Università degli Studi di Verona

Report del progetto di Intelligenza Artificiale

Ambiente acquatico OpenAI Gym e policy DQN per collision avoidance

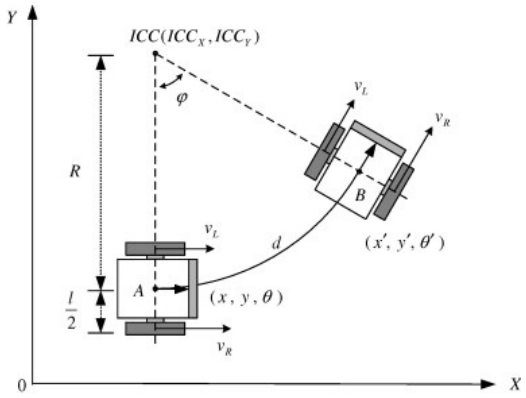
Sebastiano Fregnan

27 gennaio 2021

1 Ambiente

1.1 Moto del drone [1]

Il drone utilizzato monta un sistema differential drive a due motori allineati su di un asse, ad una distanza ℓ tra loro; ognuno di essi genera una spinta (*thrust*) che produce una certa velocità, rispettivamente v_L e v_R. Per descrivere la cinematica, impostiamo il sistema di riferimento (*frame*) del drone Σ_D con origine O_D al centro dell'asse (le ruote saranno quindi esattamente a $\ell/2$ da Σ_D, sulla stessa direzione ma in verso opposto) e, rispetto ad un frame mondo Σ_W, definiamo il punto O_D come la posizione p del drone; definiamo inoltre θ come l'angolo tra Σ_D rispetto a Σ_W.



Dato lo stato (x, y, θ) del drone al tempo t, per trovarne lo stato (x', y', θ') dopo un tempo T troviamo innanzitutto la velocità angolare ω ed il raggio di rotazione R a partire dalle formule di velocità angolare

$$\begin{cases} \omega = (R - \ell/2)v_L \\ \omega = (R + \ell/2)v_R \end{cases} \implies R = \frac{\ell}{2}\frac{v_R + v_L}{v_R - v_L} \qquad \omega = \frac{v_R - v_L}{\ell}\,.$$

Successivamente identifichiamo il *centro di curvatura istantaneo ICC* come il punto attorno al quale avviene la rotazione

$$ICC \doteq \begin{bmatrix} ICC_x \\ ICC_y \end{bmatrix} = \begin{bmatrix} x - R\sin\theta \\ y + R\cos\theta \end{bmatrix}$$

ed ora ci basta semplicemente calcolare la rotazione durante il periodo T, ovvero per l'angolo ωT

$$\begin{cases} p' = \mathcal{R}_z(\omega T)\,[p - ICC] + ICC \\ \theta' = \theta + \omega T \end{cases} \implies \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega T) & -\sin(\omega T) & 0 \\ \sin(\omega T) & \cos(\omega T) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega T \end{bmatrix}$$

Con questo risultato modelliamo *quasi* completamente la cinematica del drone: un caso particolare si ha quando $v_L = v_R$, che ottiene $\omega = 0$ e $R \to \infty$, ovvero un movimento puramente lineare. Qui ci basta semplicemente applicare la velocità $v_{tot} = v_L + v_R$ su entrambe le direzioni (opportunamente aggiustata rispetto all'angolo θ del drone ovviamente)

$$\begin{cases} p' = p + \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} v_{tot} \\ \theta' = \theta \end{cases} \implies \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 0 \end{bmatrix} v_{tot}$$

In alternativa a questo approccio a due casi possiamo ridurre tutta la cinematica ad una singola forma se ammettiamo un piccolo errore nel caso lineare: impostiamo una soglia minima $\varepsilon > 0$ per $|v_R - v_L|$. In questo caso, per velocità molto vicine otterremo un rotazione con raggio enorme, ovvero un moto quasi-lineare. A questo punto rimpiazziamo $v_R - v_L$ nelle equazioni, riscrivendole come

$$\delta = \operatorname{sgn}(v_R - v_L)\,\max\{|v_R - v_L|, \varepsilon\} \implies R = \frac{\ell}{2}\frac{v_R + v_L}{\delta} \qquad \omega = \frac{\delta}{\ell}$$

e utilizziamo solamente la cinematica rotazionale già descritta sopra.

1.2 Moto delle onde

Per simulare la corrente dell'acqua viene utilizzato un vettore velocità variabile superimposto sull'intero mondo (ignoriamo quindi la dinamica dei fluidi nei pressi delle superfici degli ostacoli e del drone). Temporalmente sincronizzato con il moto del drone (quindi ogni T), il moto ondoso subisce una leggera variazione nell'intervallo $[-\sigma, +\sigma]$ su entrambe le direzioni, senza mai superare una certa velocità massima. Questo ci permette di simulare un cambio di orientazione della corrente relativamente dolce.

[1] CS W4733 NOTES - Differential Drive Robots, tratto da *Dudek and Jenkin - Computational Principles of Mobile Robotics*

1.3 Implementazione

L'implementazione dell'ambiente è stata effettuata in codice Python utilizzando la libreria OpenAI Gym, che offre delle buone API per la creazione ed il rendering.

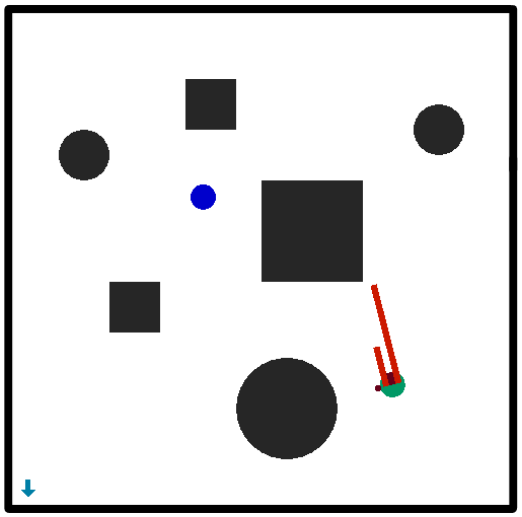


Figura 1: Render di `AquaEnv-v2`

L'ambiente è un quadrato di lato 100 *pixel*; il range di spinta dei singoli motori è in $[0.2, 0.5]$ (con $\varepsilon = 1e{-}8$ per i casi quasi-lineari), mentre la velocità delle onde è in $[-0.05, 0.05]$ per entrambe le direzioni, con una varianza ad ogni istante temporale di al più 0.001; l'angolo del drone è espresso in $[-\pi, +\pi]$, dove lo 0 indica la direzione verso nord. L'istante temporale è scandito dalla variabile `tau`, di default settata `tau` $= 1$, dando quindi a tutte le velocità sopra descritte l'unità di misura *pixel/instant*; per confronto possiamo pensare a convertire tale unità in m/s, ottenendo un drone con velocità lineare massima di 1 m/s.

Il drone ha raggio 2.5 *pixel* e la lunghezza dell'asse del differential drive è 2.5 *pixel*; il goal ha anch'esso raggio 2.5 *pixel*. Gli ostacoli sono di forma circolare oppure quadrata, con dimensioni variabili anche imponibili dall'utente (negli ostacoli di default le dimensioni sono piccole, medie o grandi, rispettivamente di raggio/lato $5, 10, 20$ *pixel*). Il drone ed il goal possono avere posizione casuale oppure fissa (rispettivamente in $(55, 30)$ e $(15, 65)$), mentre gli ostacoli hanno posizione fissa; ostacoli e moto ondoso possono essere disabilitati singolarmente.

Ad ogni istante temporale viene eseguita una azione, con un massimo di `time_limit` $= 1000$ azioni; ad ognuna di esse corrisponde un certo reward:

- $+10$ raggiunto lo stato di goal;
- -10 avvenuta una collisione con un ostacolo o con il bordo dell'ambiente oppure raggiunto `time_limit`;
- $\big(\text{dist}(prev, goal) - \text{dist}(curr, goal)\big) * 0.7$ in tutti gli altri casi, ovvero la differenza tra la distanza attuale dal goal e quella all'istante precedente (andando quindi a premiare il drone solo quando si avvicina allo stato di goal).

Quando si esegue il render dell'ambiente ci viene mostrato:

- il drone (in verde), provvisto di direzione e ICC (in vinaccia) e dei vettori per le spinte (in rosso se positive, in blu se negative, queste ultime mai utilizzate ovviamente a causa del range $[0.2, 0.5]$);
- il goal (in blu) e gli ostacoli (in grigio);
- il vettore del moto ondoso superimposto (in azzurro, sempre posizionato in basso a sinistra).

Tutti i vettori (spinte e moto ondoso) hanno dimensione di base 8 *pixel* e vengono scalati in lunghezza in base alla magnitudo dei vettori che mantengono i valori. Di default, lo scaling visivo applicato a tutti i valori dell'ambiente è 5 (quindi si aprirà una finestra 500×500, gli ostacoli avranno raggio $25, 50, 100$ *pixel*, ecc).

L'ambiente è disponibile nella versione con stato delle azioni continuo oppure discreto; in quest'ultimo le azioni sono 3, rispettivamente $(0.2, 0.5)$, $(0.5, 0.2)$ e $(0.5, 0.5)$, ovvero *ruota a sinistra*, *ruota a destra*, *vai dritto*.

1.4 Utilizzo

L'ambiente è istanziabile come un qualsiasi ambiente OpenAI Gym, ovvero con il comando

```
env = gym.make("AquaEnv-v0", **params)
```
oppure
```
env = gym.make("AquaContinuousEnv-v0", **params)
```

dove `params` è un dizionario di valori booleani definito come

```
params = {"obstacles": bool|list, "waves": bool,
          "random_goal": bool, "random_boat": bool}
```

in cui `"obstacles"` e `"waves"` indicano la presenza o meno di ostacoli e moto ondoso; i valori di default di questi parametri sono rispettivamente `False`, `True`, `True`, `True`. Gli ostacoli di default sono

- circolare in $(15, 75)$, raggio 5;
- circolare in $(20, 35)$, raggio 10;
- quadrato in $(65, 85)$, lato 5;
- circolare in $(85, 20)$, raggio 10;
- circolare in $(85, 75)$, raggio 5.

ma si possono impostare degli ostacoli arbitrari passando una lista di tuple in `"obstacles"` anziché il valore booleano; tali tuple devono essere nella forma

```
(numpy.ndarray(2,), "c"|"r", int|(int,int))
```

dove l'ultimo elemento è il raggio del cerchio se si usa `"c"` oppure le dimensioni (lunghezza e altezza) se si usa `"r"`, mentre il primo elemento è la posizione dell'ostacolo. Per comodità offerti anche gli ambienti

- `"-v1"`, in cui di default `"obstacles = True"`;
- `"-v2"`, in cui anziché quelli di default vengono utilizzati gli ostacoli:
  - circolare in $(15, 70)$, raggio 5;
  - quadrato in $(25, 40)$, lato 10;
  - quadrato in $(40, 80)$, lato 10;
  - circolare in $(55, 20)$, raggio 10;
  - quadrato in $(60, 55)$, lato 20;
  - circolare in $(85, 75)$, raggio 5.

Come in ogni ambiente OpenAI Gym, chiamare `state = env.reset()` riporta l'ambiente allo stato iniziale (generando delle nuove posizioni per il drone e il goal e impostando una nuova velocità alle onde se richiesto), `state, ... = env.step(action)` esegue l'azione specificata, mentre `env.render()` mostra l'ambiente a schermo (il quale deve essere chiuso al termine utilizzando `env.close()`). Oltre al nuovo stato, la funzione `step` ritorna il reward ottenuto (come descritto nella sezione 1.3), un booleano di completamento dell'espisodio e il dizionario `info` contenente la causa della terminazione, con uno solo dei campi `"Termination.success"`, `"Termination.collided"`, `"Termination.time"` posto a `True`. Per l'ambiente continuo `action` deve essere un array oppure un `ndarray` Numpy, per quello discreto un intero da 0 a 3. Nelle funzioni sopra, `state` è un `numpy.ndarray` di 5 valori, rispettivamente lo stato (x, y, θ) del drone e la posizione (x_G, y_G) del goal.

2 Policy

Per verificare la buona costruzione dell'ambiente sono stati effettuati diversi test utilizzando DQN; prima di mostrare i risultati, descriviamo brevemente l'algoritmo.

2.1 Deep Q-Network

Il Reinforcement Learning è uno dei tre paradigmi del Machine Learning (oltre a Supervised e Unsupervised Learning) e, vedendo l'ambiente come un Markov Decision Process, permette all'agente di imparare attraverso il giusto bilancio tra *exploration* ed *exploitation* dell'ambiente, ovvero il provare nuovi percorsi e lo sfruttare quelli *buoni* già scoperti.

Nel caso di action/state-space discreti, questo bilanciamento è ottenuto valutando ripetutamente le azioni compiute nei vari stati e misurando la *qualità* di ogni coppia stato-azione in base all'outcome presente e *futuro* (ovvero quanto ci aspettiamo di ottenere sulla base di ciò che abbiamo precedentemente scoperto), salvando il tutto in una tabella $Q(S, A)$; questo da origine agli algoritmi Q-Learning in cui, trovandoci nello stato S, dopo aver compiuto l'azione A effettuiamo l'aggiornamento tramite la riscrittura dell'equazione di Bellman

$$Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha\big(R + \gamma \max_a Q(S', a)\big)$$

dove α è il *learning rate* e γ è il *discount factor*, e alla fine costruiamo la policy usando, per ogni stato, l'azione con la Q-value maggiore.

Nel caso di action-state discreto ma space-state continuo non possiamo più rappresentare la funzione $Q(S, A)$ come una tabella, ma dobbiamo fare affidamento su una sua approssimazione: qui si apre il campo del Deep Reinforcement Learning, che mira a risolvere i problemi del Reinforcerment Learning usando le tecniche del Deep Learning, ovvero mediante l'utilizzo di reti neurali profonde (DNN). Il proseguimento naturale di Q-Learning nel caso di state-space continuo prende il nome di Deep Q-Learning, che può essere implementato con l'algoritmo Deep Q-Network; questo usa una DNN *modello* per approssimare la quality function (ottiene $Q_{\mathbf{w}}(s, a) \approx Q(s, a)$) e, ad ogni passo di ogni episodio, viene allenata utilizzando un *minibatch* casuale di tutte le osservazioni (s, a, r, s', d) raccolte. L'allenamento aggiorna i pesi usando $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$, dove la funzione di costo

$$E(\mathbf{w}) = \Big(Q_{\mathbf{w}}(s, a) - \big(R(s, a, s') + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(s', a')\big)\Big)^2$$

è un ulteriore rivisitazione dell'equazione di Bellman adattata usando Mean Squared Error, per essere utilizzata nell'ambito DNN. Notiamo la presenza di $Q_{\bar{\mathbf{w}}}(s,a)$, ovvero di un'altra DNN identica detta *target*, che rappresenta la rete ideale a cui vogliamo convergere, avente pesi $\bar{\mathbf{w}}$; ovviamente questa rete non ci è nota ed è nostro obiettivo ottenerla, ma possiamo utilizzare al suo posto una versione identica alla rete modello, aggiornata più di rado, utilizzando $\bar{\mathbf{w}} \leftarrow \tau\mathbf{w} + (1-\tau)\bar{\mathbf{w}}$, dove τ è un iperparametro molto piccolo tra 0 e 1. Aggiungendo l'*exploration rate* ε per aumentare l'esplorazione iniziale da parte della rete (incentivando la scelta casuale delle azioni) e diminuendolo man mano possiamo garantire, a patto di aver impostato correttamente tutti gli iperparametri, la convergenza dell'algoritmo alla policy ottimale $\pi_*(s) = \arg\max_a Q_{\mathbf{w}}(s,a)$.

2.2 Implementazione

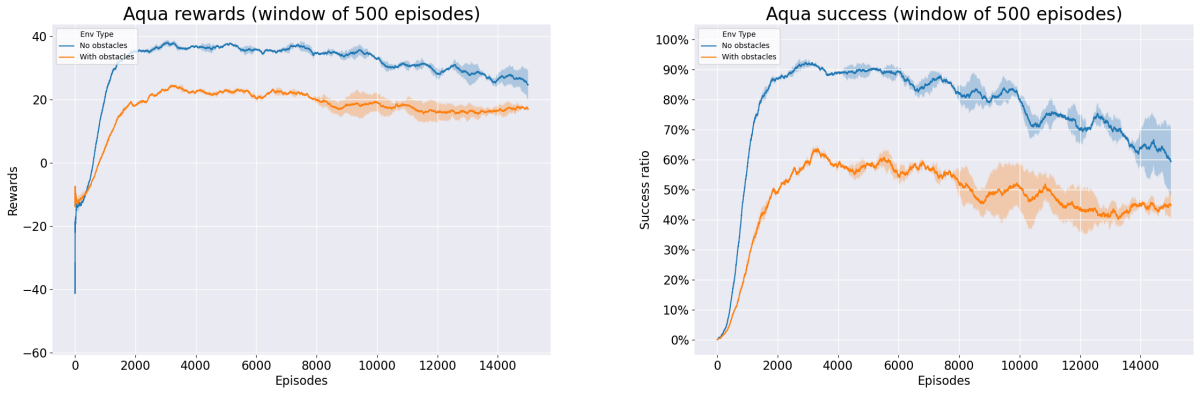
Per generare delle policy è stato utilizzato l'algoritmo DQN, con le seguenti caratteristiche:

- rete modello con 5 input (lo stato `state`), 2 hidden layer a 64 neuroni ReLU e 4 output (la Q-value di ognuna delle azioni discrete); rete target ovviamente identica (architettura e pesi iniziali) alla rete modello, aggiornata con `tau = 0.005`;
- input (stato) normalizzato tra 0 e 1;
- `epsilon_init = 1.0`, `epsilon_final = 0.01`, `epsilon_decay = 0.999`
- replay buffer di 50000 osservazioni, minibatch da 64, `gamma = 0.98` (discount);
- funzione di costo "doppia" $E(\mathbf{w}) = \left(Q_{\mathbf{w}}(s,a) - \left(R(s,a,s') + \gamma\, Q_{\bar{\mathbf{w}}}\left(s', \arg\max_{a'} Q_{\mathbf{w}}(s',a')\right)\right)\right)^2$, in cui anziché usare la a' che massimizza $Q_{\bar{\mathbf{w}}}(s',a')$ usiamo l'azione che massimizzerebbe la q-value per lo stesso s' ma sulla rete modello, ovvero $\arg\max_{a'} Q_{\mathbf{w}}(s',a')$;

2.3 Risultati

I test sono stati effettuati valutando l'algoritmo nei due casi di studio, cioè *con* e *senza* ostacoli, impostando l'ambiente con posizione iniziale e del goal casuali e moto ondoso presente.

I dati delle performance sono raccolti durante il training: per ognuno dei due casi di studio vengono effettuati 4 addestramenti da 15000 episodi ciascuno e si collezionano reward e successo di ogni episodio. Per ognuno dei casi di studio viene rappresentato il reward medio (a sinistra) e la percentuale media di successo (a destra), entrambi con finestra di 500 episodi; i grafici mostrano la media e la varianza dei valori nei 4 addestramenti.



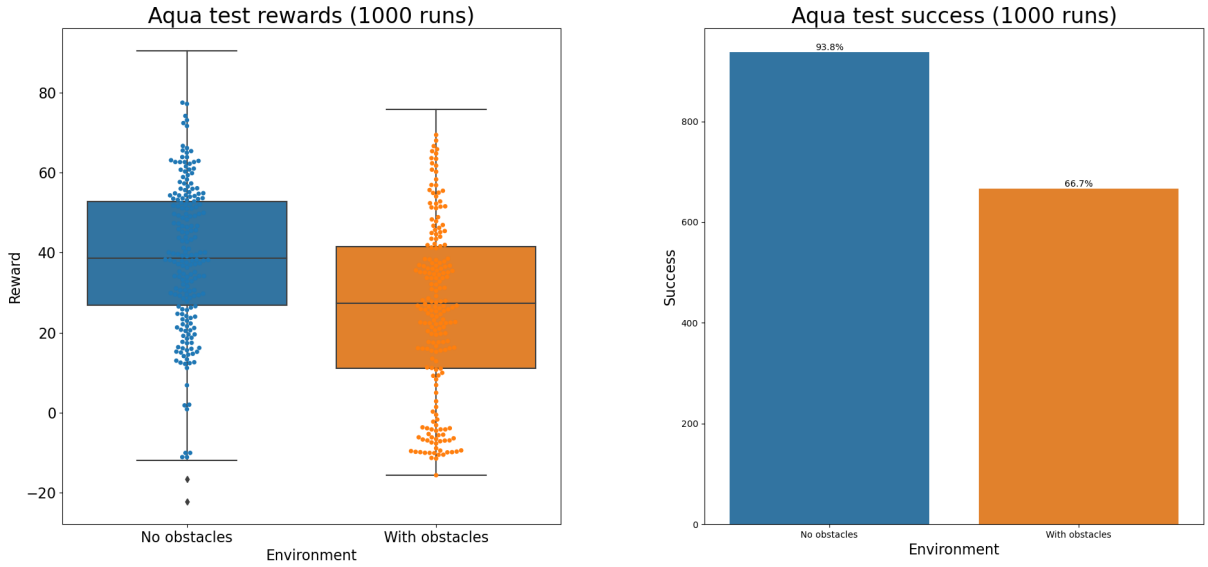
(a) Reward dei training

(b) Successi dei training

Figura 2: Grafici dei risultati ottenuti durante il training. Notiamo l'andamento discendente dopo i primi 4000 episodi, molto probabilmente legato all'apprendimento da parte della policy ad ottenere la stessa reward senza raggiungere lo stato goal.

Notiamo che mentre il reward, raggiunto il plateau, rimane pressoché costante, il successo cala: questo mette in evidenza che la rete ha imparato a sfruttare il sistema di reward e sta ottenendo gli stessi risultati evitando di andare allo stato di goal. In ogni caso, notiamo che tra i 3000 e i 3200 episodi in entrambi i casi di studio troviamo le migliori policy, che ottengono circa 93% per il caso senza ostacoli e circa 66% per l'altro.

Per conferma, utilizziamo il checkpoint 30 della policy per l'ambiente senza ostacoli e il checkpoint 32 per quella con ostacoli; lanciamo 1000 volte i test per entrambi i casi e collezioniamo reward e successi: notiamo che, come predetto, le percentuali di successo sono del 93.8% e del 66.7%.



(a) Distribuzione dei reward dei 1000 test

(b) Successi dei 1000 test

Figura 3: Grafici dei risultati ottenuti nei 1000 test. Notiamo chiaramente la disparità tra i due problemi, che vede quello privo di ostacoli, chiaramente, come il più semplice da risolvere con DQN.

3 Conclusioni

In vista di un miglioramento di questi risultati sarebbe utile studiare più in dettaglio la reward function per ogni test, così da migliorare la persistenza dei successi. Un'ulteriore miglioria sarebbe aumentare la profondità della rete, ma ciò comporterebbe tempi di training molto più lunghi. Infine, può essere di ampio beneficio l'applicazione di un algoritmo di *safe improvement* come *Safe Policy Improvement with Baseline Bootstrapping* (SPIBB), che permette di incrementare il success rate delle policy senza intaccarne mai il comportamento quando esso è già ottimale.