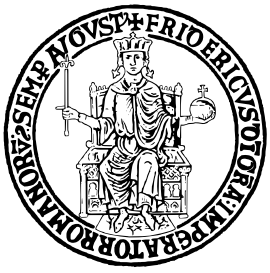
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Intelligent Robotics

Guida autonoma tramite Reinforcement Learning in ambiente Unity-MLAgents

Docente
Prof. Alberto Finzi

Studenti
Salvatore Davide Amodio
N97000369

Monteiro Del Prete
N97000370

ANNO ACCADEMICO 2021–2022

Pagina intenzionalmente lasciata in bianco.

Indice

Introduzione

Il seguente elaborato tecnico mostra uno studio svolto sulla guida autonoma ottenuta tramite algoritmi di reinforcement learning in ambiente Unity-MLAgents. Il task è non competitivo in quanto le macchine hanno come obiettivo il completamento della pista senza tener conto delle performance degli altri agenti.

La prima sezione descrive lo scenario e l'ambiente di sviluppo per la risoluzione del task, entrando nel dettaglio del circuito realizzato per il training.

La seconda sezione presenta l'agente da un punto di vista del modello, oltre che a descriverne il controllo e la sensoristica necessari per un movimento corretto.

La terza sezione è dedicata all'apprendimento nel framework del reinforcement learning, dunque evidenzia le varie fasi di osservazione-azione-reward e di come queste siano riportate nell'ambiente Unity. È esposta, inoltre, una sotto sezione più teorica sul Deep Reinforcement Learning e in particolare sui due algoritmi adoperati, PPO e SAC.

La quarta sezione porta avanti il paragone tra i due algoritmi valutandoli su determinate statistiche e sulla capacità di generalizzazione.

Scenario ed ambiente

Per poter congegnare un agente in grado di imparare a muoversi autonomamente nello spazio e, più nello specifico, riuscire a destreggiarsi adeguatamente nelle basilari dinamiche stradali è stato fondamentale creare un ambiente in cui il robot potesse apprendere al meglio il movimento. Trattandosi di un task di apprendimento automatico è stato opportuno progettare due scenari distinti che fossero dedicati alle fasi di training e di testing. La scena che verrà approfondita in questa sezione è la `TrainScene`; nonostante ciò, come si vedrà nella sezione dedicata alla valutazione del progetto, la `TestScene` differirà solo nella forma e nelle posizioni degli oggetti, ma la logica e la funzionalità insite negli ambienti sono le stesse.

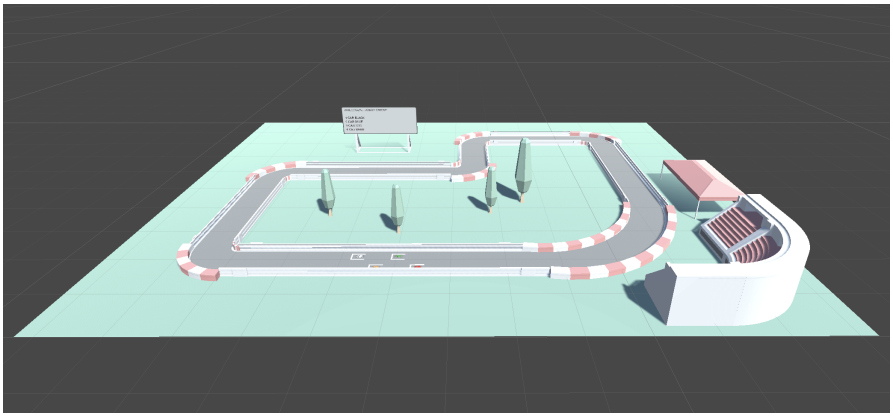


Figura 1: Circuito di training

Tramite il Racing Kit [GR], un tool dedicato prettamente alla progettazione di piste da corsa in Unity, è stato progettato un circuito congeniale ai requisiti del task da affrontare. Il circuito presenta un totale di sei curve, di cui cinque orientate a destra ed una nell'altro verso. La pista è nel complesso abbastanza lineare e presenta nel rettilineo più lungo una griglia di partenza. Non essendo un ambiente competitivo, la griglia è solo un artificio utile a dettare un po' d'ordine; lo spawn delle macchine (posizione locale e rotazione) ad ogni episodio viene così effettuato con criterio posizionando le vetture sulla griglia. Lo scopo è rendere autonomo l'agente, di modo che riesca a completare la pista evitando le varie collisioni. Osservando la conformazione della pista e la figura (2) si può dedurre facilmente come i muri bianchi posti ai margini della pista siano i principali oggetti con cui l'agente non dovrebbe

collidere. Si vorrebbe infatti che alla fine dell'apprendimento la vettura riuscisse a mantenere una traiettoria piuttosto centrale, quanto più equidistante dai muri.

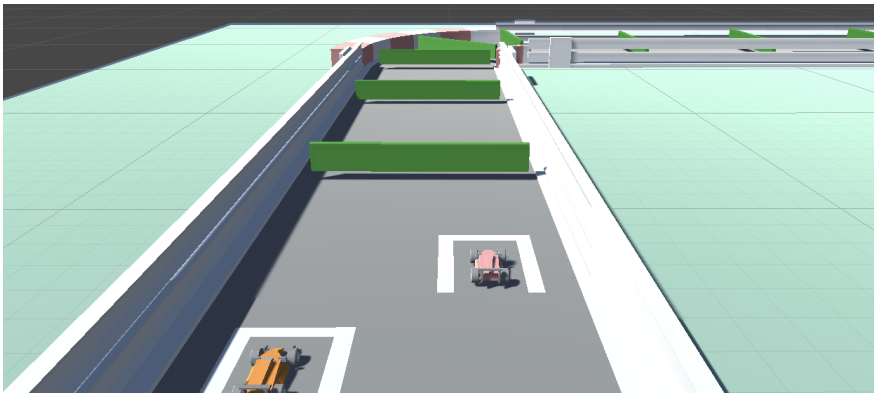


Figura 2: Checkpoint

La pista è popolata anche da altre vetture, per cui è bene che l'agente riesca a non urtarle mentre completa il proprio giro. I muri verdi penzolanti lungo tutto il tracciato non sono altro che checkpoint di natura non solida; sono utili all'agente in fase di learning poiché, come si vedrà nelle sezioni successive, ogni qual volta vengano oltrepassati, forniscono feedback che permettono all'agente di orientarsi.

Descrizione dell'agente

L'agente scelto per il task di guida autonoma è il modello di macchina appartenente al Racing Kit, la cui modularità ha permesso di lavorare su meccaniche di movimento più complesse.

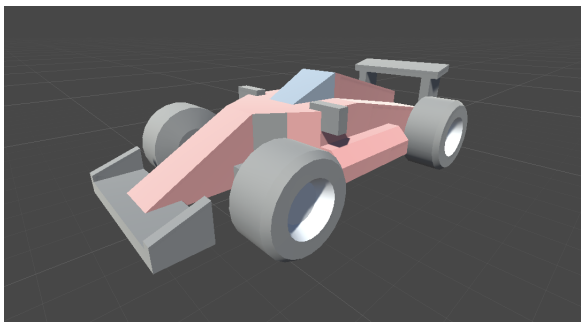


Figura 3: Modello 3D della macchina

Strutturalmente la macchina è composta da un corpo e quattro ruote singole indipendenti.

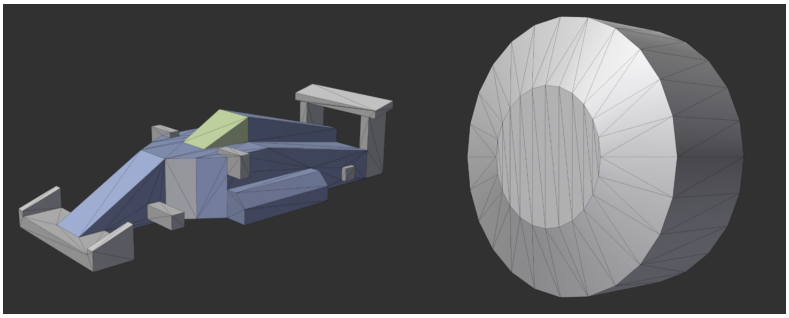


Figura 4: Componenti modello della macchina.

Le masse dei vari componenti sono state impostate in modo da rispettare i valori reali e le esigenze fisiche del movimento. In particolare, la massa del corpo è pari a 750 kg, mentre quella di una singola ruota è pari a 20 kg. Infine, seguendo l'inspector del modello è possibile impostare una trazione integrale, anteriore o posteriore, oltre che a stabilire quali ruote siano dedicate alla sterzata.

Controllo

Il controllo dell'agente è basato sull'applicazione di forze fisiche sulle ruote motrici, piuttosto che sull'intero corpo. La seconda alternativa avrebbe comportato un apprendimento più efficace, tuttavia il movimento sarebbe stato meno naturale.

Affinché il moto fosse possibile, è stato necessario lavorare con i collisori (`Collider`). Questi conferiscono una forma agli oggetti all'interno della scena (`GameObject`) ai fini delle collisioni fisiche. Il collisore non si adatta alla forma effettiva del componente a cui è associato, bensì ne è un'approssimazione. Collisori sono stati creati sia per le ruote, così da rilevare il contatto con la pista sottostante, sia per il corpo della macchina, in modo da individuare scontri con altri elementi della scena (e.g. altre macchine, muri etc.).

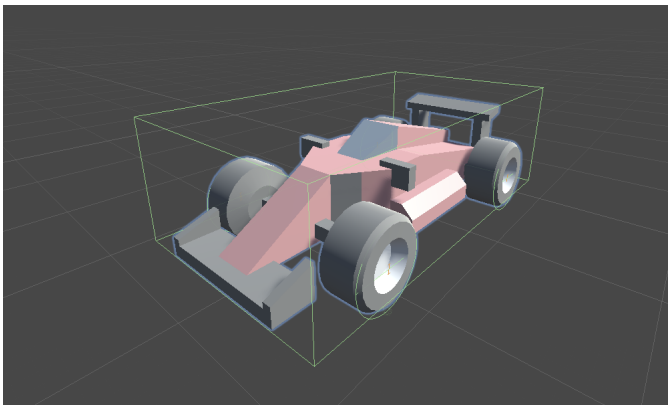


Figura 5: Collider applicati al corpo e alle ruote.

Un'ultima aggiunta al modello è l'elemento `Rigidbody` che conferisce una simulazione fisica all'oggetto di interesse, dunque rendendolo capace di subire la gravità e di reagire alle collisioni.

Il movimento viene definito tramite due valori nello spazio continuo, uno per rappresentare l'input verticale f, dunque andare avanti o indietro, l'altro per indicare l'input orizzontale t, ovvero per sterzare a destra o a sinistra. Le due quantità continue sono state ottenute come segue:

$$m = m_{max} \cdot f \cdot 5 \qquad \text{(motor)} \qquad (1)$$

$$s = s_{max} \cdot t \qquad \text{(steering)} \qquad (2)$$

dove m_{max} e s_{max} sono rispettivamente i valori massimi di avanzamento (o retromarcia) e di sterzata; f e t sono valori nell'intervallo $[-1, +1]$. Entrambe le forze sono state poi fornite alle ruote motrici.

A livello implementativo l'intera logica è presente nello script `CarDriver.cs` agganciato poi all'oggetto macchina. La funzione principale è una funzione evento di Unity nominata `FixedUpdate()` e sincronizzata con il Physics Engine dell'ambiente di sviluppo: viene chiamata ad intervalli regolari indipendentemente dalla sequenza di fotogrammi (FPS). Dunque, l'aspetto fisico del movimento della macchina è implementato tramite tale funzione. Questa viene invocata ogni 0.02 secondi, ovvero una velocità di 50 fotogrammi al secondo. L'ambiente gira in un intervallo di 180-260 FPS, dunque la funzione viene eseguita circa 4 volte per ogni frame.

All'interno della funzione l'idea è iterare su tutte le ruote del modello e, a seconda della natura della ruota, applicarvi la forza di avanzamento (Eq. 1) o di rotazione (Eq. 2). In aggiunta è stato definito un valore di frenata così che la macchina potesse fermarsi in assenza di input.

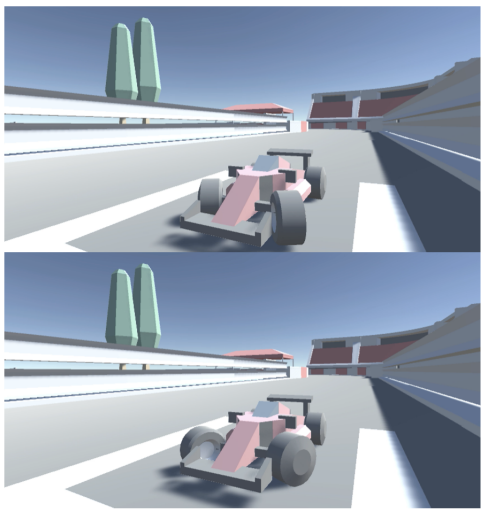


Figura 6: Rotazione delle ruote anteriori.

È importante far notare che la reale rotazione avviene applicando le forze sui collisori; le ruote sono solo una componente visiva e seguono ognuna il proprio collisore associato. Fissando la macchina a trazione anteriore si è in tal modo ottenuta una composizione vicina a quella dello sterzo di Ackermann.

Sensoristica

La sensoristica adoperata è stata fondamentale per l'ottenimento di un controllo corretto che permettesse alla macchina di muoversi senza scontrare muri o altri agenti all'interno della pista. Essa è basata sul Raycasting, ovvero una tecnica utilizzata per emettere raggi in un ambiente 3D e valutare le collisioni risultanti.

Tramite il `RayPerceptionSensorComponent3D` si definisce un punto di origine ed un orientamento dei raggi, con la possibilità di configurare la quantità di raggi per ogni direzione, l'angolazione e l'offset verticale del punto di origine. Tali sensori rappresentano osservazioni aggiuntive collezionate dall'agente durante l'apprendimento, dunque è stato necessario stabilire un compromesso tra il numero di osservazioni e la complessità dell'apprendimento stesso.

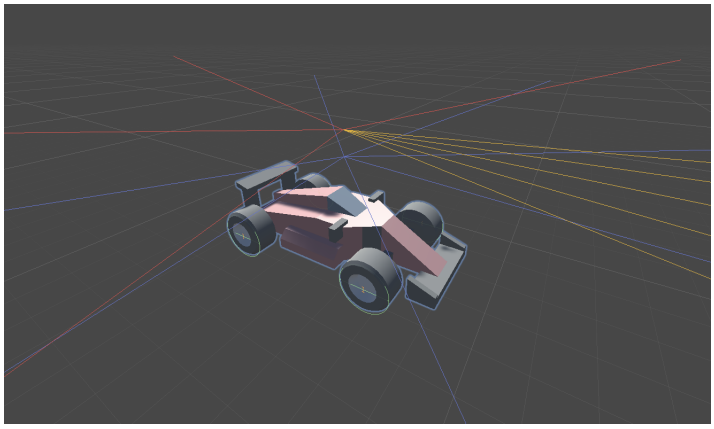


Figura 7: Sensori raycast.

Com'è possibile notare dalla figura (7), vi sono tre sensori (tre punti di origine) identificati da una diversa colorazione per facilitarne la visione. I sensori rosso e giallo, aventi il medesimo punto di origine, servono ad individuare i checkpoint necessari per indicare all'agente la direzione verso cui muoversi; il sensore blu è utilizzato per individuare muri e altri agenti nella scena. L'insieme delle osservazioni è calcolato tramite la seguente formula:

$$(observationStacks) \cdot (1 + 2 \cdot raysPerDirection) \cdot (numDetectableTags + 2).$$

Dunque, dal sensore blu si ottengono 28 osservazioni, da quello rosso e arancione 15. Ulteriori parametri impostati sono stati: distanza dei raycast e raggio della sfera (se uguale a 0 è un raycast).

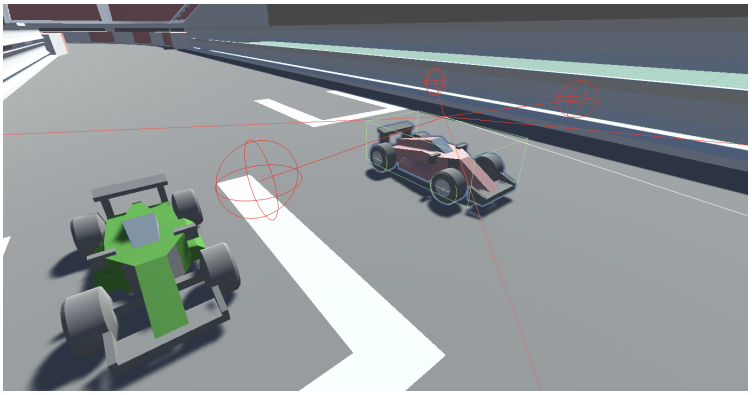


Figura 8: Rilevazione muro e agente.

Tali sensori hanno permesso all'agente di muoversi considerando la distanza dai muri e da altre macchine così da avere un andamento quasi privo di collisioni.

Learning

Come tecnica di apprendimento automatico, per questo task, si è preferito un algoritmo di reinforcement learning; un agente, in un ambiente interattivo mediante una tecnica di "trial and error" cerca di apprendere valendosi dei feedback ricevuti sulle proprie azioni e dell'esperienza cumulata.

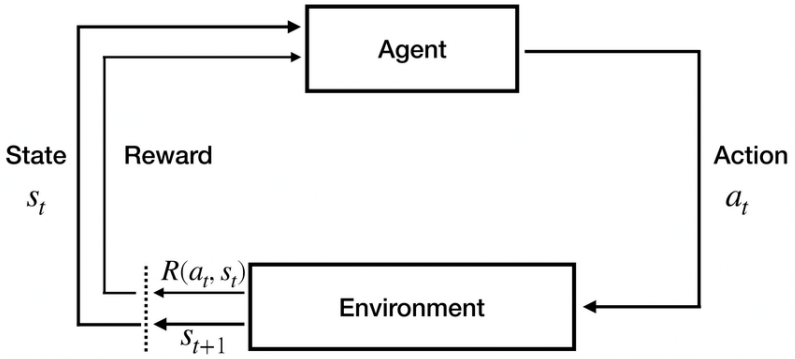


Figura 9: Schema algoritmo reinfoncement learning [SB18].

L'agente ad ogni passo temporale effettua una serie di osservazioni dell'ambiente che ne definiscono lo stato in cui si trova; in base allo stato sceglie l'azione da effettuare e ottiene delle reward che quantificano la bontà dell'azione appena eseguita. L'obiettivo dell'algoritmo è ricercare un modello delle azioni che massimizzi il totale delle reward cumulate dall'agente.

Logica dell'agente

All'inizio di ogni episodio viene effettuato lo spawn delle vetture che, come già accennato, riguarda la loro posizione ed orientamento sul circuito; inoltre vengono resettati tutti i checkpoint presenti sulla pista (il loro utilizzo nel dettaglio verrà approfondito più avanti).

```
public override void OnEpisodeBegin() {
    carDriver.SpawnPosition();
    trackCheckpoints.ResetCheckpoint(transform);
}
```

Successivamente, aiutandosi anche con lo schema precedente, è facile intuire come sia necessario definire lo stato in cui l'agente si trova. Lo stato è composto da diverse osservazioni; alcune sono autogestite direttamente da Unity sulla base dei sensori posizionati sulla vettura, mentre altre sono calcolate mediante il metodo `CollectObservations(...)`.

```
public override void CollectObservations(VectorSensor sensor){
    Vector3 checkpointDistance = checkpointTransform.position − transform.position;
    sensor.AddObservation(agentRb.velocity);
    sensor.AddObservation(checkpointDistance);
    AddReward(−0.01f);
}
```

In particolare è possibile notare come vengano aggiunte al vettore `sensor` la velocità istantanea dell'agente e la distanza dal prossimo checkpoint. Il metodo si conclude assegnando una piccola reward negativa. L'utilità di tale reward si può constatare quando l'agente si trova in stati dove non riceve nessuna reward negativa; in tal caso potrebbe esser tentato di restare nello stesso stato, cadendo così in una fase di stallo. In questo modo, assegnando ad ogni osservazione un piccola reward negativa si sprona l'agente ad evolvere il proprio stato in cerca di più proficui. Nel circuito ciò si traduce in agenti che cercano di completare il più velocemente possibile il giro di pista.

Lo stato, una volta calcolato, diventa l'input dell'algoritmo di reiforcement learning; l'output è possibile esaminarlo implementando il metodo `OnActionReceived(...)`, ovvero un listener che viene chiamato alla fine dell'esecuzione dell'algoritmo.

```
public override void OnActionReceived(ActionBuffers actionBuffers) {
    float horizontal = actionBuffers[0];
    float vertical = actionBuffers[1];
    carDriver.SetInputs(vertical, horizontal);
}
```

Come già accennato nella sezione del controllo dell'agente, le due variabili `horizontal` e `vertical` sono continue in un dominio $[-1, +1]$ e definiscono lo spostamento del veicolo nello spazio. Volendo approfondire in modo

dettagliato la logica di funzionamento dei vari step all'interno di un episodio bisogna introdurre il concetto di decision period. Il decision period è la frequenza con cui l'agente richiede una decisione. Si è scelto di impostare `Decision Period = 5`, ciò significa che l'agente richiederà una decisione ogni 5 academy step [1]. Ogni 5 academy step quindi si eseguono nuove osservazioni e riparte il ciclo (Fig. 9). Reward più cospicue sono assegnate durante il moto del veicolo, conseguenza delle azioni che l'algoritmo suggerisce di compiere all'agente. Le reward sono fondamentali per produrre un apprendimento che abbia senso. Sono lo strumento principale che consente all'agente di capire quali siano le azioni migliori da prendere a seconda dello stato in cui si trova. La logica di assegnazione delle reward ruota attorno ad un'idea di base; per fare in modo che l'agente compia il giro della pista è necessario che apprenda in che direzione e verso procedere. A tal proposito sono stati inseriti i checkpoint.

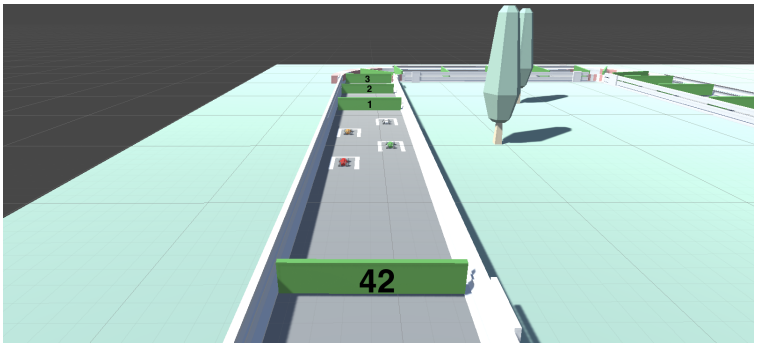


Figura 10: Numerazione dei checkpoint

Questi ultimi sono presenti lungo tutta la pista e hanno un ordine ben preciso. Seguono una numerazione crescente che parte dal primo, posizionato subito dopo il traguardo, arrivando all'ultimo che invece è posizionato appena prima. L'agente acquisisce reward positive se collide con i checkpoint seguendo l'ordine stabilito, mentre ne ottiene di negative se collide più volte con lo stesso checkpoint o addirittura torna indietro. Ciò garantisce che

[1] con un academy step ci si riferisce al processo che avviene nel momento il cui si attiva il metodo RequestDecision() sull'agente. Semplificando, ciò provoca l'invio all'ambiente Python delle osservazioni raccolte e l'esecuzione della rete neurale; il processo ovviamente è bloccato fino al termine dell'elaborazione. Infine, l'oggetto Academy che gestisce l'intero processo è incaricato di triggerare il metodo OnActionReceived() dell'agente passandogli il buffer delle azioni.

il veicolo impari durante l'apprendimento a completare la pista seguendo il verso giusto, mentre le reward assegnate alla collisione con i muri insegnano all'agente a mantenere orientativamente sempre una posizione centrale in pista. Traducendo il discorso in codice, è possibile osservare i metodi della clsse `CarAgentDriver.cs` che si attivano quando il veicolo entra in contatto con gli oggetti della pista.

```
private void OnCollisionEnter(Collision collision ){
    if ( collision .gameObject.TryGetComponent<Wall>(out Wall wall))
        AddReward(-0.5f);
    if ( collision .gameObject.TryGetComponent<CarDriver>(out CarDriver
        carDriver))
        AddReward(-0.4f);
}
```

È punito con reward negativa anche quando la vettura collide con altre macchine. In realtà non viene punito il solo contatto (e quindi la collisione), ma anche il tempo che il veicolo trascorre in collisione con l'oggetto (il muro o un'altra macchina).

```
    private void OnCollisionStay(Collision collision ) {
        if ( collision .gameObject.TryGetComponent<Wall>(out Wall wall))
            AddReward(-0.005f);
        if ( collision .gameObject.TryGetComponent<CarDriver>(out CarDriver
            carDriver)) AddReward(-0.004f);
    }
```

La collisione con i checkpoint non ha effetti fisici essendo quest'ultimi degli oggetti non solidi, per cui il listener che si attiva in questo caso è diverso, ed è `OnTriggerEnter(...)`.

```
private void OnTriggerEnter(Collider other){
    if(other.gameObject.tag == "Checkpoint"){
        checkpointSingle = other.GetComponent<CheckpointSingle>();
        isCorrect = checkpointSingle.IsCorrectCheckPoint(carDriver);
        isLastCheckpoint = checkpointSingle.IsLastCheckpoint();
```

```
        if (isCorrect){
            AddReward(+1f);
            if (isLastCheckpoint)
                AddReward(+2f);
        } else {
            AddReward(-0.05f);
        }
    }
}
```

Il metodo `IsCorrectCheckPoint(...)` controlla che il checkpoint colliso sia corretto, cioè che sia il checkpoint che la macchina dovrebbe colpire seguendo l'esatta numerazione e disposizione in pista. Su tale base vengono poi assegnate le reward.

Deep Reinforcement Learning

Il task di guida autonoma è definito come un ambiente multi agente articolato, le cui osservazioni riguardano svariati oggetti della scena, come muri e checkpoint, oltre che parametri fisici dell'agente, ad esempio la velocità di movimento. Dunque, la natura e la complessità del task hanno portato alla scelta di algoritmi di Deep Reinforcement Learning piuttosto che all'applicazione di algoritmi tradizionali quali Q-learning e SARSA. In particolare, il motivo deriva dalla difficoltà di rappresentare una funzione Q nel modo convenzionale, ovvero sotto forma di tabella.

state	action	$Q(s, a)$
...	...	...

Così, la soluzione è quella di approssimare la funzione Q tramite una *value network*, ovvero una rete neurale, in modo da predire il valore $Q(s, a)$ dati in ingresso lo stato s e l'azione a. Nel nostro caso parliamo di *policy network*, cioè una funzione che dato uno stato in ingresso produce una distribuzione di probabilità sulle possibili azioni.

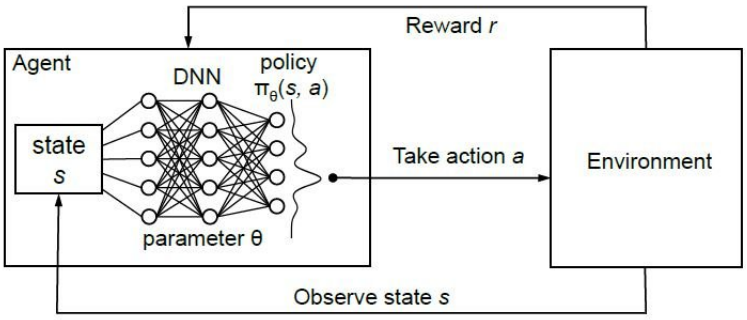


Figura 11: Schema di Deep RL.

Durante l'apprendimento è stata addestrata una rete neurale profonda caratterizzata da una serie di neuroni di input corrispondenti alle osservazioni dell'agente, e da neuroni di output rappresentanti le possibili azioni (Fig. 12). Queste ultime, come specificato nella sezione del Controllo, sono definite come "orizzontale" e "verticale" in quanto specificano il movimento in avanti (e indietro) e il movimento a destra (e a sinistra).

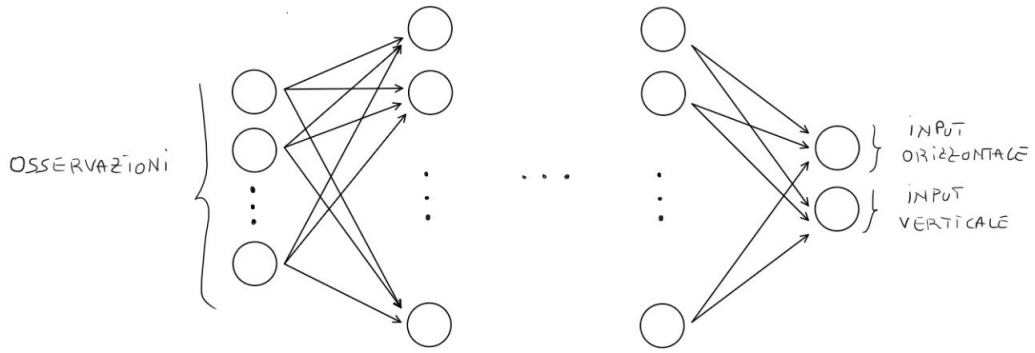


Figura 12: Policy network.

L'algoritmo di Reinforcement Learning utilizzato è stato il PPO (Proximal Policy Optimization) da cui è seguito un confronto necessario con il SAC (Soft Actor-Critic) che verrà riportato nella sezione di Valutazione.

PPO (Proximal Policy Optimization). Il PPO è un algoritmo on-policy proposto da OpenAI che apre la strada ad una nuova famiglia di metodi Policy Gradient. L'idea dietro questi metodi è di massimizzare la *expected reward* seguendo una policy parametrizzata (e.g. coefficienti di un polinomio complesso o pesi e bias di una rete neurale)

$$\mathcal{J}(\theta) = \mathbb{E}[r(\tau)]$$

dove τ rappresenta la traiettoria (sequenza di stato, azione e reward che indica ciò che è accaduto in a sequenza temporale) e $r(\tau)$ la reward totale per una data traiettoria. L'obiettivo è trovare i parametri θ che massimizzano $\mathcal{J}$. Un modo per farlo è tramite la salita o discesa del gradiente.

Tornando al PPO, questo non è che un alternarsi di campionamento dei dati tramite l'interazione dell'agente con il mondo esterno, e di ottimizzazione di una funzione obiettivo tramite *stochastic gradient ascent*. Prima di arrivare allo pseudocodice dell'algoritmo, vediamo come si ottiene la funzione obiettivo del PPO.

La funzione obiettivo dei metodi di Policy Gradient è la seguente

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t \mid s_t)\hat{A}_t]. \tag{3}$$

dove $L^{PG}(\theta)$ è la *policy loss*; π_θ è l'output della policy network (Fig. 12); $\hat{A}$ è detto vantaggio, ovvero una stima del valore relativo dell'azione

selezionata nello stato corrente; $\hat{\mathbb{E}}$ è una media empirica su un determinato lotto di campioni, cioè su un'esperienza fissata dell'agente. Da ciò si ottiene che se

- $\hat{A}$ è positivo: l'azione presa dall'agente nella traiettoria campionata è meglio di quella attesa, dunque verrebbe aumentata la probabilità di scelta di quella azione quando si incontrerà di nuovo uno stato simile.
- $\hat{A}$ è negativo: l'azione non è migliore di quella attesa, dunque avverrà l'esatto opposto.

A questo punto derivando la (3) si ottiene uno stimatore del gradiente $\hat{g}$ che verrà messo nell'algoritmo di stochastic gradient ascent.

Tuttavia, ottimizzare L^{PG} su una medesima traiettoria (esperienza) potrebbe portare ad "aggiornamenti della policy distruttivi". Gli autori, dunque, scelgono di attingere dal TRPO (Trust Region Policy Optimization) per risolvere questa condizione: limitare il policy gradient step in modo da non allontanarsi troppo dalla policy originale, in quanto causerebbe degli aggiornamenti che rovinerebbero interamente la policy.

Iniziamo col definire il rapporto tra l'azione sotto la policy corrente e l'azione sotto la policy precedente

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \quad \text{così } r(\theta_{old}) = 1.$$

$r(\theta)$ sarà maggiore di 1 se un'azione è più probabile per la policy corrente piuttosto che per la policy vecchia; sarà compresa tra 0 e 1 quando l'azione è meno probabile per la policy corrente. Ora moltiplicando $r(\theta)$ per il vantaggio $\hat{A}$ piuttosto che per il logaritmo delle probabilità (come avviene nella (3)), otteniamo la funzione obiettivo del TRPO

$$\begin{aligned} &\text{maximize}_\theta \quad \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \\ &\text{subject to} \quad \hat{\mathbb{E}}_t[KL[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

Il vincolo KL limita il gradient step nell'allontanarsi troppo dalla policy originale (da qui "trust region"). Nondimeno, il KL risulta troppo pesante nel processo di ottimizzazione. Così si giunge alla *Clipped Surrogate Objective*, ovvero alla funzione obiettivo del PPO

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{4}$$

Da un punto di vista grafico si può comprendere come il risultato del vincolo KL sia ugualmente ottenibile, ma in modo più efficiente.

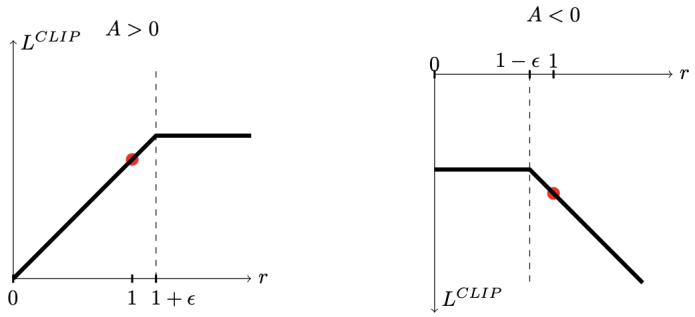


Figura 13: [SWD+17]

Osserviamo il grafico a sinistra, il caso in cui l'azione scelta è meglio dell'effetto atteso. Nel grafico, la loss function si appiattisce quando r diventa troppo grande o quando l'azione è molto più probabile sotto la policy corrente. Dato che non si vuole eccedere con l'aggiornamento andando troppo avanti con lo step, l'idea è tagliare (clip) la funzione così come bloccare il gradiente tramite una linea piatta. Analogamente accade nel caso in cui il vantaggio è negativo: la linea si appiattisce quando r si avvicina a zero, ovvero quando l'azione scelta è molto meno probabile rispetto alla policy corrente (non si vuole che l'azione abbia probabilità zero).

Il tutto viene riassunto dalla loss function ultima che considera due termini aggiuntivi

$$L_t^{CLIP+VS+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)],$$

dove il primo termine è la (4), il secondo termina è la Mean Squared Error della value function, l'ultimo è un termine di entropia che assicura l'esplorazione dell'agente; c_1 e c_2 sono iperparametri.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Come detto inizialmente, l'algoritmo è un alternarsi di raccolta di esperienze e di ottimizzazione. Il ciclo più interno porta diversi attori ad interagire con l'ambiente generando sequenze per calcolare il vantaggio $\hat{A}$. La seconda parte ottimizza L tramite discesa del gradiente (in particolare stochastic gradient ascent).

SAC (Soft Actor-Critic). SAC è un algoritmo off-policy il cui obiettivo è trovare la policy ottimale che massimizza la reward attesa a lungo termine e l'entropia a lungo termine

$$J(\pi_\theta) = E_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) + \alpha H(\pi(\cdot \mid s_t))\right]$$

Come suggerisce il nome, l'apprendimento dell'attore è basato su approccio policy gradient, mentre l'apprendimento del critico avviene in modo value-based. In SAC ci sono tre reti: la prima rappresenta una state-value function V_ψ, la seconda una policy function π_ϕ e la terza una Q function Q_θ.

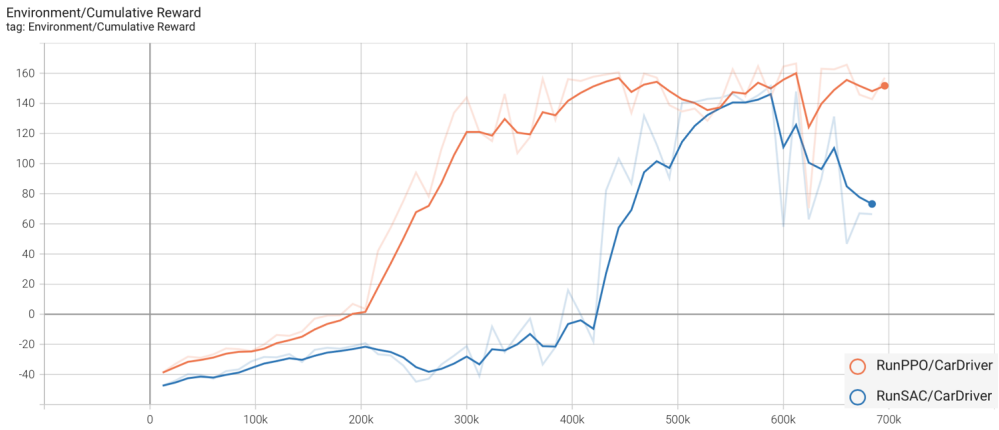
Valutazione

La fase di valutazione ha riguardato il confronto tra i due algoritmi di Deep RL, PPO e SAC, per capire quale dei due fosse più adatto al task di guida autonoma. Inizialmente sono stati svolti diversi tentativi di apprendimento al fine di individuare la configurazione migliore degli iperparametri. Una volta selezionati, sono stati lanciati due apprendimenti ciascuno con un algoritmo diverso. La valutazione è avvenuta sui seguenti valori:

- *Environment/Cumulative Reward:* la reward cumulativa media dell'episodio rispetto a tutti gli agenti.
- *Policy Loss:* l'ampiezza media della loss function della policy. È indicativa di quanto la policy cambi durante l'apprendimento.
- *Value Loss:* la perdita media dell'aggiornamento della loss function. Riguarda la capacità del modello nel predire il valore di ogni stato.

I due algoritmi non sono stati valutati conferendo al training il medesimo tempo di esecuzione, bensì fissando lo stesso numero di step che è circa 700.000.

Partendo dal primo valore, il grafico mostra la reward cumulativa al crescere degli step. In generale, l'apprendimento con SAC impiega circa 200.000 step in più rispetto al PPO per ottenere una reward positiva.



Inoltre, la reward cumulativa del SAC è sempre al di sotto di quella ottenuta con PPO. Agli ultimi step, indicati dai due pallini nel grafico, si hanno queste statistiche:

Algoritmo	Env. Reward	Step	Durata
PPO	157	696k	23m 11s
SAC	66.4	684k	1h 21m 33s

Volendo soffermarci unicamente su una valutazione delle reward cumulate in relazione al tempo impiegato, il PPO è di gran lunga superiore. Nondimeno, per comprendere al meglio la differenza tra i due addestramenti, sono stati valutati i valori di loss. La Policy Loss valuta la loss che si ha nella predizione della migliore azione.

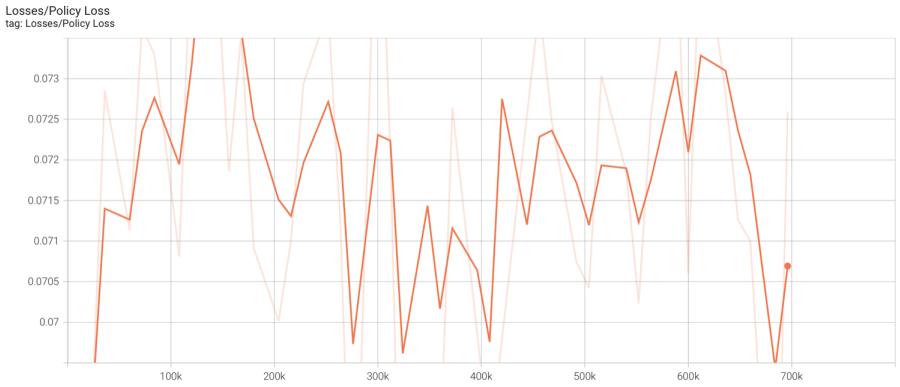


Figura 14: Policy Loss PPO

Nel caso del PPO (Fig. 14) si nota una forte variazione nei vari step, indicativo di come la policy cambi molto durante l'apprendimento. L'andamento generale è di decremento, che è ciò che ci si aspetta da un buon training.

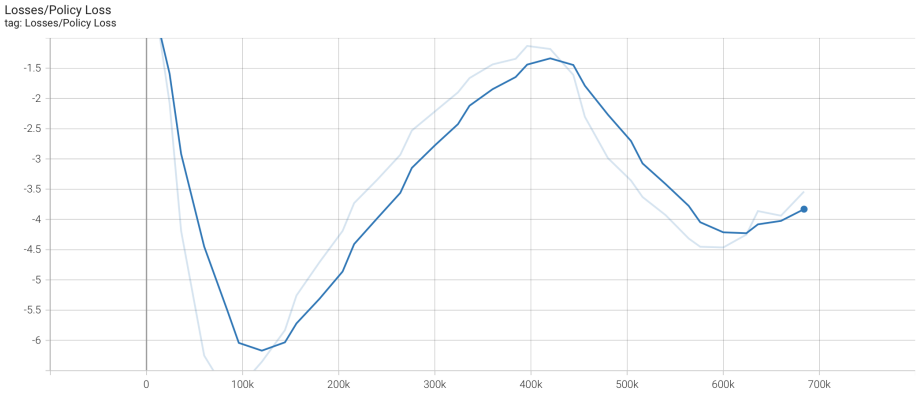


Figura 15: Policy Loss SAC

Riguardo il SAC (Fig. 15) si nota come la loss della policy tenda inizialmente a decrescere, dunque mostrando una forte capacità dell'agente nel predire l'azione corretta. Dall'episodio 120.000 circa la policy inizia a cambiare e le predizioni peggiorano. Dando uno sguardo agli ultimi episodi, si osservano le seguenti statistiche:

Algoritmo	Policy Loss	Step	Durata
PPO	0.007	696k	22m 47s
SAC	-3.54	684k	1h 21m 33s

Sebbene il valore ultimo della Policy Loss per SAC sia inferiore, è necessario ricorrere ad una considerazione dei tempi: è probabile che con la medesima durata il PPO avrebbe quantomeno raggiunto quel valore.

Infine, è stata posta una disamina sulla Value Loss, ovvero l'errore nella predizione del valore di ogni stato.

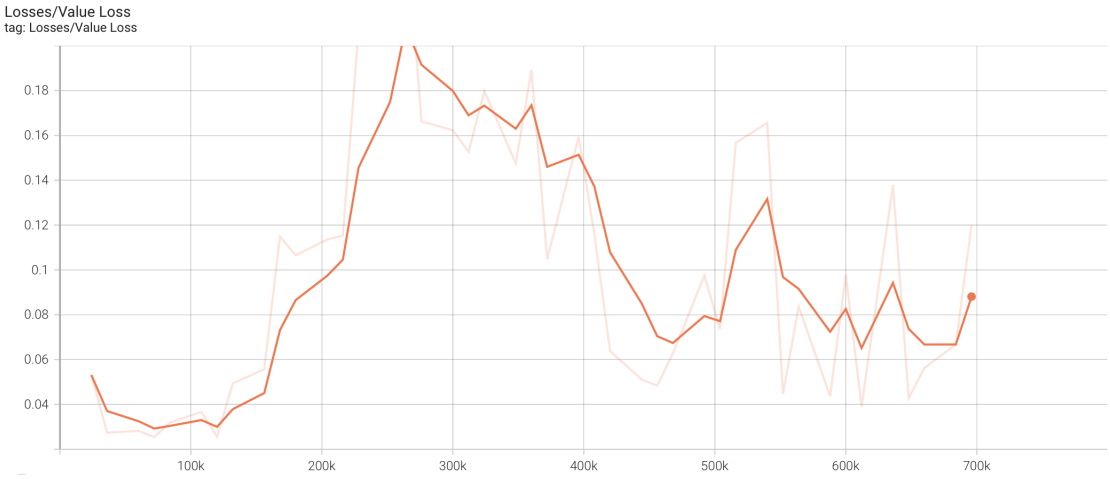


Figura 16: Value Loss PPO

La tendenza dovrebbe essere di incremento nella fase iniziale, precedente ad una fase di decremento della loss una volta che le reward si sono stabilizzate. Riguardo il PPO (Fig. 16) è ciò che si ottiene. Dall'altro lato non si ha lo stesso andamento.

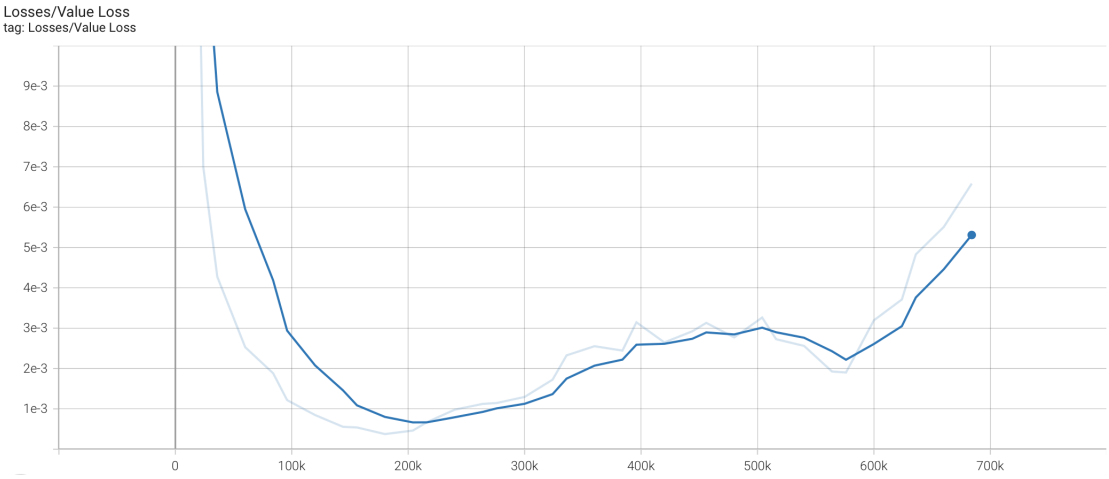


Figura 17: Value Loss SAC

Il SAC (Fig. 17) tende ad predire bene nei primi step, per poi mostrare un peggioramento dal 200.000esimo. Le statistiche agli ultimi step sono le seguenti:

Algoritmo	Value Loss	Step	Durata
PPO	0.1201	696k	22m 47s
SAC	6.583e-3	684k	1h 21m 33s

Da ciò notiamo anche in questo caso che il valore della Value Loss del SAC è nettamente inferiore, tuttavia è da considerare il tempo impiegato per arrivarci.

In conclusione, il PPO si è mostrato più adatto al task di guida autonoma e i motivi sono legati alla natura dei due algoritmi. La caratteristica off-policy del SAC lo porta ad essere migliore in un ambiente lento, ovvero con una durata di step di circa 0.1 (100ms), la dove l'ambiente considerato ha uno step di circa 0.02 (20ms). Inoltre, nonostante il SAC abbia raggiunto valori migliori sulle loss, esso è visibilmente più lento nell'apprendimento e meno capace di generalizzare. Su quest'ultimo punto ci si è soffermati maggiormente in quanto vincolo fortemente richiesto.

Per valutare la generalizzazione dell'apprendimento sono stato create due piste: una per il training ed una per il testing. La prima è mostrata in figura (1). La figura (18) mostra il circuito di testing.

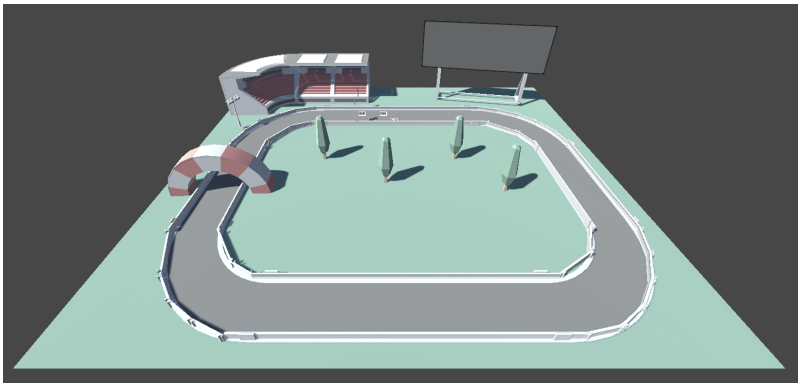


Figura 18: Circuito di testing

Come si può notare, il circuito (1) è più complesso per permettere agli agenti di apprendere curve più ostiche. Una volta terminato l'addestramento, gli agenti sono stati testati sul circuito (18) mostrando da parte del PPO una migliore generalizzazione.

Conclusione

Lo studio e lo sviluppo di auto a guida autonoma è estremamente complesso a causa degli innumerevoli fattori da dover prendere in esame; ricreando un ambiente più semplificato, lo scopo del progetto prevedeva l'utilizzo di algoritmi di reinforcement learning al fine di realizzare un agente che riuscisse a muoversi nello spazio, imparando con un certo grado di confidenza a navigare all'interno di vari circuiti cercando di mantenere una traiettoria che fosse abbastanza pulita da evitare collisioni con i muri adiacenti alla pista e con le altre macchine.

Tramite la piattaforma Unity è stato possibile progettare il modello della vettura e definire due diversi ambienti, fondamentali per le fasi di training e di testing. Entrambi gli algoritmi di apprendimento principali rilasciati da Unity (PPO e SAC) sono stati adoperati in fase di training e confrontati in fase di testing con l'intento di valutare quale si adattasse meglio al task in esame. Dalle diverse considerazioni fatte, esposte nella sezione precedente, è risultato come il PPO in definitiva sia risultato più appropriato.

Gli sviluppi futuri possono essere molteplici. In primo luogo si potrebbe pensare di rendere più complesso il controllo dell'agente aggiungendo una forma di accelerazione. Inoltre, il task si potrebbe rendere competitivo aggiustando appositamente le reward e conferendo all'apprendimento osservazioni utili a tale scopo, come valori di performance di altre macchine o velocità di completamento del giro di pista.

Bibliografia

[GR] Nadine Govers and Rowan. `https://kenney.nl/assets/racing-kit`.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.