avoid obstacle per semplicità.

3.1 Ambiente

Come ambiente di addestramento abbiamo utilizzato una arena simulata nell'ambiente argos, questa arena ha 4 pareti lunghe 2 mt. che non consentono al robot di uscire dall'ambiente, abbiamo posizionato i robot in partenza in posizioni random e abbiamo lasciato l'ambiente libero, privo di ostacoli.

3.2 Qlearning

Per istruire il robot è stata utilizzata la libreria Qlearning, la quale utilizza una tabella csv mappata come segue: le righe sono i possibili stati i cui si può trovare il robot, le colonne sono le possibili azioni che può decidere di prendere e i valori della tabella sono i reward ricevuti in un determinato stato dopo aver eseguito una determinata azione.

3.3 Implementazione

Anche in questo caso abbiamo scelto di utilizzare il sensore range and bearing per poter capire la distanza e l'angolo del robot avversario.
Per prima cosa abbiamo dovuto scegliere gli stati, le azioni e la funzione di reward per poter istruire il robot:

3.3.1 Stati

Inizialmente avevo pensato di creare uno stato per ogni combinazione possibile tra i possibili cm di distanza rilevabile dal sensore range and bearing (max 300cm) con ogni grado di angolo rilevabile dallo stesso sensore (max 360°), in totale sarebbero usciti 360x300 = 108000 stati, troppi per effettuare un addestramento rapido.
Allora ho creato dei range per i valori disponibili mappandoli come segue: 10 range per la distanza (uno ogni 30cm) e 16 range per l'angolo (uno ogni 22,5°). A questo punto ho creato due array con i vari range (array da 1 a 16 per l'angolo e array da 1 a 10 per la distanza), grazie a questi array è stato possibile implementare una funzione in grado di mappare una coppia di distanza e angolo in uno valore univoco per identificare lo stato attuale, ottenendo valori tutti compresi tra 1 e 160 (10x16 le possibili combinazioni di stato e angolo).

```lua
function get_index_of_state(state)
    local index_ang = 0

    for i = 1, #angle_states do
        if state.angle <= angle_states[i] then
            index_ang = i
            break
        end
    end
    for i = 1, #distance_states do
        if state.range <= distance_states[i] then
            index_dist = i
            break
        end
    end

    return (((index_ang - 1) * #distance_states) + index_dist)
end
```

Figura 3.1: Metodo per ricevere l'indice numerico univoco dello stato attuale.

```lua
--States: 128 in total (16 angle states * 8 distance states)
angle_states = { -157.5, -135, -112.5, -90, -67.5, -45, -22.5, 0, 22.5, 45, 67.5, 90, 112.5, 135, 157.5, 180 }
distance_states = {  30, 60, 90, 120, 150, 180, 210, 300}
number_of_states = #angle_states * #distance_states
```

Figura 3.2: Gli array dei possibili angoli e distanze.

Aggiornamento: Come vediamo in foto sono stati successivamente uniti gli ultimi 3 range della distanza, calando il numero di range da 10 a 8 e di conseguenza il numero degli stati da 160 a 128. Questo perchè si era notato anche dopo tante epoche il robot rischiava di non aver mai esplorato tutti gli stati compresi tra i 250cm circa fino a 300cm.

3.3.2 Azioni

Per agevolare l'addestramento ma senza limitare troppo i possibili movimenti ho pensato di stabilire 8 direzioni (Riferimenti cardinali dei possibili movimenti, considerando dal fronte del robot: N, S, E, O, NE, NO, SE, SO). Quindi il robot che insegue potrà decidere di andare in una di queste direzioni, quindi per ogni stato, nella tabella csv, troveremo 8 azioni. In totale otteniamo 8x160 = 1280 combinazioni di stati e azioni.

```lua
--Actions: 8 in total
velocity_direction_names = {"N", "NW", "W", "SW", "S", "SE", "E", "NE"}
velocity_directions = {
    ["N"] = 0,
    ["NW"] = math.pi / 4, -- 45 degree
    ["W"] = math.pi / 2, -- 90 degree
    ["SW"] = 3 * math.pi / 4, -- 135 degree
    ["S"] = math.pi, -- 180 degree
    ["SE"] = - 3 * math.pi / 4, -- -135 degree
    ["E"] = - math.pi / 2, -- -90 degree
    ["NE"] = - math.pi / 4, -- -45 degree
}

number_of_actions = #velocity_direction_names
```

Figura 3.3: Gli array delle possibili azioni.

3.3.3 Reward

La funzione di reward ha subito diverse variazioni nel corso dell'implementazione: inizialmente ho pensato ad utilizzare la distanza rispetto al robot target come punto di riferimento per calcolare il reward, calcolandolo come differenza della distanza allo step precedente.
Questo approccio si è rivelato fallimentare in quanto il robot non riusciva a scegliere in modo ottimale la direzione da prendere e tendenzialmente si ritrovava ad allontanarsi dal robot target, perdendo quasi subito il segnale del robot target.
Allora ho proseguito calcolando il reward come somma tra la differenze del nuovo e del vecchio angolo e la differenza della nuova e della vecchia distanza, normalizzando il tutto da 0 a 1.
Anche in questo caso ho notato che la distanza avesse ancora troppo peso e il robot non riusciva a scegliere in modo corretto la direzione migliore. Valutando in seguito che la direzione ottimale l'avrebbe data in particolar modo l'angolo a cui si trovava ho fatto in modo che i reward fossero pesati per 1/3 con la differenza delle distanze e per 2/3 con la differenza degli angoli, in questo modo l'angolo ha avuto più peso sul reward causando una notevole differenza sulle scelte del robot.
I calcoli sono stati fatti come segue:

- Distanza: semplice differenza tra la distanza attuale e la distanza prima di effettuare il movimento, normalizzato in valori da 0 a 0.33, in caso la differenza dovesse risultare negativa il reward relativo sarebbe 0;
- Angolo(valori da -180 a 180, angolo del fronte del robot 0): se l'angolo = 0 allora reward = 0.66, se angolo <= 10 e angolo >= -10 allora reward = 0.65, se vecchio angolo e nuovo angolo sono entrambi positivi o entrambi negativi allora calcola la differenza del vecchio rispetto al nuovo (utilizzando il valore assoluto per i valori negativi allora se risultato positivo normalizza la differenza nel valore da 0 a 0.66 se negativo reward = 0 in tutti gli altri casi reward = 0. Così facendo sono riuscito a evitare che il robot si bloccasse sulla retro, non riuscendo a decidere se muoversi a destra o sinistra.

Per velocizzare ulteriormente l'addestramento ho pensato di ridurre lo spazio dell'arena di training in modo tale da limitare il più possibile la situazione in cui i robot smettono di "vedersi".

```lua
function get_reward(state, old_state)
    if state.range == -1 then
        --If i don't see the hero
        return 0
    elseif state.range < RANGE_MIN then
        --If i touch the hero
        return 1
    else
        if (state.angle > 0 and old_state.angle > 0) or (state.angle < 0 and old_state.angle < 0) then
            --If I'm entering the goal towards my center, calculate the difference from old state angle and new state angle normalazite to 0 and 0.66
            angle_reward = (math.abs(state.angle) - math.abs(old_state.angle)) / 540
        elseif state.angle == 0 then
            -- If the hero is in front of me i take the max reward for the angle
            angle_reward = 0.66
        elseif state.angle < 10 and state.angle > -10 then
            -- If the hero is in front of me i take 0.65 reward for the angle
            angle_reward = 0.65
        else
            -- Else 0
            angle_reward = 0
        end
        -- Calculate the difference from old state distance and new state distance normalazite to 0 and 0.33
        distance_reward = (state.range - old_state.range) / 900
        if angle_reward < 0 then
            angle_reward = 0
        end
        if distance_reward < 0 then
            distance_reward = 0
        end
        -- Reward calculated as the sum of the two rewards (66% angle reward + 33% distance reward)
        return angle_reward + (2*distance_reward)
    end
end
```

Figura 3.4: Funzione per il calcolo del reward.

3.4 Parametri di calcolo

Una volta ottimizzata la funzione di reward ho provato a variare leggermente i tre parametri alpha e gamma, notando che la configurazione ottimale fosse lasciare alpha a 0,1 e gamma a 0,9.
Per quanto riguarda i movimenti, la variabile MOVE_STEPS è stata abbassata da 5 a 3 per poter prendere decisioni più frequentemente, in modo tale da evitare che il robot facesse una rotazione troppo lunga evitando di perdere tempo in movimenti, rimanendo indietro rispetto al target.
Infine è stata aumentata la velocità massima del robot inseguitore in modo tale da potersi, ogni tanto, avvicinare al robot target.

3.5 Addestramento

Una volta ultimato il progetto è stato effettuato un primo addestramento da 10 epoche per testare che funzionasse correttamente, il risultato è stato notevolmente buono nonostante il breve addestramento.
In seguito è stato avviato un secondo training da 100 epoche, per cercare di coprire anche gli stati che non erano ancora stati esplorati. Nonostante questo training più lungo ho notato che alcuni stati non venivano mai considerati, tutti quelli che rappresentavano una distanza superiore a 240cm, perciò ho deciso di eliminare gli stati che comprendevano distanze da 240 a 270 e da 270 a 300, creandone una unica da 210 a 300. Così facendo ho visto che subito dopo 10 epoche risultava un netto miglioramento in termini di copertura degli stati esplorati, provando ad allenare ulteriormente per un totale di 100 epoche ho notato come queste ulteriori 90 epoche abbiano portato dall'avere circa 100 situazioni inesplorate, ad averne solo 50 circa.

Capitolo 4

Conclusioni

In conclusione sostengo che i requisiti iniziali siano stati soddisfatti, il robot inseguitore, nonostante la velocità del target sia la stessa,in parecchi casi è riuscito a raggiungere il target. Queste situazioni si verificano nel momento in cui il robot che scappa si ritrova in un vicolo cieco con alle spalle il robot inseguitore.
Vorrei anche sottolineare come il numero di epoche di addestramento dopo le 10 possa essere ininfluente in quanto le uniche differenze sostanziali incontrate sono la problematica di non riuscire a coprire fin da subito tutti gli stati possibili, notando quindi alcuni comportamenti insoliti in determinate situazioni che risultano più frequenti nell'addestramento di 10 epoche rispetto al training di 100 epoche.
Concludendo poi con un ultimo training da 1000 epoche, che in questo caso ha coperto circa altre 10 situazioni inesplorate delle 45 circa totali rimaste inesplorate dopo 100 epoche.

4.1 Miglioramenti futuri

Avrei pensato ad alcuni possibili sviluppi futuri per poter migliorare il comportamento del robot:

- Istruire l'inseguitore anche ad evitare gli ostacoli;
- Includere anche la possibilità di scegliere e monitorare la velocità del robot;
- Cercare di eliminare l'oscillazione del movimento "N";