Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

pyTORCS

simulatore di guida per deep reinforcement learning

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Gianluca Galletti

Sessione II
Anno Accademico 2018/2019

Sommario

In questo elaborato è presentato pyTORCS, un ambiente open-soruce per lo sviluppo e il test di metodi per la guida autonoma, in particolare il deep reinforcement learning. pyTORCS è basato su una versione modificata di TORCS, un simulatore di guida open-source che permette guida su una varietà di tracciati e strade urbane e abilita l'uso di un ampio raggio di sensori virtuali. pyTORCS eredita le caratteristiche di TORCS e lo estende con una interfaccia stile OpenAI Gym per Python, fortemente configurabile e modulare, e l'integrazione con Docker per una più semplice configurazione. Verranno mostrate e confrontate le performance di alcuni popolari algoritmi per il reinforcement learning applicati a pyTORCS e affrontate le problematiche che sorgono durante la risuluzione di un ambiente di simulazione di guida.

Elenco delle figure

1 Interazione agente-ambiente . . . 3
2 Alcuni ambienti OpenAI Gym: (da sinistra) Atari Breakout, BipedalWalker, MountainCar, Pendulum, Cartpole, LunarLander . . . 4
3 Alcuni frame di gioco . . . 6
4 Struttura completa semplificata di pyTORCS . . . 9
5 Alcuni sensori in pyTORCS. Da sinistra: trackPos, angle e track . . . 10
6 Esempio di MDP. In verde i nodi stato, in arancione i nodi azione, per ogni transizione sono indicate le probabilità e per alcune transizioni il reward . . . 14
7 Somma dei reward per episodio in un agente soggetto a catastrophic forgetting . . . 18
8 Semplice rappresentazione di un minimo locale e quello globale . . . 18
9 Semplice rappresentazione visualizzazione del problema di moving target . . . 19
10 Q learning (sinistra) confrontato con Actor-critic in DDPG (destra) . . . 20
11 Pseudocodice di DDPG . . . 22
12 Loss di PPO nei casi con advantage positivo o negativo . . . 24
13 Pseudocodice di PPO . . . 24
14 Alcuni dei tracciati su cui è stato eseguito il training . . . 25
15 Tempi e ritorni dell'agente di riferimento . . . 26
16 Reti di DDPG . . . 26
17 Step e ritorno su circa 600 episodi di training con DDPG (g-track-1) . . . 27
18 Training complessivo di DDPG, con cambio di tracciato (g-track-1, aalborg) . . . 28
19 Tempi e ritorni dell'agente DDPG dopo 1.5 milioni di step di training . . . 28
20 Reti di PPO . . . 28
21 Step e ritorno di PPO bloccato su un minimo locale (g-track-1) . . . 29
22 PPO su 700 episodi (400k step) con aggiunta entropy loss (g-track-1) . . . 30
23 Tempi e ritorni dell'agente PPO dopo 3 milioni di step di training . . . 30
24 Ritorni di PPO (blu) e DDPG (arancio) a confronto (g-track-1) . . . 31

Indice

1 Ambiente pyTORCS **4**
1.1 Ambienti per Deep RL . . . 4
1.1.1 Caratteristiche . . . 4
1.1.2 API stardard Gym . . . 4
1.2 TORCS . . . 4
1.3 pyTORCS . . . 5
1.3.1 Motivazione . . . 5
1.3.2 Features . . . 5
1.3.3 Installazione e uso . . . 6
1.3.4 Struttura . . . 7
1.3.5 Spazi di stato e azione . . . 10
1.3.6 Funzione di reward . . . 11
1.3.7 Terminazione episodi . . . 14

2 Il (deep) reinforcement learning **14**
2.1 Reinforcement learning . . . 14
2.2 Terminologia . . . 15
2.3 Sfide e debolezze . . . 17
2.3.1 Dataset e catastrophic forgetting . . . 17
2.3.2 Minimi locali . . . 18
2.3.3 Reward hacking . . . 19
2.3.4 Moving target . . . 19

3 Approcci moderni al Deep RL **19**
3.1 DDPG 19
3.1.1 Actor critic 20
3.1.2 Explore-exploit 20
3.1.3 Azioni continue 20
3.1.4 Loss 21
3.1.5 Implementazione 21
3.2 PPO 21
3.2.1 Idea 21
3.2.2 Policy-Value networks 22
3.2.3 Explore-exploit 22
3.2.4 Funzione obiettivo e loss 23
3.2.5 Implementazione 24

4 Risultati **25**
4.1 Reward e terminazione 25
4.2 Agente di riferimento 26
4.3 DDPG 26
4.4 PPO 28
4.5 Confronto e conclusioni 30

Introduzione

Il deep reinforcement learning è un approccio particolarmente emergente, sia in ricerca che industria. La sua forza sta nella posizione che occupa: rappresenta infatti un punto di contatto tra teoria dei controlli e machine learning, caratteristica che dona al deep reinforcement learning forti potenzialità nella robotica moderna.

Una particolarità del reinforcement learning è la necessità del sistema da controllare. Il reinforcement learning è infatti caratterizzato dall'interazione tra un agente e un ambiente (figura (1)).

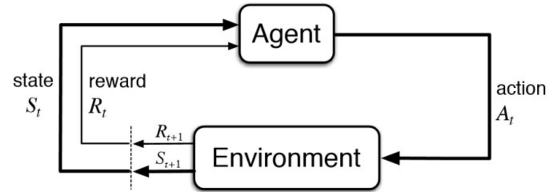


Figura 1: Interazione agente-ambiente

In questo elaborato si presenta un ambiente complesso per il deep reinforcement learning, chiamato pyTORCS. Si percorrono le caratteristiche dell'ambiente, difficoltà incontrate nello sviluppo e le loro soluzioni. Infine si esplora e si sperimenta l'utilizzo in pratica di tale ambiente applicando alcune tecniche di deep reinforcement learning.

L'elaborato è diviso in 4 capitoli. Nel primo capitolo viene introdotto il concetto di ambiente per deep RL e viene descritto pyTORCS, con alcuni dei problemi che sono stati incontrati. Il secondo capitolo è incentrato sul descrivere le forze e debolezze del deep RL, e tutte le difficoltà che insorgono qualora lo si utilizzi per problemi complessi. Il terzo capitolo illustra gli algoritmi di reinforcement learning usati: DDPG e PPO. Infine l'ultimo capitolo è riservato ai risultati ottenuti con ogni approccio, confrontati con un agente di riferimento che non fa uso di tecniche di learning.

1 Ambiente pyTORCS

Per poter sviluppare, testare e utilizzare metodi deep RL è necessario avere il modello completo dell'ambiente da risolvere. Chiamiamo *ambiente* tale modello completo.
Sono stati di conseguenza sviluppati una varietà di modelli appositamente per il deep RL, talvolta ispirate da problemi di controllo ottimo (pendolo, cartpole), altre volte prese da videogiochi (Atari, Starcraft). Alcuni esempi si vedono in figura (2).

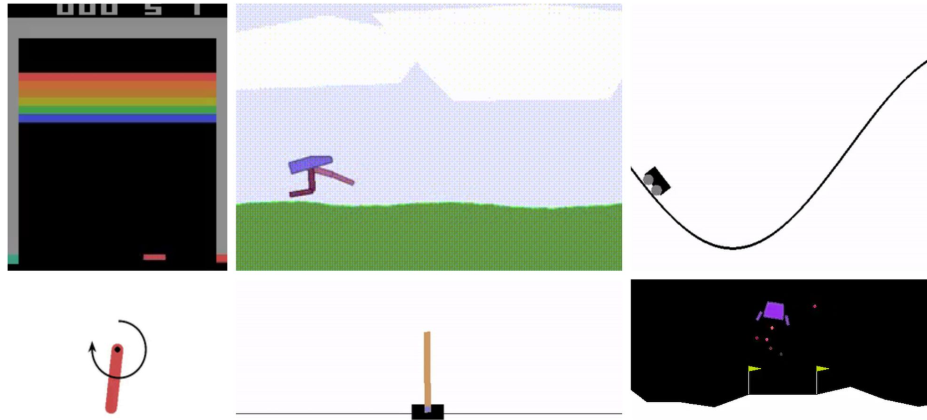


Figura 2: Alcuni ambienti OpenAI Gym: (da sinistra) Atari Breakout, BipedalWalker, MountainCar, Pendulum, Cartpole, LunarLander

1.1 Ambienti per Deep RL

1.1.1 Caratteristiche

Un ambiente per deep RL è genericamente caratterizzato da

- uno spazio di stati (nel deep RL solitamente continuo) – $\mathcal{S}$
- uno spazio di azioni (discreto o continuo) – $\mathcal{A}$
- una funzione di reward – $\mathcal{R}\colon \langle \mathcal{S}, \mathcal{A} \rangle \to \mathbb{R}$
- una funzione di transizione – $\mathcal{T}\colon \langle \mathcal{S}, \mathcal{A} \rangle \to \mathcal{S}$

In più, anche se normalmente è contenuta nella funzione di transizione $\mathcal{T}$ per comodità di implementazione e utilizzo si aggiunge una funzione di terminazione $\mathcal{F}$. $\mathcal{F}$ sarà una funzione booleana che ritorna $\top$ quando uno stato è terminale. Decide quindi quando un episodio è finito.

- una funzione di terminazione – $\mathcal{F}\colon \langle \mathcal{S}, \mathcal{A} \rangle \to \{\top, \bot\}$

1.1.2 API stardard Gym

Buona parte delle simulazioni per deep RL sono realizzate seguendo l'API standard di OpenAI Gym [1], per interfacciarsi verso l'esterno.
Gym prevede due metodi fondamentali:

- ***reset()*** $\to s_0$ – Riporta la simulazione allo stato iniziale.
- ***step(*** u_t ***)*** $\to s_{t+1}, r_{t+1}, d$ – Avanza di un passo la simulazione.

dove s_0 è lo stato iniziale, u_t è l'azione, s_{t+1} è lo stato su cui si transita, r_{t+1} è il reward associato a $\langle s_{t+1}, u_t \rangle$ e d indica se lo stato è terminale.

1.2 TORCS

TORCS [2] sta per *The Open Racing Car Simulator* . Si tratta di un gioco open source di corse. Sviluppato originariamente nel 1997 da Eric Espié and Christophe Guionneau, era pensato come videogioco. Negli anni successivi lo sviluppo è passato a Bernhard Wymann, che ha trasformato TORCS sempre più in una piattaforma per la competizione tra AI e la ricerca in vari settori.

Ad oggi TORCS è attivamente usato per sperimentare e confrontare metodi di motion planning (offline [3] e online), e talvolta anche per tecniche learning-based (generalmente imitation learning o reinforcement learning).

La versione più recente risale al 2016, TORCS 1.3.7 [4]. Incluso nell'ultima versione si ha *scr_server* [5], estensione che permette a agenti esterni di guidare in TORCS, in aggiunta a oltre 40 tracciati e 30 vetture.

1.3 pyTORCS

pyTORCS [6] è un interfaccia a TORCS scritta in Python. Essendo realizzato seguendo l'API Gym permette di utilizzare il videogioco come un qualsiasi ambiente reinforcement learning.

1.3.1 Motivazione

Il recente progresso nel deep RL ha aumentato la richiesta di ambienti complessi, in particolare negli ambiti di robotica (controllo di bracci o robot) e guida autonoma. Testimone è AWS DeepRacer, ambiente realizzato da Amazon con lo scopo di insegnare ad una piccola auto a completare giri attorno a semplici tracciati. Uno dei meriti di DeepRacer è stato quello di rendere tecniche SoA reinforcement learning facilmente accessibili a chiunque.
DeepRacer mostra però alcune debolezze sul lato della versatilità:

- La sua facilità d'uso non si riporta sul codice: visto che il sorgente è molto complesso, cambiamenti significativi sono resi quasi impossibili.
- Inizialmente pensato per essere eseguito sui server AWS. Si può comunque eseguire in locale [7], ma non senza ostacoli.
- L'ambiente è piuttosto resource intensive.

Usare un software completamente open source, d'altra parte, permette di avere pieno controllo dei suoi meccanismi interni. La scelta di TORCS è stata ovvia: il gioco ha un forte supporto e interesse di ricerca, e grazie a *scr_server* è è già predisposto per connettersi a agenti esterni.
TORCS aveva già un interfaccia Python per il reinforcement learning: *gym_torcs* [8].
gym_torcs permetteva di usare TORCS come un ambiente Gym, ma usa una vecchia versione di TORCS e presenta altri problemi significativi:

- la versione usata, 1.3.2, ha un memory leak che impedisce di riavviare la gara molte volte
- l'ambiente si bloccava spesso, a causa di alcuni bug nella comunicazione con *scr_server*.
- non si può usare il PC durante il training. Ciò è dovuto a *xautomation*, un software per automatizzare la tastiera, usato per scegliere la gara dal menù di gioco.
- Il buffer UDP si riempiva di vecchi pacchetti (buffer bloat). Questo faceva si che un agente lento non riuscisse ad avere stati aggiornati dall'ambiente, ma ricevesse stati vecchi anche di alcuni secondi.

pyTORCS mira a risolvere tutti questi problemi, e allo stesso tempo a avvicinarsi alla semplicità verso l'utente di DeepRacer.

1.3.2 Features

- API seguendo lo standard Gym
- TORCS è eseguito all'interno di un container Docker, per facilitare installazione e configurazione
- Il sorgente di TORCS è modificato appositamente per evitare il menù di selezione gara. Non è quindi necessario usare *xautomation*
- Configurazione estensiva tramite file *.yaml* (ad esempio il tracciato o l'auto possono essere cambiati da configurazione)
- Supporta lettura di frame di gioco RGB (visti in figura (3))
- Ultima versione di TORCS (1.3.7)

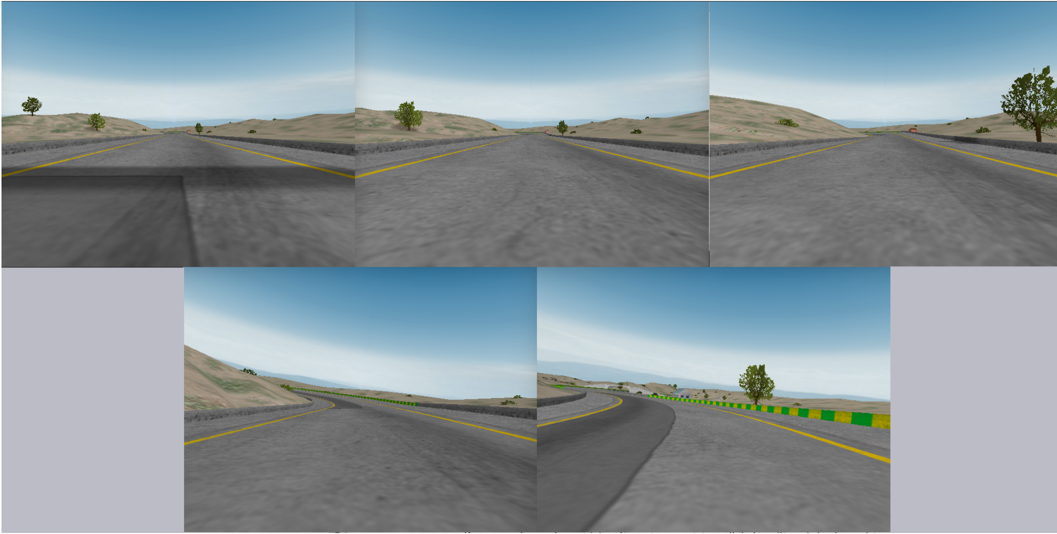


Figura 3: Alcuni frame di gioco

1.3.3 Installazione e uso

pyTORCS è pensato per funzionare su sistemi Linux, idealmente con GPU Nvidia. L'immagine Docker rende possibile l'installazione tra distro diverse.
Per entrambe le installazioni occorre clonare la repository di pyTORCS e cd nella cartella:

```
git clone https://github.com/gerkone/pyTORCS-docker
cd pyTORCS-docker
```

Installazione versione container Per installare pyTORCS usando Docker è necessario avere *docker* e *nvidia-docker2* installati. Di conseguenza anche i driver Nvidia sono necessari.
Dopo averli installati si può clonare l'immagine con:

```
docker pull gerkone/torcs
```

Installare le dipendenze python:

```
python3 -m pip install -r requirements.txt
```

Installazione su host La seguente guida è fatta per Ubuntu 20.04.
Su altre distro occorre usare il rispettivo packet manager, e i nomi delle dipendenze potrebbero variare leggermente. Installare le dipendenze:

```
apt-get install libglib2.0-dev libgl1-mesa-dev libglu1-mesa-dev freeglut3-dev libplib-
    dev libopenal-dev libalut-dev libxi-dev libxmu-dev libxrender-dev libxrandr-dev
    libpng-dev libvorbis-dev
```

cd in TORCS

```
cd  torcs
```

Build di TORCS

```
export CFLAGS="-fPIC"
./configure
make
make install
make datainstall
```

Per trasferire le configurazioni di pyTORCS su TORCS appena installato copiarle da *torcs/configs*. Questo passo è opzionale ma raccomandato

```
cp -R configs/* /usr/local/share/games/torcs
cp -R configs/* /root/.torcs
```

Uso base Per lanciare pyTORCS usare semplicemente:

```
python pytorcs.py
```

Sono inoltre disponibili una serie di argomenti e configurazioni, in base alle preferenze. Ad esempio si può scegliere di eseguire l'ambiente in una finestra di terminale distaccata (--*console <konsole|terminator|...>*), attivare il logging (--*verbose*), oltre che tutte le opzioni disponibili nei file di configurazione.

Usare un algoritmo custom Per testare un proprio agente su pyTORCS ci sono due vie: è possibile usare la *run function* già presente, oppure scriversi la propria *run function*.
Usando la *run function* di base basta cambiare dal file di configurazione *.yaml* i campi

```
algo_name: "<nome della classe dell'algoritmo>"
algo_path: "<path al file contenente la classe>"
```

per poter funzionare però il proprio algoritmo deve seguire una semplice interfaccia: deve essere descritto da una classe, con i metodi

- *__init__*(self, state_dims, action_dims, action_boundaries, hyperparams):
- *get_action*(state, episode_number)
- (optional) *learn*(episode_number)
- (optional) *remember*(state, state_new, action, reward, terminal)
- (optional) *save_models*()

Scrivere la propria *run function* è anche possibile. In questo modo si ha più libertà. Un esempio di *run function* è il seguente:

```python
from torcs_client.torcs_comp import TorcsEnv
def run(...):

  env = TorcsEnv(throttle = False, verbose = verbose, state_filter = sensors)

  agent = ...
  for i in range(N_EPISODES):
      # resets the environment to the initial state
      state = env.reset()
      terminal = False
      score = 0
      while not terminal:
          action = agent.predict()
          # transizione in base all'azione dell'agente
          state_new, reward, terminal = env.step(action)
          # operazioni sull'agente - e.g. salva nel replay buffer, agent.learn(i), ...
          # prossimo stato
          state = new_state
```

Anche in questo caso si deve agire sui file di configurazione *.yaml*, per specificare modulo e path della *run function*

```
mod_name: "<nome della funzione di run>"
run_path: "<path al file contenente la funzione>"
```

1.3.4 Struttura

pyTORCS è assimilabile a un sistema client-server, dove TORCS rappresenta il server e la compatibilità Gym Python il client (figura (4)). La comunicazione avviene via UDP (con timeout) e shared memory: UDP per le azioni e lo stato, shared memory per i frame di gioco. L'idea di isolare l'immagine è quella di non spendere potenza di calcolo per convertire l'immagine in stringa (come per gli altri sensori) e non saturare il canale UDP di stringhe contenenti immagini.
Inoltre, siccome TORCS è eseguito all'interno di un container, dal lato server tutti i messaggi passano attraverso Docker. Le porte sono esposte in entrambe le direzioni da Docker (con il flag *-p 3001:3001/udp* al comando di run). Anche l'accesso a memoria condivisa è fatto verso l'interno di un container (il flag --*ipc=host*).

Client Il lato client è responsabile di comunicare con TORCS ed esporre verso l'esterno le funzioni *step* e *reset*. È diviso in due componenti principali: il client vero e proprio (classe *Client*) e l'ambiente (classe *TorcsEnv*).
Questa divisione permette di delegare a *Client* tutta la comunicazione (interfaccia UDP e memoria condivisa, parser delle stringhe di stato e azione) astraendo buona parte delle operazioni specifiche, lasciando a *TorcsEnv* la gestione dell'API Gym e di parte delle configurazioni.
La classe *Client* ha tre metodi fondamentali per la comunicazione con TORCS:

- ***setup_connection*** – usato per gestire l'init di scr_server. Il client si identifica tramite il sid e invia opzionalmente l'impostazione relativa gli angoli del rangefinder.
- ***get_server_input*** – usato per le comunicazioni in entrata. Gestisce i diversi stati di *scr_server* (*identified*, *restart*, *drive* o *shutdown*) e agisce di conseguenza.
  identified è semplicemente la risposta all'init fatto in *setup_connection*.
  Lo stesso vale per *restart*, corrisponde alla risposta (positiva) alla richiesta di restart.
  In caso di stato *drive* (normale comunicazione) viene letta la stringa di stato in arrivo, parsata e ritornato un dizionario contenente lo stato completo.
  Infine, lo stato *shutdown* è la risposta alla fine della gara: il client chiude la comunicazione e eventualmente termina il container e torcs.

  Per semplicità di astrazione *get_server_input* ritorna sia lo stato da UDP che l'immagine letta e ricostruita da memoria condivisa.
- ***respond_to_server*** – usato per le comunicazioni uscenti. Riceve un azione dall'agente come array, la parsa nel formato di *scr_server* e la invia tramite UDP.

Per quanto riguarda *TorcsEnv* vengono esposti, oltre a *step* e *reset*, anche altri metodi per applicare parte delle configurazioni (come il cambio del tracciato o dell'auto).
Il metodo *step* fa uso dei metodi *get_server_input* e *respond_to_server* per interfacciarsi con TORCS.
Il codice (semplificato) dello step è il seguente:

```
1  def step(action):
2      # ottengo l'ultimo stato
3      obs_curr = client.get_servers_input()
4      # invio l'azione a torcs
5      client.respond_to_server(action)
6      # ricevo il prossimo stato
7      obs_new = client.get_servers_input()
8      # calcolo il reward e controllo se lo stato terminale
9      episode_terminate = terminal(obs_new)
10     reward = reward(obs_new, obs_curr, action)
11
12     return obs_new, reward, episode_terminate
```

La doppia chiamata a *get_server_input* fa si che si riceva sempre lo stato più recente e successivo all'applicazione dell'azione: la prima chiamata svuota il buffer UDP (ridimensionato per contenere una sola stringa di stato), la seconda è bloccante finché non arriva un nuovo stato aggiornato.

Per quano riguarda il reset invece il codice semplificato è il seguente:

```
1  def reset():
2      # informa scr\_server del riavvio
3      reset_torcs()
4      # istanzia il client e attiva la comunicazione con torcs
5      client = Client(...)
6      client.setup_connection()
7      # ottieni l'osservazione iniziale
8      obs = client.get_servers_input()
9
10     return obs
```

TorcsEnv si occupa anche delle operazioni connesse col container TORCS, ovvero avviare il container con le giuste impostazioni, riavviarlo in caso di necessità e terminarlo una volta che la simulazione è completata.

Server TORCS è costruito da moduli isolati che operano tra di loro con strutture interfacce e puntatori a funzioni. Ogni parte del gioco (dal menu al motore di gioco, dal motore di render all'audio) è divisa in moduli separati. I "*robot*" non fanno eccezione: ogni *robot* è un package separato, che espone una struttura dati apposita (*RobotItf*, robot interface).

```
typedef struct RobotItf {
  tfRbNewTrack rbNewTrack;
  tfRbNewRace  rbNewRace;
  tfRbEndRace  rbEndRace;
  tfRbDrive  rbDrive;
  tfRbPitCmd   rbPitCmd;
  tfRbShutdown rbShutdown;
} tRobotItf;
```

Ogni campo di questa struttura rappresenta un puntatore ad una funzione. Il motore di gioco si occuperà poi di chiamare le funzioni.

- RobotItf.rbNewTrack – Per settare il tracciato sul robot
- RobotItf.rbNewRace – Inizio di una gara
- RobotItf.rbEndRace – Fine di una gara
- RobotItf.rbDrive – Chiamata a ogni timestep durante la gara per guidare l'auto.
- RobotItf.rbPitCmd – Operazioni per i pit-stop
- RobotItf.rbShutdown – Chiamata prima di fare l'unload del modulo

rbDrive è sicuramente la funzione più interessante, visto che è quella dedicata a calcolare le azioni. In particolare in *scr_server* si occupa di costruire e inviare la stringa di stato, oltre che leggere e applicare l'azione ricevuta. Tutti i valori di stato sono gestiti da *scr_server* fatta eccezione i frame di gioco. Questi sono infatti gestiti dal game engine e il motore di render. Ogni frame viene poi scritto in un area di memoria condivisa.
L'azione viene passata dal robot al motore di gioco di TORCS, che esegue un'iterazione del modello fisico e aggiorna il motore di render. Il motore di gioco è dotato di clock: è eseguito a frequenza fissa di 50 hz (ogni iterazione dura 20 ms).

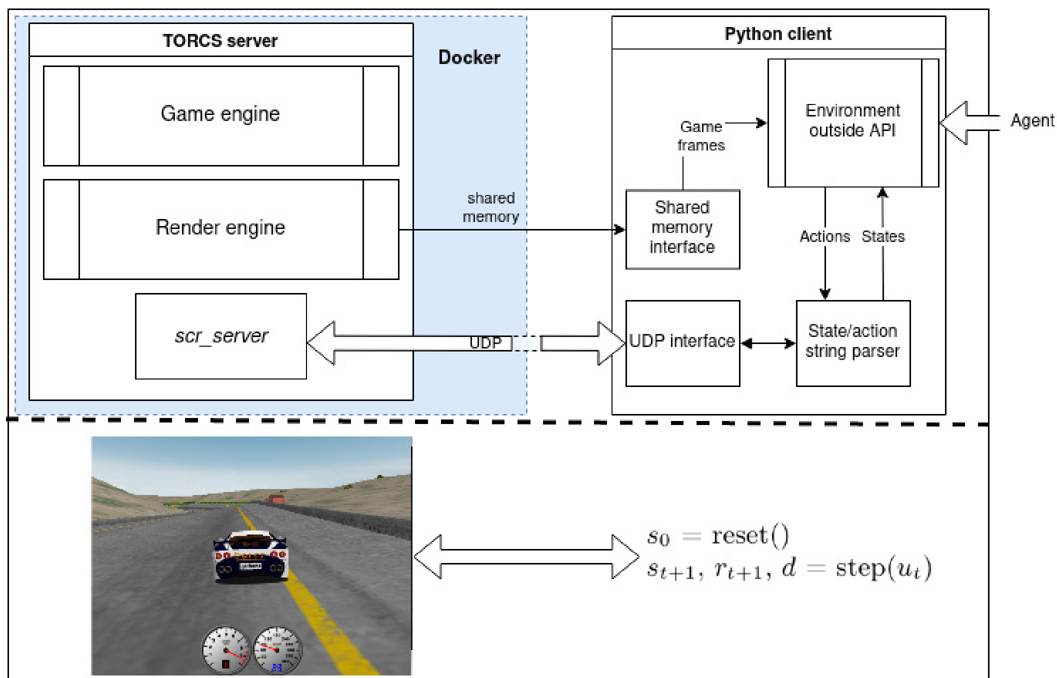


Figura 4: Struttura completa semplificata di pyTORCS

1.3.5 Spazi di stato e azione

Spazio degli stati Dato che lo spazio degli stati è modificabile da file di configurazione nella tabella sottostante è descritto lo stato di default in pyTORCS. Questo stato contiene 29 valori provenienti da diversi "*sensori*" (figura (5)), in aggiunta ai frame di gioco. Ognuno dei valori provenienti dai sensori è calcolato e inviato da *scr_server* utilizzando dati dal modello veicolo, dal motore di gioco e dal tracciato.

Nome Sensore	Range	Descrizione	Notazione
angle	[-π, +π]	Angolo tra l'asse longitudinale all'auto e l'asse tangente al tracciato.	θ
speedX, speedY, speedZ	(-inf, +inf)	Velocità dell'auto lungo gli assi x, y e z.	v_x, v_t, v_z
track	[0, 200]	Vettore di 19 rangefinder, disposti frontalmente sull'auto. Ogni valore rappresenta la distanza dal bordo del tracciato tracciato, incontrato disegnando una retta con diversi angoli rispetto all'asse dell'auto. Limitato a 200 metri.	-
wheelSpinVel	[0, +inf)	Vettore di 4 elementi contenente la velocità di rotazione di ogni ruota.	-
trackPos	[-1, +1]	Distanza tra l'auto e l'asse del tracciato, normalizzato rispetto alla larghezza del tracciato in quel punto.	δ_i
rpm	[0, +inf)	Rotazioni per minuto del motore.	-
totalTime	[0, +inf)	Tempo totale dall'inizio della gara, in secondi.	T_i
damage	[0, +inf)	Danno corrente al veicolo.	dmg
distFromStart	[0, +inf)	Distanza (dell'auto) dalla linea di partenza, lungo l'asse del tracciato.	d
img	[0, 255]	Vettore di 921600 elementi (risoluzione default 640x480 RGB) contenente il frame di gioco corrente. **Disattivato di default.**	-

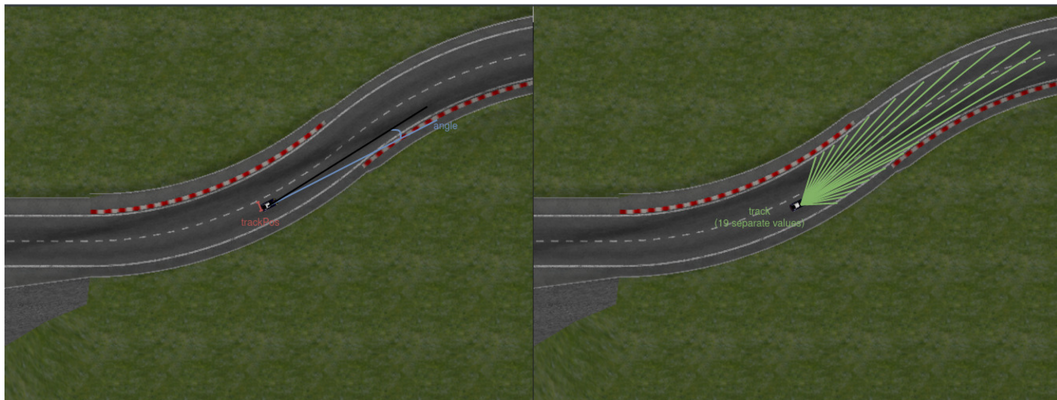


Figura 5: Alcuni sensori in pyTORCS. Da sinistra: trackPos, angle e track

Spazio delle azioni Le azioni che l'agente può dare a pyTORCS sono due: sterzo e richiesta di coppia, come descritto in tabella sottostante. Richiesta di coppia non è altro che acceleratore e freno combinati in un unico valore. Questo è stato fatto per ridurre lo spazio di ricerca dell'agente.

Nome azione	Range	Descrizione	Notazione
steering	[-1, 1]	Valore di sterzo. -1 corrisponde a completamente a destra, 1 completamente a sinistra. L'angolo di sterzo è uguale per ogni vettura: 0.36 rad.	α
richiesta di coppia	[-1, 1]	Acceleratore e freno combinati. Il freno occupa lo spazio [-1, 0). L'acceleratore occupa [0, 1].	τ

1.3.6 Funzione di reward

Una delle sfide teoriche più difficili quando si crea un ambiente per il reinforcement learning è la funzione di reward. Trovare una funzione che riesca a descrivere bene e in modo completo l'obiettivo da raggiungere, senza imporre un "carattere" e rimanendo comunque "comprensibile" alla rete, è spesso estremamente complesso.
Questo passo prende il nome di reward shaping, ed è il motivo per cui recentemente si sta facendo ricerca su tecniche di machine learning che inferiscono il reward da un comportamento.

Mentre in un videogioco Atari si può semplicemente usare il punteggio come reward, in un ambiente come TORCS si ha sia l'imbarazzo della scelta che nessun riferimento immediato sulla qualità della guida: un possibile reward potrebbe essere il tempo sul giro (o sul settore, per evitare la forte sparsità di dover completare un giro prima di ricevere un reward), e mentre potrebbe suonare come una buona idea, in pratica è troppo sparso e complesso per essere applicato in modo affidabile. Necessita inoltre di un tempo di riferimento.
Seguono alcune funzioni di reward, divise per categoria in base a cosa valorizzano. Le prime tre classi riguardano caratteristiche locali, la quarta fa uso di un agente di riferimento per premiare il tempo sui settori, la quarta considera comportamenti cattivi da non seguire mentre l'ultima mette insieme alcune delle funzioni viste in precedenza.

Basati su distanza L'idea dietro a questa categoria di reward sta nel premiare andare lontano. Grazie al Q valore e al numero limitato di step per ogni episodio idealmente l'agente dovrebbe tendere a ottimizzare il modo in cui attraversa il tracciato, evitando anche di uscire, per massimizzare la somma di reward futuri attesi: più velocemente e più distanza attraversa, più alto sarà il reward futuro.

Base $\mathcal{R}_d(s_i, a_i) := d_i$

Progress $\mathcal{R}_{dprog}(s_i, a_i) := d_i - d_{i-1}$

Clipped progress $\mathcal{R}_{dclip}(s_i, a_i) := clip(\mathcal{R}_{dprog}(s_i, a_i))$

Basati sul tracciato Questa categoria sfrutta caratteristiche locali del tracciato per approssimare un buon comportamento. A differenza degli altri reward locali questo tende a imporre un comportamento non ottimale.

Track axis $\mathcal{R}_{ax}(s_i, a_i) := -v_{x,i} \cdot |\delta_i|$

definito *straight* come

$$straight := \begin{cases} \top & \text{se l'auto è attualmente su un rettilineo} \\ \bot & \text{altrimenti} \end{cases}$$

Straights

$$\mathcal{R}_{str}(s_i, a_i) := \begin{cases} 1 & \text{se } |\alpha_i| < threshold \land straight = \top \\ -1 & \text{altrimenti} \end{cases}$$

Basati su velocità

Piuttosto simili ai reward basati su distanza. Sfruttano la definizione di velocità, come derivata della distanza sul tempo. Di conseguenza lo *spostamento* è l'integrale della velocità rispetto al tempo:

$$x = \int_0^n v(t)\ \mathrm{d}t$$

Nel nostro caso particolare, prendendo un episodio con step $\in [0..n]$, l'integrale diventa una sommatoria visto che i passi sono discreti:

$$d_n = \sum_{i=0}^{n} v_{x,i} \cdot \mathrm{d}t$$

Base $\mathcal{R}_s(s_i, a_i) := v_{x,i}$

Acceleration $\mathcal{R}_a(s_i, a_i) := v_{x,i} - v_{x,i-1}$

Un'alternativa è data dall'aggiunta di $cos(\theta_i)$:

$$d_n = \sum_{i=0}^{n} v_{x,i} \cdot cos(\theta_i) \cdot \mathrm{d}t$$

Il motivo della presenza di $cos(\theta_i)$ è che v_x, è longitudinale all'auto. Facendone l'integrale si otterrebbe la distanza percorsa complessiva, anche non verso il completamento del giro.
$v_x \cdot cos(\theta)$ è invece la proiezione della velocità sull'asse del tracciato.

With angle $\mathcal{R}_{s'}(s_i, a_i) := v_{x,i} \cdot cos(\theta_i)$

Punish swirling $\mathcal{R}_{s''}(s_i, a_i) := v_{x,i} \cdot cos(\theta_i) - v_{x,i} \cdot sin(\theta_i)$

Basati su riferimento

I reward basati su riferimento prendono in considerazione il tempo su ogni settore. Per poter fare ciò è necessaria una mappa dei tempi, presa da un agente di riferimento. In questo caso il riferimento viene dal robot avversario base di TORCS: *berniw.* Dopo aver raccolto i tempi per ogni tracciato usando berniw, il reward è strutturato per premiare in base a quanto si è vicini (o oltre) il tempo di *berniw.*
Chiamiamo $\Delta_{ag,c}$ il tempo sul settore e $\Delta_{ref,c}$ il tempo di riferimento sul settore, C_i la distanza dal via del settore più vicino, e definiamo c_i come

$$c_i = \begin{cases} \top & \text{se } d_i > C_i \wedge d_{i-1} < C_i \\ \bot & \text{altrimenti} \end{cases}$$

c_i rappresenta il concetto di aver completato un settore.
La funzione di reward sarà descritta da:

$$\mathcal{R}_{time}(s_i, a_i) := \begin{cases} \Delta_{ref,c} - \Delta_{ag,c} & c_i = \top \\ 0 & \text{se } c_i = \bot \end{cases}$$

In questo modo il reward viene assegnato solo quando si completa uno dei settori.

Basati sul punire comportamenti

Questa categoria di reward è pensata semplicemente per evitare *direttamente* che l'agente prenda delle "cattive abitudini", come ad esempio rimanere fermi o frenare quando la velocità è molto bassa.
Invece di aspettare che l'agente "capisca" che, ad esempio, per percorrere una distanza maggiore deve accelerare e rimanere in pista, questo tipo di reward mira a punirlo se fa il contrario.

Boring speed

$$\mathcal{R}_{bs}(s_i, a_i) := \begin{cases} -1 & \text{se } v_{x,i} < boringspeed \\ 0 & \text{altrimenti} \end{cases}$$

Out-of-track

$$\mathcal{R}_{oot}(s_i, a_i) := \begin{cases} -1 & \text{se } \delta_i > 1 \\ 0 & \text{altrimenti} \end{cases}$$

Breaking still

$$\mathcal{R}_{brake}(s_i, a_i) := \begin{cases} -1 & \text{se } \tau_i < 0 \wedge v_{x,i} < boringspeed \\ 0 & \text{altrimenti} \end{cases}$$

Con *boringspeed* velocità minima (idealmente vicina a 0).

Composite $$\mathcal{R}_{beh}(s_i, a_i) := \mathcal{R}_{oot}(s_i, a_i) + \mathcal{R}_{brake}(s_i, a_i)$$

Wobbly steering

$$\mathcal{R}_{wob}(s_i, a_i) := \begin{cases} -1 & \text{se } |\alpha_i - \alpha_{i-1}| > threshold \\ 0 & \text{altrimenti} \end{cases}$$

Funzioni composte

I reward composti non sono altro che composizioni intercategoria dei reward visti fin'ora. L'idea è quella di otterere il meglio da ogni reward e comporli in uno unico. La maggior parte di questi mettono insieme uno o più dei reward locali con uno di quelli sui comportamenti. Inoltre sono introdotti dei pesi per ogni reward.
Seguono alcuni esempi:

Speed and breaking $$\mathcal{R}_{comp_1}(s_i, a_i) := \mathcal{R}_{s'}(s_i, a_i) + \mathcal{R}_{brake}(s_i, a_i)$$

Distance and breaking $$\mathcal{R}_{comp_2}(s_i, a_i) := \mathcal{R}_{d}(s_i, a_i) + \mathcal{R}_{brake}(s_i, a_i)$$

Speed and behaviour $$\mathcal{R}_{comp_3}(s_i, a_i) := \mathcal{R}_{s'}(s_i, a_i) + \mathcal{R}_{beh}(s_i, a_i)$$

Swirl and behaviour $$\mathcal{R}_{comp_6}(s_i, a_i) := \mathcal{R}_{s''}(s_i, a_i) + \mathcal{R}_{beh}(s_i, a_i)$$

Speed and axis $$\mathcal{R}_{comp_4}(s_i, a_i) := \mathcal{R}_{s'}(s_i, a_i) + \mathcal{R}_{ax}(s_i, a_i)$$

Swirl and axis $$\mathcal{R}_{comp_5}(s_i, a_i) := \mathcal{R}_{s''}(s_i, a_i) + \mathcal{R}_{ax}(s_i, a_i)$$

Forse nessuna di queste funzioni soddisfa tutti i requisiti, ma alcune sono buone approssimazioni, mentre altre non riescono a dare sufficienti informazioni sull'obiettivo all'agente sull'obiettivo.

1.3.7 Terminazione episodi

Nella maggior parte dei contesti di deep reinforcement learning la funzione di terminazione è rappresentabile come una semplice condizione booleana.
In pyTORCS la funzione di terminazione è piuttosto semplice, ma composta da più condizioni:

Out-of-track

$$\mathcal{F}_{oot}(s_i, a_i) := \begin{cases} \top & \text{se } \delta_i \geq 1 \\ \bot & \text{altrimenti} \end{cases}$$

Standing still

$$\mathcal{F}_{speed}(s_i, a_i) := \begin{cases} \top & \text{se } v_{x,i} < boringspeed \wedge i > k \\ \bot & \text{altrimenti} \end{cases}$$

Con *boringspeed* velocità minima e k numero di step di ritardo per l'inizio del controllo

Spun

$$\mathcal{F}_{spun}(s_i, a_i) := \begin{cases} \top & \text{se } cos(\theta_i) \leq 0 \\ \bot & \text{altrimenti} \end{cases}$$

La condizione risultante è semplicemente la disgiunzione delle precedenti. Se soltanto una si verifica l'episodio è da terminare.

$$\mathcal{F}(s_i, a_i) = \mathcal{F}_{oot}(s_i, a_i) \vee \mathcal{F}_{speed}(s_i, a_i) \vee \mathcal{F}_{spun}(s_i, a_i) \tag{1}$$

2 Il (deep) reinforcement learning

2.1 Reinforcement learning

Il reinforcement learning [9] è un settore del machine learning che si occupa di imparare strategie di controllo, interagendo attivamente con un ambiente. In alternativa il reinforcement learning può anche essere visto come una tecnica che permette di imparare (a interagire in) un ambiente basandosi sull'esperienza.

In principio il RL si basava su strutture stocastiche discrete come processi decisionali di Markov (MDP) [10]. Un MDP è definito da uno spazio degli stati $\mathcal{S}$ e uno spazio delle azioni $\mathcal{A}$, una funzione stocastica di transizione $\mathcal{T}$ e una ricompensa per ogni transizione. Ci si può immaginare un MDP come un grafo con stati e azioni come nodi e transizioni probabilistiche e pesate (figura (6)). Gli MDP forni-

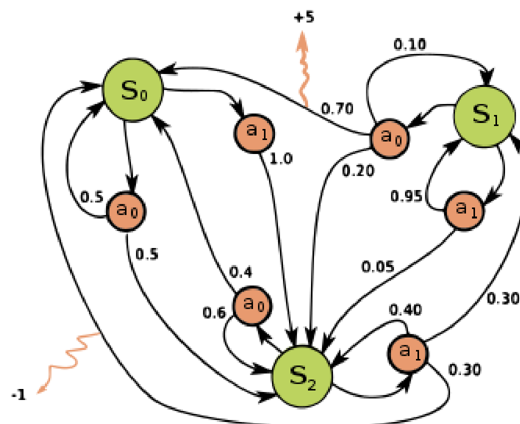


Figura 6: Esempio di MDP. In verde i nodi stato, in arancione i nodi azione, per ogni transizione sono indicate le probabilità e per alcune transizioni il reward

scono un modo per modellare probabilisticamente processi decisionali e si applicano comodamente al

problema del reinforcement learning [11]. Si può infatti usare un MDP per rappresentare direttamente l'ambiente e la policy.

A livello implementativo presentano però un limite: gli MDP reali sono finiti, dispongono quindi di un numero finito di stati. Un problema con spazio degli stati continui non è formalizzabile tramite MDP (se non discretizzando lo spazio e incorrendo in un'altra serie di problemi: curse of dimentionality). Per poter gestire problemi con spazio di stati continuo (o molto grande, come gli scacchi) si è ricorsi al deep learning. Al livello più semplice, invece di usare l'MDP per calcolare Q valore e *state-value* si usa una rete neurale che approssima direttamente le funzioni e la si allena sui Q valori calcolati direttamente usando l'ambiente.
Tuttavia buona parte della teoria del reinforcement learning continua a valere sia con gli MDP che con le reti neurali.
Il reinforcement learning si può dividere in due problemi:

- **Problema di struttura** – capire come interagire con l'ambiente è la sfida principale del reinforcement learning. Un agente deve imparare com'è fatta la struttura dell'ambiente (gli spazi, la sua funzione di transizione, i reward) e codificare quelle informazioni per renderle utilizzabili in un qualche problema di ottimizzazione. Per poter imparare l'ambiente e di conseguenza la policy un agente dovrà essere capace di **esplorare** nuove coppie stato-azione, e di conseguenza trovare nuovi reward.
- **Problema di ottimizzazione** – dopo aver esplorato e raccolto dati rimane la necessità di "capire" come usare le informazioni guadagnate, ed **esploitare** le conoscenze acquisite. Bisogna riuscire quindi a dedurre quali azioni hanno portato ad un certo reward, e modificare la policy a seconda. Questo processo è noto come temporal credit assignment problem (CAP) ([12]). Nel caso generale è difficile formulare un metodo: ad esempio pensando al agli scacchi, dove il reward si ha solo a fine partita, l'ottimizzazione è estremamente ardua e ricca di rumore.

Questi due problemi vengono anche chiamati dilemma explore-exploit, ed è una presenza pratica, oltre che concettuale, nel reinforcement learning: infatti ogni algoritmo per reinforcement learning si occupa nei modi più variegati (ϵ greedy, processi di rumore, policy stocastiche ...) di gestire il bilanciamento tra esplorazione e ottimizzazione sui dati esplorati.

La forza principale del reinforcement learning, ed il motivo per cui sta ricevendo un crescente interesse di ricerca, è che rappresenta un punto di contatto tra due potenti strumenti: teoria del controllo e machine learning. Unisce infatti la robustezza del controllo con la versatilità del machine learning, e nel caso del deep RL, col l'universalità e la forza semantica del deep learning. Tuttavia le potenzialità del deep reinforcement learning non si limitano alla robotica o ai settori in cui si usavano tipicamente tecniche di controllo ottimo, ma si estende in molti campi: dalle AI dei videogiochi a modelli di ottimizzazione sulle reti energetiche o sui treni.

2.2 Terminologia

Spazio di stati

Modello matematico di un dato sistema o ambiente rappresentabile come insieme di tutti gli stati assumibili dall'ambiente.
Lo spazio degli stati può essere discreto, dunque rappresentabile in modo finito tramite una semplice matrice, o continuo. Nel caso di uno spazio continuo la rappresentazione è delegata ad una funzione parziale.
Chiameremo $\mathcal{S}$ lo spazio degli stati, e $s \in \mathcal{S}$ uno stato generico.

Spazio delle azioni

Come lo spazio degli stato ma relativo alle azioni che l'agente può compiere; queste possono essere dettate da limiti fisici di, ad esempio, attuatori e motori, o dalle regole di un gioco, come le mosse disponibili a Go o a scacchi.
Chiameremo $\mathcal{A}$ uno spazio delle azioni, e $u \in \mathcal{A}$ un azione generica.

Transizioni

Funzione (o mappatura) parziale che può essere sia deterministica (videogiochi, simulazioni semplici o a regole fisse) sia stocastica che associa alle coppie stato-azione $\langle s, a \rangle$ lo stato su cui si transita.
Nel caso discreto la si può vedere come la lista di adiacenza di un grafo Markoviano (quindi con una

probabilità associata ad ogni transizione di stato), con gli stati come nodi e le azioni come archi.
Nel caso continuo è invece una distribuzione di probabilità sullo spazio degli stati.

$$\begin{aligned}\mathcal{T}\colon\ \langle\mathcal{S},\mathcal{A}\rangle &\to \mathcal{S}\\ \langle s,u\rangle &\mapsto s'\end{aligned}$$

Reward

Particolare funzione (o mappatura) che associa ad ogni coppia stato-azione $\langle s,a\rangle$ un numero reale, idealmente un premio r. Il reward è solitamente dipendente dal contesto, dovrebbe essere una associazione significativa al fine che si vuole raggiungere (ad esempio negli scacchi potrebbe essere la differenza tra il valore dei propri pezzi e quelli avversari).

$$\begin{aligned}\mathcal{R}\colon\ \langle\mathcal{S},\mathcal{A}\rangle &\to \mathbb{R}\\ \langle s,u\rangle &\mapsto r\end{aligned}$$

Si evince dalla definizione della funzione che il reward va definito ad-hoc per il problema scelto e risulta determinante per la bontà dell'allenamento.

Policy

Funzione (o mappatura) prodotta dall'agente. Ritorna l'azione scelta dall'agente per ogni stato s.
Si può dire che rappresenti il cervello dell'agente autonomo.
La policy può essere deterministica o stocastica, a seconda dell'ambiente su cui deve essere applicata.
Una policy deterministica è solitamente indicata con μ

$$\mu\colon\ \mathcal{S}\to\mathcal{A}$$

Mentre invece una policy stocastica è solitamente indicata con π

$$\begin{gathered}\pi:\mathcal{S}\to[0,1]\\ \pi(\,\cdot\mid s)\approx P[a|s]\end{gathered}$$

Tale nuova azione a, se presa, rappresenta la previsione che punta a massimizzare la somma infinita (eventualmente scontata) dei reward. In un'implementazione reale questa "somma infinita" va adeguatamente approssimata e stimata. Nel deep reinforcement learning le policy sono rappresentate da reti neurali, così come la rappresentazione della somma infinita di reward.

Funzione V

La *funzione state-value*, solitamente descritta con $V^{\pi}(s)$, è una funzione che descrive il "valore" in un dato stato s, secondo la policy π. In altre parole denota la somma (scontata) dei reward, o il ritorno, se si partisse dallo stato s seguendo la policy π.

$$V^{\pi}(s)=\mathop{\mathbb{E}}_{\pi}[G_t|s_t=s]$$

Dove G_t rappresenta la somma scontata di reward a partire dallo step t, che corrisponde allo stato s.
Chiamiamo $V^{*}(s)$ la funzione V ottimale, ovvero il valore che si avrebbe seguendo la policy perfetta

Q valore

La *funzione action-value*, nota come *Q valore* e descritta come $Q^{\pi}(s,a)$, è simile alla *funzione state-value* ma si concentra sulla prossima azione. Rappresenta infatti il ritorno atteso nello stato s, secondo la policy π ma prendendo l'azione a come prossima azione. Si può definire Q in funzione di V come

$$Q^{\pi}(s)=\mathcal{R}(s,a)+\gamma V^{*}(\mathcal{T}(s,a))$$

Dove γ rappresenta il fattore di sconto dei reward.
La relazione tra Q e V è utile per il reinforcement learning, in quanto il $V(s)$ è dato dal prodotto tra il Q valore $Q(s,a)$ e la probabilità di prendere l'azione a

$$V^{\pi}(s)=\sum_{a\in\mathcal{A}}[\pi(a|s)\cdot Q^{\pi}(s,a)]$$

In particolare la *funzione state-value* ha dimensionalità minore del Q valore ed è spesso più facile da approssimare.

Struttura Actor-Critic

Il modello actor-critic (AC) è un modo di strutturare gli algoritmi in fase di training che riduce potenziale errore e rumore nella stima e previsione della somma del Q valore.
In un algoritmo deep reinforcement learning non AC la stima del Q valore è inclusa nella rete agente, che ha già da prevedere la prossima migliore azione. Con AC si dividono questi compiti, lasciando la scelta dell'azione all'Attore, e la stima dei Q valori al Critico.

Replay buffer

Il replay buffer $\mathcal{D}$ è un insieme di tuple (s, a, r, s', t) provenienti dall'esperienza dell'agente durante l'intero training. I vantaggi del replay buffer sono la possibilità di sfruttare meglio l'esperienza, eseguendo più step di training su di essa.

On-policy Vs off-policy

Un algoritmo si distingue tra on e off-policy in base a quale policy viene usata per generare i dati per il training.
Si dice on-policy se la policy che genera dati è la stesa che viene aggiornata (viene raccolto un orizzonte di transizioni, eseguito il training offline su quelle transizioni e infine l'orizzonte è scartato); off-policy significa invece che i dati di training provengono da policy diverse da quella che subisce l'aggiornamento durante il training (i dati usati provengono da esperienza passata e policy precedenti, usando un replay buffer).

Policy gradient

Policy gradient (PG), chiamato anche Policy optimization (PO), è uno dei due approcci principali per rappresentare "cosa imparare" per un agente nel model free deep reinforcement learning.
Questi metodi rappresentano la policy esplicitamente (solitamente π_θ) e mirano ad ottimizzare i parametri θ in modalità "on-policy". Il motivo per cui si chiamano (policy gradient) è che l'ottimizzazione (il gradiente) è applicato direttamente sulla rete che rappresenta la policy.
L'algoritmo più semplice che implementa il policy gradient è vanilla policy gradient (VPG) [13].

Q learning

Q learning non allena esplicitamente la policy ma impara un approssimatore del Q valore $Q_\theta(s, u)$.
La "policy" μ non è altro che l'azione che massimizza il valore dell'approssimatore

$$\mu(s, a) = \arg\max_a Q_\theta(s, a)$$

In altre parole l'azione migliore si sceglie prendendo quella corrispondente al Q valore massimo.
θ si attimizza utilizzando l'equazione di Bellman (o una sua variante), con dati provenienti sia dalla policy corrente che anche da policy di iterazioni precedenti (con la tecnica del replay buffer). Usa quindi la modalità "off-policy".
L'algoritmo più semplice che implementa il Q learning puro è deep Q learning (DQN) [14].

2.3 Sfide e debolezze

Nonostante i numerosi vantaggi del reinforcement learning, ci sono anche molti ostacoli da oltrepassare prima di poter a pieno beneficiare delle sue potenzialità.
Diversi di questi ostacoli sono vere e proprie caratteristiche intrinseche al reinforcement learning, che possono essere soltanto mitigate ma in nessun modo oltrepassate.

2.3.1 Dataset e catastrophic forgetting

Nel deep reinforcement learning, a differenza del supervised deep learning, il dataset non è dato dall'esterno ma prodotto dall'agente, attraverso l'interazione con l'ambiente. Così facendo non abbiamo modo di agire manualmente sui dati e verificarne la validità e la qualità, ma affidiamo questo controllo direttamente alla policy.

Visto che le azioni vengono selezionate usando una policy costantemente variabile, e spesso non ben performante, si finisce per produrre spesso cattivi dati di training. Quando questi dati vengono poi usati per allenare la policy possono portare a un degrado della stessa. Il ciclo rischia di ripetersi con probabilità crescente alla iterazione successiva.

Questo degrado della policy è chiamato catastrophic forgetting, o interferenza catastrofica. Il termine è usato per descrivere la tendenza delle reti neurali a "dimenticare" le informazioni acquisite nel momento in cui si esegue il training su nuovi dati, e si tratta di un problema ben studiato nel deep learning.
Anche se non si tratta quindi di un problema specifico del reinforcement learning, ma del deep learning e delle reti neurali artificiali, il catastrophic forgetting affligge fortemente il deep reinforcement learning dato che le informazioni di training sono sempre variabili. Come si vede dalla figura (7) un agente con catastrophic forgetting riuscirà ad aumentare la somma dei reward, fino ad un certo punto in cui si "dimentica" la policy, e il reward crolla. Si può anche vedere che il ciclo si ripete con un pattern simile.

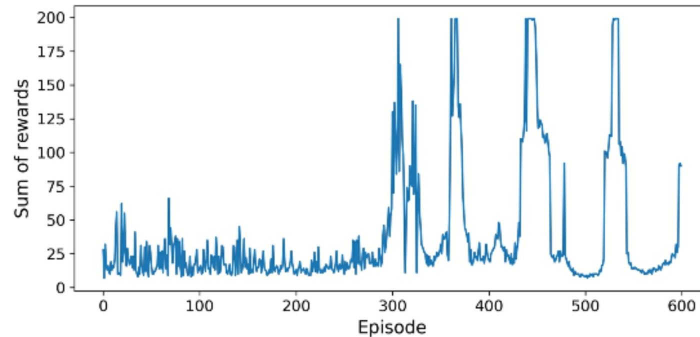


Figura 7: Somma dei reward per episodio in un agente soggetto a catastrophic forgetting

Esistono modi per limitare i danni, come le tecniche trust region (che vedremo in seguito), ma è ancora un problema generalmente non risolto del deep learning.

2.3.2 Minimi locali

Quando una policy si fissa sul ripetere una sequenza di azioni che portano al guadagno di un ritorno sicuro ma non ottimale si ha un minimo locale (figura (8)). Notare che il minimo si riferisce alla funzione di *loss*, non al reward.
Questa debolezza non è isolata al deep reinforcement learning. Anche i metodi supervised ne risentono: quando la discesa del gradiente arriva a un insieme di parametri che riduce l'errore sotto le aspettative rischia di rimanere bloccato. Similarmente nel reinforcement learning un minimo locale si ha quando l'agente trova una sequenza di azioni che porta a un reward maggiore di quello iniziale, ma non ottimale.
Sia si tratti di policy stocastica che deterministica il motivo per cui cade in un minimo locale è la poca o cattiva esplorazione degli spazi di stato, azione e reward. In particolare in una policy deterministica la colpa è da attribuirsi all'algoritmo di esplorazione. Similarmente, in una policy stocastica il colpevole è che l'ottimizzazione sull'agente è stata troppo rapida, facendo diventare la policy troppo deterministica troppo in fretta.

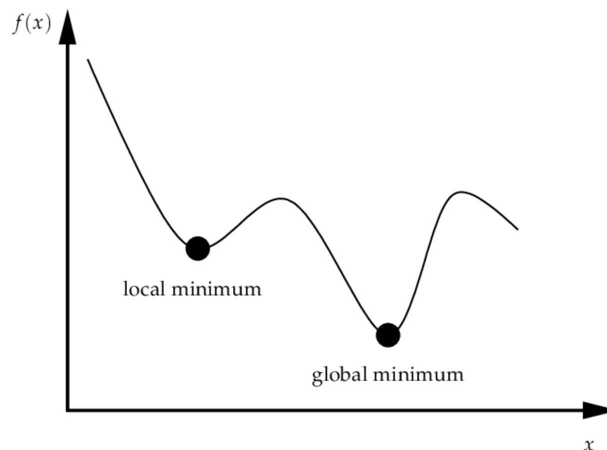


Figura 8: Semplice rappresentazione di un minimo locale e quello globale

Una tecnica generica per evitare i minimi locali è la *entropy loss*: aggiungendo il fattore entropia delle previsioni dell'agente alla funzione di loss si incentiva l'incertezza delle azioni e di conseguenza l'esplorazione.
Quindi, spesso, si possono evitare minimi locali semplicemente con un migliore metodo di esplorazione o dei parametri di tuning corretti.

2.3.3 Reward hacking

In parole povere lo scopo di un agente nel (deep) reinforcement learning è quello di massimizzare il ritorno. Quando però un agente aumenta il proprio ritorno violando lo spirito o le regole dell'ambiente si ha il reward hacking. Quindi l'agente trova metodi inaspettati e tipicamente non accettabili per massimizzare il ritorno. In altre parole il reward cumulato aumenta, ma la performance effettiva dell'agente no.
Ancora una volta questa non è una debolezza del reinforcement learning, bensì dell'ambiente su cui si sta sperimentando, e in particolare della funzione di reward.
Se il processo di reward shaping non è stato adeguato infatti si rischia di lasciare acquisire reward anche seguendo comportamenti inopportuni. Ad esempio se in un gioco di auto si premia lo stare in pista più che l'andare veloce un agente è incentivato a scegliere di non muoversi (non rischiando quindi di uscire di pista).
L'unica soluzione è quello di disegnare il reward accuratamente tentando di evitare ogni scappatoia, anche arrivando al punto di punire con reward negativi i comportamenti che ricadono nell'hacking della funzione di reward.

2.3.4 Moving target

Il problema del moving target si verifica quando un agente cerca di stimare i Q valori basandosi su target che cambia ad ogni step di training (figura (9)).
Questo era un problema dei vecchi approcci di deep reinforcement learning, ma ad oggi è quasi

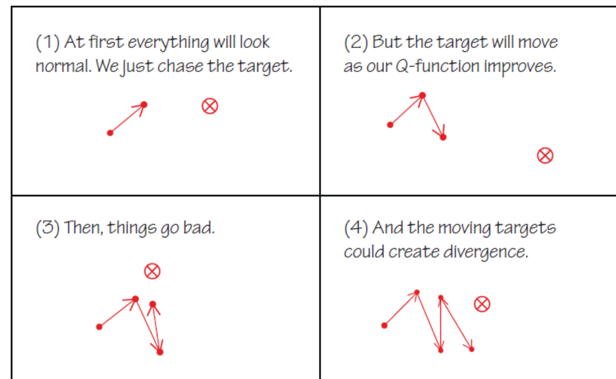


Figura 9: Semplice rappresentazione visualizzazione del problema di moving target

completamente assente grazie all'introduzione delle *reti target*: invece di una singola rete per stimare i Q valori si usano due reti, di cui una rappresenta il target e i cui pesi sono aggiornati periodicamente a partire dalla rete principale.

3 Approcci moderni al Deep RL

Il primo approccio sviluppato per il Deep reinforcement learning è il deep Q learning (DQN). Nella sua versione base DQN è molto semplice, ma soffre gravemente di tutte le debolezze elencate nel capitolo precedente. Le mancanze di DQN, unite con la necessità di risolvere problemi sempre più complessi ha creato la necessità di sviluppare nuovi approcci, spesso ispirati a DQN.
Esaminiamo due degli algoritmi più rivoluzionari nel reinforcement learning degli ultimi anni: DDPG e PPO.

3.1 DDPG

Deep Deterministic Policy Gradient o DDPG è stato proposto per la prima volta nel 2015 da Lillicrap et al. [15]. Essendo pensato specificamente per il controllo continuo promette una discreta stabilità su

spazi di azione continui, contesto in cui altri algoritmi spesso falliscono.
Il nome ci suggerisce che DDPG si tratta di un algoritmo a policy deterministica, ovvero la policy mappa stati a azioni: $\mu(s_i) = a_i$. Una policy deterministica funziona solo se anche la funzione di transizione è deterministica, quindi non c'è incertezza sull'ambiente.

DDPG è basato su Deterministic Policy Gradient (DPG) [16]. Idealmente rappresenta il punto di incontro tra Policy optimization e Q learning; infatti è progettato per imparare contemporaneamente $Q(s, a)$ e una rappresentazione della policy $\mu(s)$. Tuttavia dal punto di vista implementativo è più simile al Q learning, dato che la policy è imparata off-policy (utilizzando un replay buffer).

3.1.1 Actor critic

DDPG segue la struttura Actor-Critic (figura (10)) con reti target. Un singolo agente DDPG è quindi composto da 4 reti separate, attore, critico, attore target e critico target. In particolare in DDPG le reti target subiscono soft update con Polyak averaging. Chiamando le reti θ e θ_{target} e dato un peso di update τ (tendente a 0, solitamente 0.001):

$$\theta_{target} \leftarrow \tau\theta + (1 - \tau)\theta_{target}$$

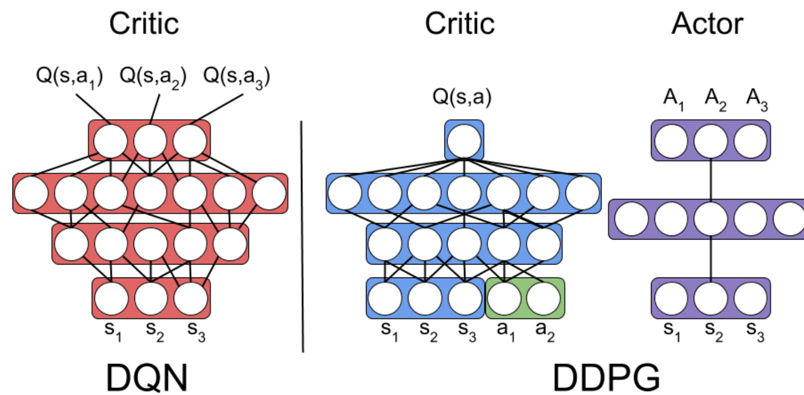


Figura 10: Q learning (sinistra) confrontato con Actor-critic in DDPG (destra)

3.1.2 Explore-exploit

Un altro tratto distintivo di DDPG è nel modo in cui si risolve il dilemma explore-exploit: invece di usare esplorazione casuale fa uso di un processo di rumore, tipicamente un processo *Ornstein–Uhlenbeck*, che si aggiunge all'azione scelta dalla policy. Scegliendo azioni rumorose l'agente è costretto a esplorare nuovi stati.
DDPG ha vari aspetti positivi ma presenta una forte debolezza: è estremamente fragile rispetto ai parametri di training (detti iperparametri). Questo fa si che cattive configurazioni di parametri portino a inconsistenze nel training.

3.1.3 Azioni continue

Conoscere la policy μ è necessario per semplificare il problema di ottimizzazione:
Dato che lo spazio delle azioni è continuo $Q(s, a)$ è sempre derivabile rispetto ad a. Questo ci permette di approssimare

$$Q(s, a) \approx \arg\max_a Q(s, \mu(s)) \tag{2}$$

E quindi utilizzare la discesa del gradiente direttamente sulla policy μ (parte "*Policy gradient*" del nome).
In compenso l'assunzione sulla derivabilità di $Q(s, a)$ rispetto ad a significa che DDPG non può funzionare su spazi di azioni discreti.

3.1.4 Loss

Per decidere la funzione di loss si è partiti dall'equazione di Bellman.
Definiamo $Q^*(s,a)$ come funzione Q ottimale. Q^* è descritta con l'equazione di Bellman

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P(\cdot|s,a)} \left[\mathcal{R}(s,a) + \gamma \max_{a'} Q^*(s',a') \right]$$

Dove $\mathcal{R}$ è la funzione di reward e γ è il fattore di discount.
Supponendo che $Q_\phi(s,a)$ sia approssimato con una rete con parametri ϕ, e dato $\mathcal{D}$ un insieme di transizioni nella forma (s,a,r,s',t) (stato, azione, reward, nuovo stato e se il nuovo stato è terminale), assimilabile al replay buffer.
Si può usare *mean-squared Bellman error* (MSBE) come loss: rappresenta la distanza tra Q_ϕ e la soluzione della funzione di Bellman.

$$L(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',t) \sim \mathcal{D}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-t) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right] \tag{3}$$

Dove $(1-t) = 1$ se lo stato s' non è terminale.
Mettendo insieme la loss vista in (3), le reti target e l'approssimazione vista in (2), si ottiene la loss di DDPG.
Supponendo che la policy $\mu_\theta(s)$ sia approssimato con una rete con parametri θ, e che θ_{target} e ϕ_{target} siano le reti target di policy e funzione Q

$$L(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',t) \sim \mathcal{D}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-t) Q_{\phi_{target}}(s', \mu_{\theta_{target}}(s')) \right) \right)^2 \right] \tag{4}$$

La loss descritta in (4) è usata con la *gradient descent* per ottimizzare il critico.
Per quando riguarda l'attore per definizione vogliamo massimizzare la funzione Q. Di conseguenza si usa *gradient ascent* sulla funzione $Q_\phi(s,a)$, ovvero sull'output del critico

$$L_{act}(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{s \sim \mathcal{D}} \left[Q_\phi(s, \mu_\theta(s)) \right]$$

3.1.5 Implementazione

Segue lo pseudocodice di DDPG (figura (11)). Come si può notare il gradiente di attore e critico sono scritti nella forma

$$\nabla \frac{1}{|B|} \sum f(x)$$

Significa semplicemente che si sta calcolando f (che sarà la rispettiva loss) media su tutta la *batch*.

3.2 PPO

Proximal Policy optimization o PPO è una classe di algoritmi per deep reinforcement learning spesso considerati lo stato dell'arte. Proposto nel 2017 dal team di OpenAI, è pensato specialmente per risolvere i problemi del reinforcement learning e combattere le difficoltà sorte con gli algoritmi precedenti, pur mantenendo semplicità di implementazione e versatilità [17].
A differenza di DDPG non è rilegato a funzionare con spazi di azione continui, ma funziona sia con azioni categoriche che ritornando una distribuzione Gaussiana con la probabilità dell'azione (rendendo quindi possibile controllo su spazi continui).
Inoltre, PPO in generale funziona con policy stocastiche. Questo significa che la policy prodotta da PPO sarà nella forma $\pi(\mathcal{A}|\mathcal{S}) = P[a_i|s_i]$.

3.2.1 Idea

In concetto PPO è basato su un altro algoritmo: TRPO (*Trust-Region Policy Optimization*) [18].
Entrambi cercano di risolvere il problema base del reinforcement learning di produrre autonomamente il dataset e dover, ad ogni step di training, fare il miglior passo possibile verso la policy ottimale senza

Algorithm 1 Deep Deterministic Policy Gradient

1: Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: **repeat**
4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** however many updates **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

13: Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s, a) - y(r, s', d))^2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

15: Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

16: **end for**
17: **end if**
18: **until** convergence

Figura 11: Pseudocodice di DDPG

però mai peggiorarla.
Esistono due varianti principali di PPO: PPO-Penalty and PPO-Clip, in base al modo in cui viene mantenuto il vincolo di non peggiorare la policy aggiornata. PPO-Clip è la versione più comunemente usata, nonché quella descritta in seguito.

L'idea che sta dietro a PPO-Clip è quella di restringere (clip) gli aggiornamenti della policy per evitare che la nuova policy si allontani troppo dalla vecchia. Questo è fatto limitando la funzione obiettivo ad un intorno fidato.
Tutti gl algoritmi nella classe PPO sono on-policy.

3.2.2 Policy-Value networks

Anche PPO segue la struttura Actor-Critic. Attore e critico però sono qui rinominati come reti Policy e Value, per convenzione. Inoltre vengono mantenute le reti risalenti allo step di training precedenti ("vecchie"). Questo è fatto per poter.
Il risultato è che un agente PPO, come DDPG, è formato da 4 reti distinte. A differenza di DDPG però l'aggiornamento delle reti "vecchie" non è soft, ma i parametri vengono rimpiazzati ad ogni step.

3.2.3 Explore-exploit

In PPO l'esplorazione è una naturale conseguenza della policy stocastica. Il livello di casualità in ogni azione dipende sia dalle condizioni iniziali dei parametri della rete, che dagli iperparametri e la funzione obiettivo. Man mano che il training avanza la policy tende a diventare sempre meno casuale: gli aggiornamenti incoraggiano a minimizzare l'entropia, e quindi exploitare reward già noti.

3.2.4 Funzione obiettivo e loss

Il punto di partenza delle funzioni obiettivo di PPO sono le tipiche Policy Gradient loss [19].
Per quanto riguarda il critico la loss è ovvia: in VPG e PPO il critico è usato per approssimare la funzione state-value; la loss non è altro che la differenza tra il ritorno effettivo e la previsione (target) del critico:

$$L_{PG_c}(\phi_{t+1}) = L_{PPO_c}(\phi t + 1) = \mathop{\mathbb{E}}_{t}\left[\sum_{i=0}^{t} \gamma^{i-1} r_i - V_\phi(s_i)\right]$$

La loss dell'attore è più complicata. Partendo dalla loss attore di VPG

$$L_{PG_a}(\theta_{t+1}) = \mathop{\mathbb{E}}_{t}\left[log\left(\pi_\theta(a_t|s_t)\hat{A}^{\pi_\theta}(s,a)\right)\right] \tag{5}$$

Dove θ_k sono i parametri della policy corrente, θ_{k-1} della vecchia policy e $\hat{A}^{\pi_\theta}(s,a)$ rappresenta l'advantage, funzione che descrive il reward in più che può essere accumulato prendendo l'azione a. Si rappresenta come la differenza tra la somma di reward futuri scontati prendendo l'azione a ($Q(s_t, a_t)$) e la somma scontata di reward attesi (baseline, $V(s)$).

$$\hat{A}^{\pi_\theta}(s,a) = Q(s,a) - V(s)$$

Il concetto dietro a (5) è intuitivo: visto il segno della loss è dato dall'advantage $\hat{A}^{\pi_\theta}(s,a)$ il gradiente sarà positivo quando l'advantage è positivo, ovvero quando si sta calcolando la loss su quelle azioni che portano ad un aumento del reward atteso. Applicando il gradiente si "andrà verso" queste azioni, aumentandone la probabilità. Vice versa con advantage negativo, si diminuisce la probabilità delle azioni che portano ad un calo di reward atteso. La debolezza della PG loss è che si affida alla stima dell'advantage, che diventa sempre meno accurata man mano che si applica il gradiente (quindi che la policy si allontana troppo dall'originale).
TRPO di fatto cerca di risolvere questo problema, tramite una funzione obiettivo modificata e un vincolo KL. In particolare il vincolo serve a limitare il cambiamento della policy.

$$L_{TRPO_a}(\theta_k) = \mathop{\mathbb{E}}_{t}\left[\frac{\pi_{\theta_k}(a_t|s_t)}{\pi_{\theta_{k-1}}(a_t|s_t)}\hat{A}^{\pi_\theta}(s_t,a_t)\right] \tag{6}$$

Con vincolo

$$\overline{KL}_{\pi_{\theta_k}}(\pi_{\theta_{k+1}}) \leqslant \delta \tag{7}$$

TRPO può anche essere visto come un problema di ottimizzazione, descritto da:

$$\begin{aligned} \max_{\theta} \quad & \sum_{i=1}^{N} \frac{\pi_{\theta_k}(a_i|s_i)}{\pi_{\theta_{k-1}}(a_i|s_i)}\hat{A}^{\pi_\theta}(s_i,a_i) \\ \text{s.t.} \quad & \overline{KL}_{\pi_{\theta_k}}(\pi_{\theta_{k+1}}) \leqslant \delta \end{aligned}$$

Notare come il valore atteso sia qui calcolato non su una distribuzione (com'era in DDPG) ma con sample provenienti direttamente dalla policy.
Una pecca di TRPO è la presenza del vincolo KL (7), che rende il problema di ottimizzazione più lento. PPO incorpora il vincolo (7) con nella funzione obiettivo, semplificando il problema.
La funzione obiettivo di PPO integra policy gradient con TRPO, ovvero (5) con (6) e (7).
Riscriviamo il rapporto tra nuova e vecchia policy come r_t

$$r_t(\theta_k) = \frac{\pi_{\theta_k}(a_t|s_t)}{\pi_{\theta_{k-1}}(a_t|s_t)}$$

Questo $r_t(\theta)$ ci dice quanto più probabile è un azione nella nuova policy rispetto alla vecchia. Se $r_t(\theta) > 1$ allora a_t è più probabile in π_{θ_k}; altrimenti, se $r_t(\theta) < 1$ allora a_t è più probabile in $\pi_{\theta_{k-1}}$. La

loss di PPO-clip è quindi scritta come

$$L_{PPO_a}(\theta_k) = \mathop{\mathbb{E}}_{t}\left[min\left(r_t(\theta)\hat{A}^{\pi_\theta}(s_t, a_t),\ clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_\theta}(s_t, a_t)\right)\right] \tag{8}$$

La loss è quindi il minimo di due termini:

- $r(\theta)\hat{A}^{\pi_\theta}$ non è altro che la loss di policy gradient (5)
- $clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_\theta}$ applica un clip alla loss di policy gradient, con ϵ vicino a 0 (≈ 0.2).

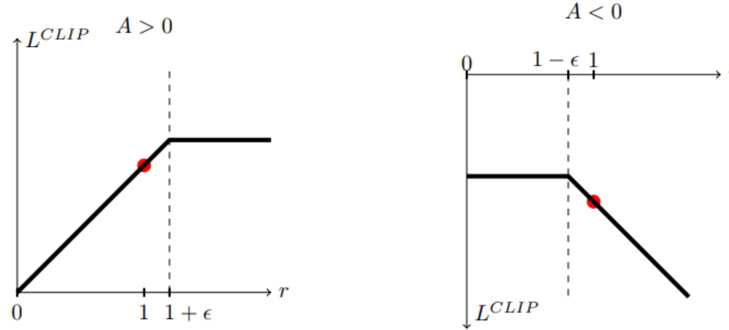


Figura 12: Loss di PPO nei casi con advantage positivo o negativo

Come si vede dalla figura (8) quando l'advantage è positivo (sinistra) la loss aumenta fino ad appiattirsi quando le azioni sono molto più probabili (ovvero $r_t(\theta) > 1$) nella nuova policy, evitare di discostarsi troppo dalla precedente. Quando l'advantage è negativo (destra) si fa lo stesso: se l'azione è molto meno probabile (ovvero $r_t(\theta) < 1$) evitiamo di renderla impossibile appiattendo la loss. In più, nella parte destra del grafico con advantage negativo si vede anche che viene penalizzata una policy in cui si è resa più probabile un azione cattiva.

3.2.5 Implementazione

Segue lo pseudocodice per l'implementazione di PPO

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

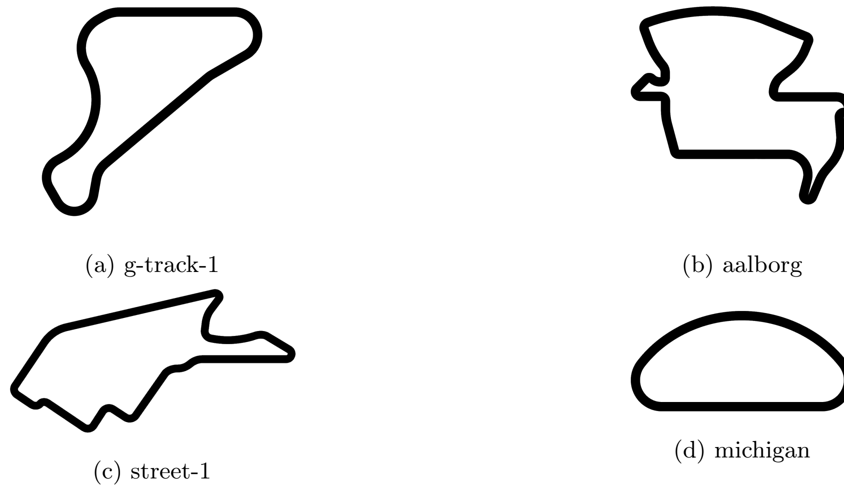
typically via some gradient descent algorithm.
8: **end for**

Figura 13: Pseudocodice di PPO

4 Risultati

Seguono alcuni risultati illustrati del training su DDPG e PPO, oltre che un semplice agente di riferimento.

Questi esperimenti mirano a verificare l'usabilità dell'ambiente pyTORCS in un ambito più realistico, applicando due degli algoritmi più usati e noti. Si tenta anche di verificare l'efficacia di tali algoritmi in un contesto complesso come la guida, per poi compararne alcuni dei risultati.
Il training è stato svolto su diversi tracciati (figura 14). La motivazione dietro il cambio di tracciato



(a) g-track-1 (b) aalborg (c) street-1 (d) michigan

Figura 14: Alcuni dei tracciati su cui è stato eseguito il training

durante il training è l'intenzione di migliorare l'esplorazione e la variabilità sui dati raccolti, per poter evitare minimi locali e overfitting: alcuni tracciati richiedono stili di guida differenti da altri, e in questo modo l'agente è spinto a non imparare un tracciato ma imparare a guidare.

4.1 Reward e terminazione

Reward Per tutti i risultati a seguire è stata usata una versione pesata (w_1, w_2, w_3, w_4 pesi) della funzione composte *Swirl and behaviour*, $\mathcal{R}_{comp_6}$

$$\mathcal{R}_{comp_6}(s_i, a_i) := v_{x,i} \cdot cos(\theta_i) \cdot w_1 - v_{x,i} \cdot sin(\theta_i) \cdot w_2 + \mathcal{R}_{oot}(s_i, a_i) \cdot w_3 + \mathcal{R}_{brake}(s_i, a_i) \cdot w_4$$

Sperimentando tra diverse funzioni si è osservato che quelle basate sulla velocità si comportavano meglio tra tutte. Tuttavia a volte, a causa di "cattive" inizializzazioni dei parametri dell'agente, molti episodi iniziali erano spesi frenando da fermi. Questo è il motivo per cui è stata aggiunta la punizione sul comportamento.

Riguardo alle altre funzioni proposte solo quelle *basate sulla distanza* riuscivano a produrre una policy con una discreta performance, anche se più lentamente di quelle basate su velocità. Questo può essere dovuto ad un vantaggio nascosto delle funzioni basate su velocità: la velocità è controllata quasi direttamente dall'agente, tramite acceleratore e freno. Questo rende potenzialmente più semplice le funzioni da approssimare, *state-value* e *Q*.
Le funzioni *basate su comportamenti* prese singolarmente non informano abbastanza la policy e c'è reward hacking: ad esempio con $R_b rake$ l'auto sta ferma senza frenare, con $R_o ot$ l'auto sta ferma non rischiando di uscire.
Quelle *basate su riferimento* sono difficili da calibrare e l'agente spesso non riesce a guadagnare alcun reward, e soprattutto richiedono dati di un cosiddetto "esperto", ovvero un agente esterno affidabile da cui prendere il riferimento. É possibile che con un ulteriore tuning il reward basato su riferimento potrebbe ottenere risultati migliori.

Terminazione La funzione di terminazione degli episodi stata usata è la disgiunzione di $\mathcal{F}_{oot}$, $\mathcal{F}_{speed}$ e $\mathcal{F}_{spun}$, come descritto in (1).

4.2 Agente di riferimento

L'agente di riferimento usato non fa uso di metodi di learning, ma si tratta semplicemente di un controllo proporzionale basato sulla distanza dal centro, l'angolo con l'asse del tracciato e la distanza frontale del rangefinder. Il risultato è un semplice agente che mantiene la linea centrale, frena nelle curve e accelera fino ad una velocità di setpoint.

Tutti i risultati riportati sono su un giro completo, con partenza da fermi. Nel caso in cui l'agente non riesca a completarlo il reward indicato è soltanto sulla porzione fatta.

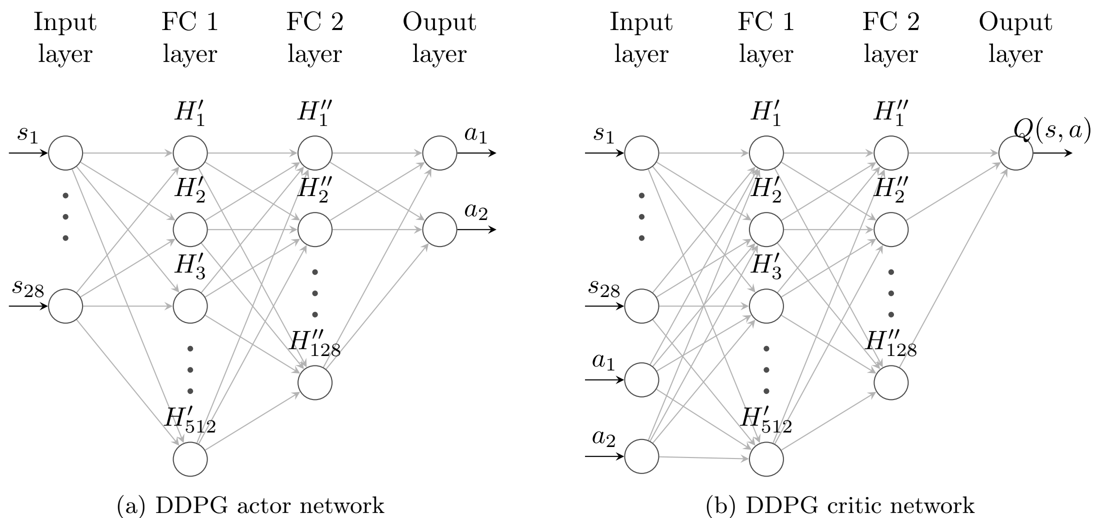
La seguente tabella rappresenta reward cumulato ed eventuale tempo sul giro dell'agente di riferimento.

Tracciato	Reward	Tempo	% completamento
aalborg	734.40	02:07.209	100%
g-track-1	589.07	01:32.148	100%
street-1	1089.10	02:37.238	100%
michigan	671.14	01:15.022	100%

Figura 15: Tempi e ritorni dell'agente di riferimento

4.3 DDPG

Come già visto nella sezione (3.1), DDPG è un algoritmo actor-critic. Le reti usate sono formate da input di dimensione 28 per l'attore e 30 per il critico, due strati nascosti rispettivamente di dimensione 512 e 128, uguali tra attore e critico, e un output di dimensione 2 per l'attore (le due azioni, sterzo e acceleratore/freno) e 1 per il critico (il Q valore stimato). Le attivazioni usate sono $ReLU$ negli strati nascosti e $tanh$ nell'output, come da paper. Nelle figure (20a) e (16b) si vede la struttura semplice di attore e critico.



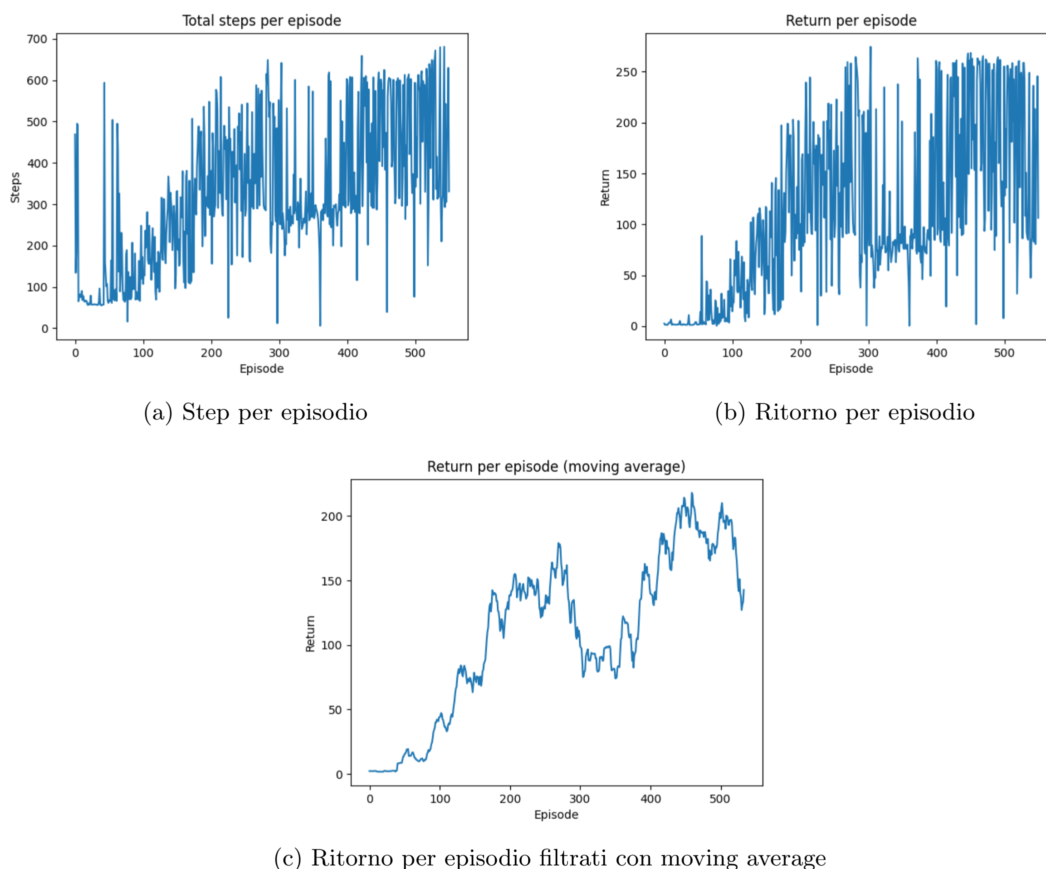
(a) DDPG actor network

(b) DDPG critic network

Figura 16: Reti di DDPG

Il training di DDPG è stato eseguito su 2 tracciati (g-track-1 e aalborg). Gli iperparametri usati sono nella tabella (4.3).
Nei grafici (17) si vedono i risultati del training su g-track-1 con 400000 step totali di training distribuiti su quasi 600 episodi.
La scostanza in step e ritorno nei primi due grafici è dovuto semplicemente al processo di rumore che consente l'esplorazione alla policy stocastica di DDPG. Come si può vedere dal grafico dei ritorni con media pesata la policy è effettivamente riuscita ad aumentare il reward.

Parametro	Valore	Descrizione
LR_{actor}	$5 \cdot 10^{-5}$	Learning rate della rete attore
LR_{critic}	$5 \cdot 10^{-4}$	Learning rate della rete critico
batch size	50	Dimensione della batch di training
buffer size	10^5	Dimensione (in step) del replay buffer
unit sizes	[512, 128]	Dimensioni dei layer di attore e critico
τ	0.005	Fattore di mixing per le reti target
γ	0.97	Fattore di discount del reward



(a) Step per episodio

(b) Ritorno per episodio

(c) Ritorno per episodio filtrati con moving average

Figura 17: Step e ritorno su circa 600 episodi di training con DDPG (g-track-1)

Nel grafico (17c) i vede chiaramente che attorno ai 300 episodi il reward è crollato. Questa è una caratteristica negativa di DDPG, visto che a differenza di PPO o altri metodi trust-region non si ha controllo sulla qualità di uno step di training. La conseguenza è che a volte si può avere la policy che perde performance e degrada, e nel peggiore dei casi catastrophic forgetting.
Il grafico (18) mostra il ritorno durante il training complessivo di DDPG su 600000 step complessivi. Come si può vedere, nel momento del cambio tracciato tra g-track-1 e aalborg il ritorno crolla, per poi tornare ad aumentare qualche episodio dopo.
Ciò è dovuto semplicemente alla differenza tra g-track-1 e aalborg: mentre g-track-1 è un tracciato veloce con poco bisogno di frenare, aalborg ha un tornante dopo il primo rettilineo. Apparentemente l'agente, dopo una fase iniziale in cui non riusciva a svolgere la prima curva, ha dovuto imparare l'importanza di frenare.
Il grafico (18) è una dimostrazione pratica della scelta di cambiare tracciato: l'agente è in questo modo riuscito a esplorare azioni che altrimenti non avrebbe potuto conoscere.

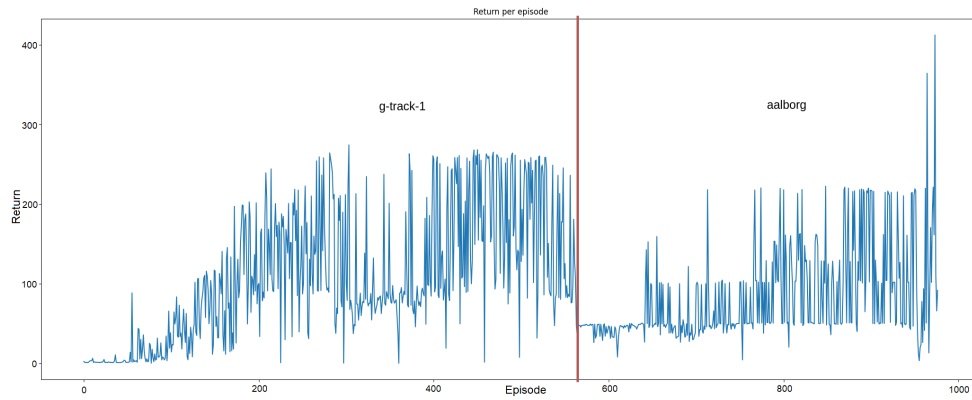


Figura 18: Training complessivo di DDPG, con cambio di tracciato (g-track-1, aalborg)

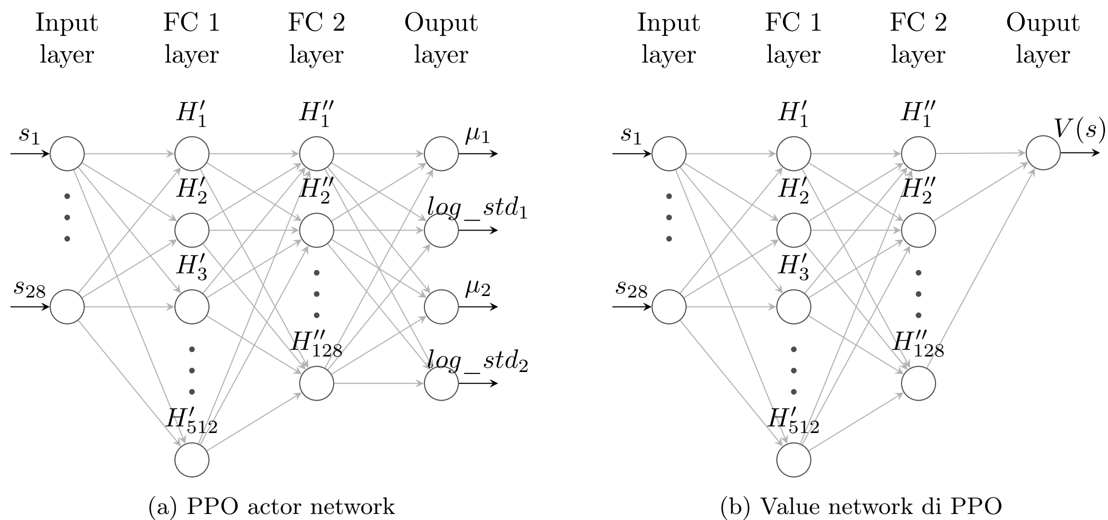
DDPG riesce a completare un giro solo in michigan, dopo un training di 1.5 milioni di step. Ciò non significa però che non abbia imparato una policy che gli permetta di guidare, ma soltanto che la guida è ancora troppo imprecisa. Notare che i risultati nella tabella (**??**) sono utilizzando direttamente la policy deterministica, senza aggiungere il rumore che altrimenti invaliderebbe i risultati.

Tracciato	Reward	Tempo	% completamento
aalborg	374.20	N/A	53%
g-track-1	270.42	N/A	47%
street-1	208.25	N/A	19%
michigan	598.13	00:54.022	100%

Figura 19: Tempi e ritorni dell'agente DDPG dopo 1.5 milioni di step di training

4.4 PPO

PPO usa una rete attore e una critico, chiamata in questo caso anche valore visto che invece di approssimare $Q(s, a)$ approssima $V(s)$. L'attore di PPO è simile a quello di DDPG (figura (20a)), con l'importante differenza che non ritorna direttamente le azioni ma i parametri della distribuzione da cui vanno selezionate, ovvero il centro o media (μ) e il logaritmo della deviazione standard (log_std).



(a) PPO actor network

(b) Value network di PPO

Figura 20: Reti di PPO

L'implementazione di PPO usata inizialmente includeva tutte le caratteristiche precedentemente descritte con alcuni accorgimenti pratici, come l'utilizzo della tecnica Generalized Advantage Estimator (GAE) [20] per il calcolo degli advantage. Gli iperparametri usati si trovano nella tabella (4.4).

Parametro	Valore	Descrizione
LR_{actor}	$5 \cdot 10^{-5}$	Learning rate della rete attore
LR_{critic}	$5 \cdot 10^{-4}$	Learning rate della rete valore
batch size	50	Dimensione della batch di training
horizon	2000	Dimensione del buffer di training
epochs	12	Numero di epoche di training sul buffer
unit sizes	[512, 128]	Dimensioni dei layer di attore e critico
c_1	0.8	Fattore moltiplicativo della PPO loss
c_2	0.005	Fattore moltiplicativo della entropy loss
ϵ	0.1	Dimensionalità dell'intorno della trust region
λ	0.997	Fattore di tuning dell'advantage (GAE)
γ	0.97	Fattore di discount del reward

Come per DDPG, il training di PPO è stato eseguito su 2 tracciati (g-track-1 e street-1).
Inizialmente PPO ha avuto problemi di esplorazione. Come si vede dai grafici (21) durante un training su oltre 2500 episodi, o 1 milione di step, il ritorno è rimasto bloccato attorno a 160. L'agente qui infatti esce di strada alla prima curva dove occorre frenare.

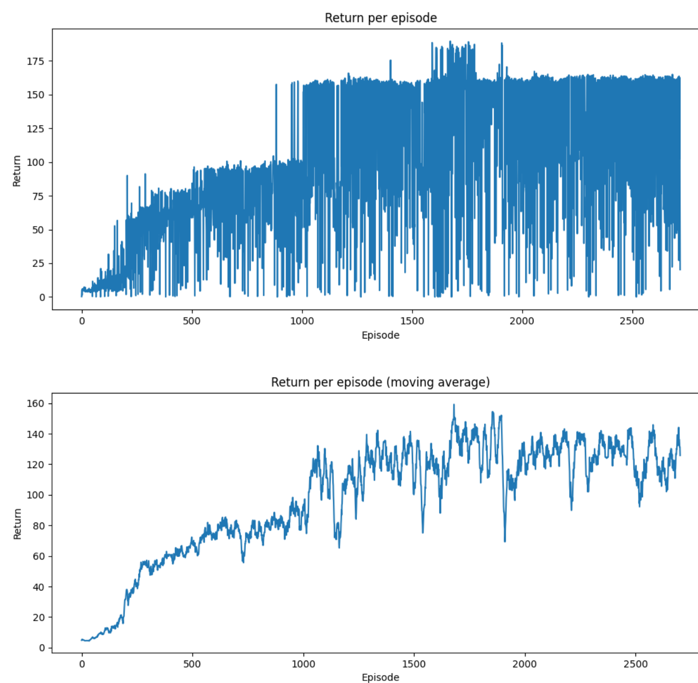


Figura 21: Step e ritorno di PPO bloccato su un minimo locale (g-track-1)

L'unico colpevole di questo comportamento è una cattiva esplorazione. Visto che in PPO l'esplorazione è fatta sfruttando la stocasticità della policy per poter avere una buona esplorazione occorre mantenere alta l'entropia nei parametri della rete attore.

Introducendo la entropy loss sull'attore si può incentivare l'entropia sulla rete. Riprendendo (8) la loss attore di PPO diventa quindi

$$L_{PPO_a}(\theta_k) = \mathop{\mathbb{E}}_t \left[c_1 \cdot min\left(r_t(\theta)\hat{A}^{\pi_\theta}(s_t, a_t),\ clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_\theta}(s_t, a_t)\right) - c_2 \cdot H(\pi_\theta(s_t, a_t))\right]$$

Dove $H(\pi_\theta(s_t, a_t))$ è l'entropia H della distribuzione di probabilità sulle azioni ritornata dall'attore. I grafici (**??**) mostrano il training dopo l'aggiunta di tale loss, che valorizza l'entropia nella rete attore. Si può notare come i risultati siano notevolmente migliorati.

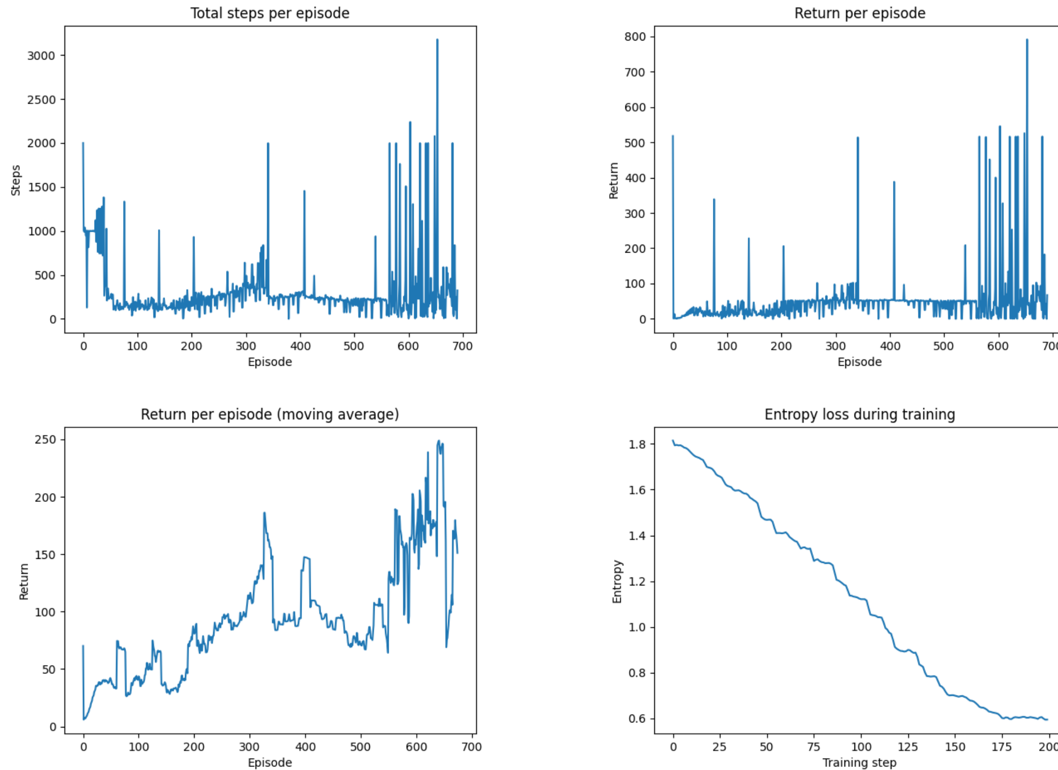


Figura 22: PPO su 700 episodi (400k step) con aggiunta entropy loss (g-track-1)

Anche qui come in DDPG i dati sono rumorosi a causa della selezione stocastica delle azioni che portano a terminazione prematura di alcuni episodi.

Tracciato	Reward	Tempo	% completamento
aalborg	103.50	N/A	15%
g-track-1	572.94	01:06.340	100%
street-1	842.70	N/A	89%
michigan	659.29	01:00.042	100%

Figura 23: Tempi e ritorni dell'agente PPO dopo 3 milioni di step di training

Anche PPO non riesce a completare un giro in tutti i tracciati scelti, dopo 3 milioni di step di training. I risultati nella tabella (23) sono ottenuti scegliendo le azioni basandosi sulla media della distribuzione ritornata dall'attore.

4.5 Confronto e conclusioni

Rispetto all'agente di riferimento entrambi gli agenti deep reinforcement learning sono nettamente più instabili: mentre il semplice controllo proporzionale riesce a completare pressoché ogni circuito, seppur lentamente, i due agente deep reinforcement learning faticano in tracciati con rettilinei seguiti

da tornanti. Questa differente performance può essere causa di un insufficiente training.
PPO generalmente si comporta meglio e completa giri più velocemente di DDPG; tuttavia ha un difetto importante: è molto più lento durante il training, e richiede molti più episodi per imparare una policy competitiva. Come si vede dal grafico (24) sui primi 500 episodi DDPG accumula un ritorno molto maggiore di PPO.

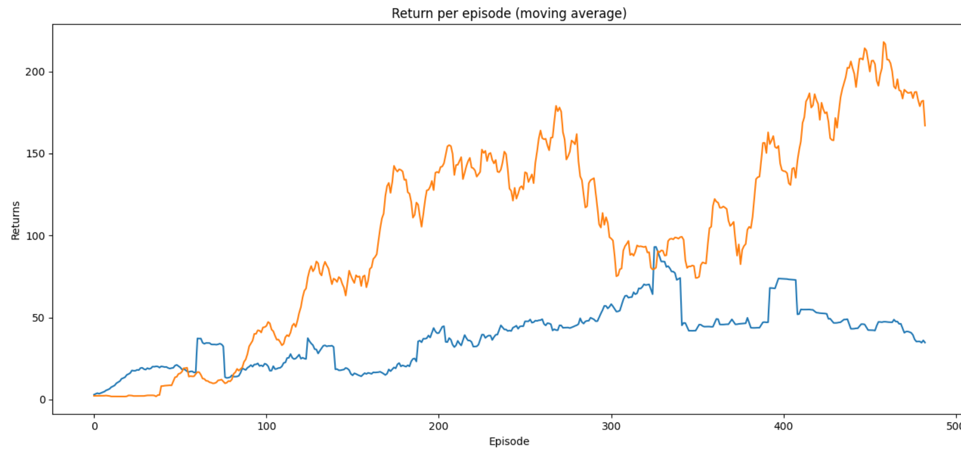


Figura 24: Ritorni di PPO (blu) e DDPG (arancio) a confronto (g-track-1)

Questa inefficienza sul training è però prevista in PPO, essendo un algoritmo on-policy e non usando dati da esperienza passata.
Sul lungo termine infatti PPO supera il ritorno di DDPG, e dimostra anche una maggiore stabilità durante il training, soffrendo raramente di cali di performance notevoli. Durante il training si notano alcune differenze nell'approccio dei due algoritmi di deep reinforcement learning al problema:

DDPG è molto più aggressivo, e tende a sbagliare molto di più all'inizio. Da un episodio all'altro cambia notevolmente il comportamento. Appare spesso che l'agente riesca a capire una buona policy da un episodio all'altro, per poi "dimenticarsi" pochi episodi dopo.
PPO al contrario procede più metodicamente, e non ci sono mai svolte improvvise nella policy. A volte la policy peggiora, ma solo di poco. Questo è grazie al clipping della loss.

Questa differenza nel "carattere" degli algoritmi sta soprattutto in come è strutturata la loss. PPO è infatti disegnato per non soffrire per tutti i problemi che affliggono il Q learning, tra cui DDPG, ma il compromesso è la perdita di efficienza sui sample di training.

Bibliografia

[1] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[2] E. Espié et al. "TORCS, The Open Racing Car Simulator". In: 2005.

[3] Luigi Cardamone et al. "Searching for the Optimal Racing Line Using Genetic Algorithms". In: set. 2010, pp. 388–394. DOI: 10.1109/ITW.2010.5593330.

[4] *patched version of torcs1.3.7 including the scr-server and a patch to send the game image to another application*. 2017. URL: https://github.com/fmirus/torcs-1.3.7.

[5] Daniele Loiacono, Luigi Cardamone e Pier Luca Lanzi. "Simulated Car Racing Championship: Competition Software Manual". In: *CoRR* abs/1304.1672 (2013). arXiv: 1304.1672. URL: http://arxiv.org/abs/1304.1672.

[6] *pytorcs + Docker - Docker-based, OpenAI Gym-like torcs environment with vision.* 2021. URL: https://github.com/gerkone/pyTORCS-docker.

[7] *DeepRacer local training (2020 version)*. 2020. URL: https://github.com/mattcamp/deepracer-local.

[8] *gym_torcs*. 2021. URL: https://github.com/ugo-nama-kun/gym_torcs.

[9] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[10] Richard Bellman. "A Markovian decision process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. URL: http://www.jstor.org/stable/24900506.

[11] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Vol. 4. Gen. 2010. DOI: 10.2200/S00268ED1V01Y201005AIM009.

[12] Marvin Minsky. "Steps toward Artificial Intelligence". In: *Proceedings of the IRE* 49.1 (1961), pp. 8–30. DOI: 10.1109/JRPROC.1961.287775.

[13] Sham Kakade. "A Natural Policy Gradient". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. NIPS'01. Vancouver, British Columbia, Canada: MIT Press, 2001, pp. 1531–1538.

[14] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[15] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[16] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on Machine Learning*. A cura di Eric P. Xing e Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. URL: https://proceedings.mlr.press/v32/silver14.html.

[17] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[18] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[19] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, 1999, pp. 1057–1063.

[20] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG].