Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Analisi dell'ambiente AWS DeepRacer per la sperimentazione di tecniche di Reinforcement Learning

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Sara Vorabbi

Sessione III
Anno Accademico 2019/2020

Sommario

Il Deep Reinforcement Learning è una tecnica di Machine Learning che negli ultimi anni è diventata sempre più diffusa e studiata dai ricercatori per la sua capacità di unire la potenza di rappresentazione del mondo attraverso le reti neurali con l'abilità di capire quella rappresentazione e conseguentemente decidere quale sia il comportamento migliore in ogni data situazione.

In questo elaborato basato sull'esperienza fatta nel corso del tirocinio, viene analizzato il progetto proposto da Amazon Web Services, DeepRacer, che si propone come trampolino di lancio nel mondo del deep reinforcement learning per coloro che non hanno esperienza in questo campo. Il progetto consiste nell'addestrare un modellino di automobile in modo che possa imparare autonomamente a gareggiare nei circuiti, sia reali che simulati. Verrà analizzata l'efficacia di questo progetto e chi può beneficiare da esso.

Introduzione

Il deep reinforcement learning è un approccio al machine learning che negli ultimi anni ha ricevuto molte attenzioni da parte dei ricercatori per le sue possibili applicazioni e per la sua capacità di risolvere problemi sempre più complessi e difficilmente affrontabili da altre tecniche. Questo metodo si basa sulla presenza di un agente che interagisce con l'ambiente circostante e impara a raggiungere un obiettivo attraverso l'interazione con esso. Questa caratteristica lo rende ideale per quei problemi che devono prendere in considerazione un ambiente nel quale il modello deve operare.

L'idea di affrontare questo argomento nasce dalla necessità di ricercare un ambiente nel quale poter sviluppare e testare algoritmi di deep reinforcement learning. Si è presentata l'opportunità di analizzare il progetto AWS DeepRacer che ha l'obiettivo di avvicinare nuove persone ai meccanismi del reinforcement learning (RL). Nel corso del tirocinio curricolare si è cercato di capire se questo ambiente potesse essere sfruttato come ambiente di testing di algoritmi RL. DeepRacer offre la possibilità di addestrare modelli di automobili autonome, che utilizzano il reinforcement learning per interagire con l'ambiente e imparare quale sia il comportamento migliore che permette di raggiungere la linea del traguardo.

L'elaborato è suddiviso in tre capitoli. Nel primo viene introdotto il concetto di intelligenza artificiale, machine learning e deep learning, con una digressione sulla topologia delle reti neurali e sui meccanismi che rendono possibile l'addestramento di queste strutture. Nel secondo capitolo viene preso in considerazione il problema del reinforcement learning. Viene presentata

la sua modellizzazione matematica come problema decisionale sequenziale, quindi vengono analizzati gli elementi che lo compongono. Nel terzo capitolo viene analizzato il progetto AWS DeepRacer nel dettaglio. Vengono descritte le caratteristiche del modello e i suoi elementi centrali; ciò su cui l'utente può apportare modifiche o meno e le conseguenti limitazioni.

Indice

Elenco delle figure

Capitolo 1

Intelligenza Artificiale e Machine Learning

L'Intelligenza Artificiale è una disciplina che ha come obiettivo quello della risoluzione di problemi con un tipo di approccio che a primo impatto potrebbe sembrare esclusivo alle competenze degli esseri umani: l'emulazione dell'intelligenza umana [24]. Sin dai primi anni di studio nell'ambito delle Intelligenze Artificiali ci si è resi conto di come problemi computazionalmente difficili per gli esseri umani potessero essere risolti velocemente da macchine e, al contrario, di come semplici azioni svolte da esseri umani potessero essere estremamente difficili da formalizzare e replicare tramite software, come l'identificazione di un viso o la capacità di comprendere un discorso. [11].

Considerando che i problemi che ci vengono sottoposti nella vita di tutti i giorni possono variare dal banale all'estremamente complesso, non sorprende la necessità di approcci che siano all'altezza di restituire risultati utili e in grado di operare in condizioni reali, spesso molto complicate.

1.1 Cos'è il Machine Learning

In termini pratici, il Machine Learning è un sottoinsieme di problemi dell'Intelligenza Artificiale che si pone come obiettivo quello di risolvere compiti

che noi tutti, in quanto esseri umani, risolviamo non tanto a partire da un ragionamento schematizzabile tramite una serie di azioni, quindi facilmente replicabile da una macchina, ma tramite l'intuizione.

L'idea è quella di creare, attraverso il Machine Learning, dei sistemi che possano risolvere problemi che vanno dalla classificazione di documenti al riconoscimento di oggetti specifici nelle immagini. Dato un qualsiasi input, non viene chiesto al sistema di svolgere un'azione specifica, ma di imparare dall'esperienza accumulata in una fase di addestramento svolta precedentemente le azioni più corrette da eseguire dato un qualsiasi input. E' intuitivo pensare che questi tipi di sistemi abbiano bisogno di una grande quantità di dati con certe caratteristiche, anche dette feature, per l'addestramento e che è proprio grazie all'analisi di queste che il sistema impara. Da qui sorge il primo problema del Machine Learning, ovvero la forte necessità di dati strutturati nei quali siano esplicitate queste features. Un secondo problema può essere quello di capire quali siano le features importanti da estrapolare che permettano al sistema di imparare più velocemente. La soluzione viene offerta dal Deep Learning, che elimina questo passaggio intermedio di esplicitazione delle features e che delega questo lavoro alle Reti Neurali.

1.2 Deep Learning

Il Deep Learning è una classe di algoritmi di Machine Learning. Prendono in input dei dati che presentano una grande quantità di caratteristiche ad alto livello, e per questo difficilmente estraibili manualmente. L'estrazione di queste caratteristiche avviene attraverso la Rete Neurale, un modello computazionale formato da due o più livelli, anche detti layer, di neuroni.

1.2.1 Neuroni

I neuroni sono delle unità di computazione che comunicano fra loro attraverso l'invio di segnali utilizzando connessioni pesate [21]. Ogni neurone prende in input tutti gli output del layer precedente e calcola un output che

invierà a sua volta a tutti i neuroni del layer successivo. Il numero calcolato "all'interno" di un neurone viene chiamato valore di attivazione e il suo calcolo è mostrato nella formula sottostante.

$$a_j^{k+1} = \sigma\left(\left(\sum_{i=1}^{k} a_i^{(k)} w_{i,j}^{(n)}\right) + b_k\right)$$

Viene fatta la somma pesata degli input, moltiplicando il numero di attivazione a_i^k in output dall'i-esimo neurone del layer k con il peso $w_{i,j}$ dell'arco (i, j) che connette il neurone i con il j-esimo neurone del layer *k+1*. A questa somma viene aggiunto un bias arbitrario e il risultato di questa operazione viene dato in pasto alla funzione di attivazione (o activation function) che permette di capire se il neurone debba attivarsi o meno [21].

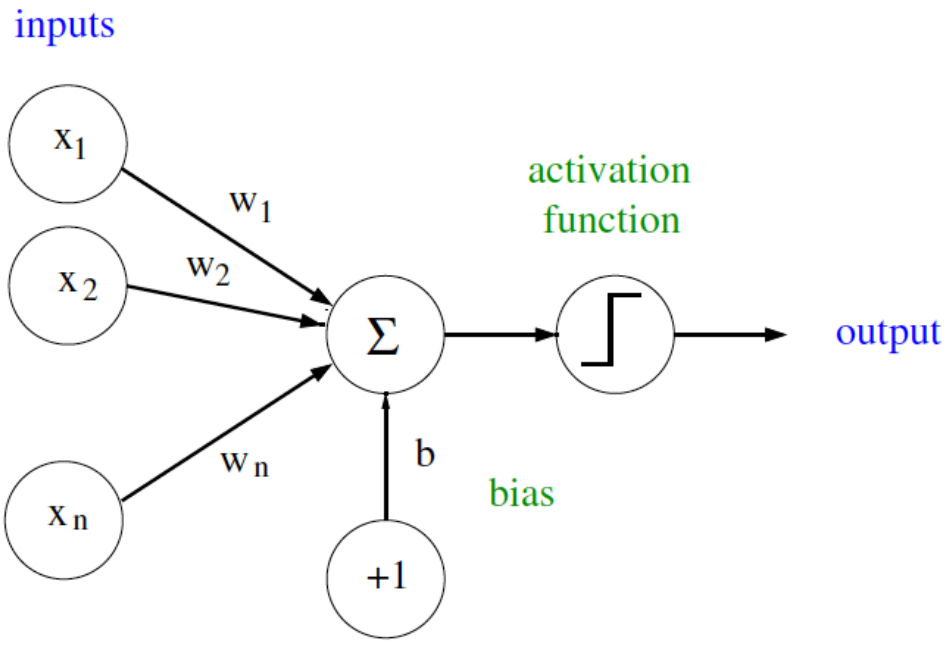

Figura 1.1: Esempio di Neurone [3]

Due esempi di funzioni di attivazione possono essere la funziona sigmoide e la funzione rettificatore (anche chiamata ReLU). La prima usata sin dagli inizi dello studio del Deep Learning e la seconda, introdotta più recentemente,

che ha portato a migliorare le performance degli algoritmi e la velocità di apprendimento in modo tale da essersi affermata come funzione di attivazione standard per i problemi di Deep Learning [14][16].

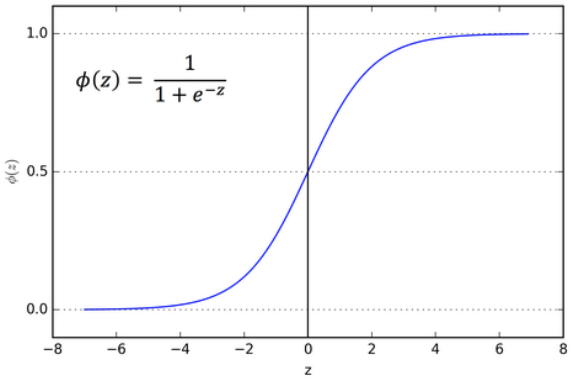

Figura 1.2: Funzione sigmoide [12]

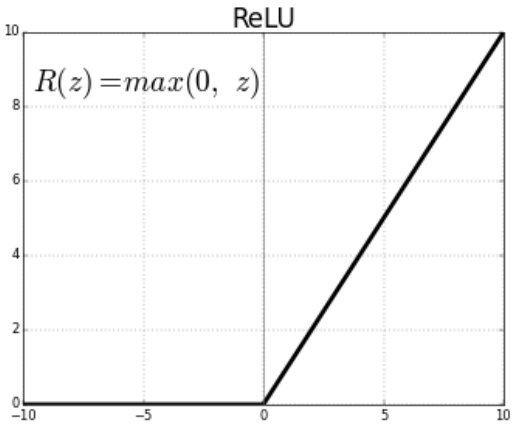

Figura 1.3: Funzione ReLU [12]

La scelta di usare la funzione rettificatore viene da osservazioni biologiche; infatti, i neuroni del sistema nervoso non sono sempre attivi e quando lo sono raggiungono raramente lo stato di saturazione. Piuttosto hanno un output proporzionale al proprio input; ciò potrebbe essere una spiegazione del perchè questo tipo di funzione risulti più adeguata. [10]

1.2.2 Topologia

Le Reti Neurali si possono suddividere in due macrocategorie, le reti Feed-Forward e le reti Ricorrenti. Le reti Feed Forward permettono ai segnali di proseguire in una sola direzione, c'è un'assenza di feedback da parte della rete e di conseguenza l'output dipende esclusivamente dall'input. Le reti Ricorrenti sono caratterizzate dalla presenza di cicli nel grafo. Questa caratteristica implica che l'output di un neurone può essere preso in input, per esempio, dal neurone stesso ed essere processato di nuovo. Degno di nota è il fatto che questo tipo di rete permetta di processare anche dati di lunghezza variabile [11]. La scelta di una Rete Neurale Feed Forward rispetto a una Ricorrente può variare in base agli obiettivi che devono essere raggiunti e alla classe di problema che deve essere risolta. Esempi importanti sono l'object detection per le reti feed-forward e lo speech recognition per le reti ricorrenti.

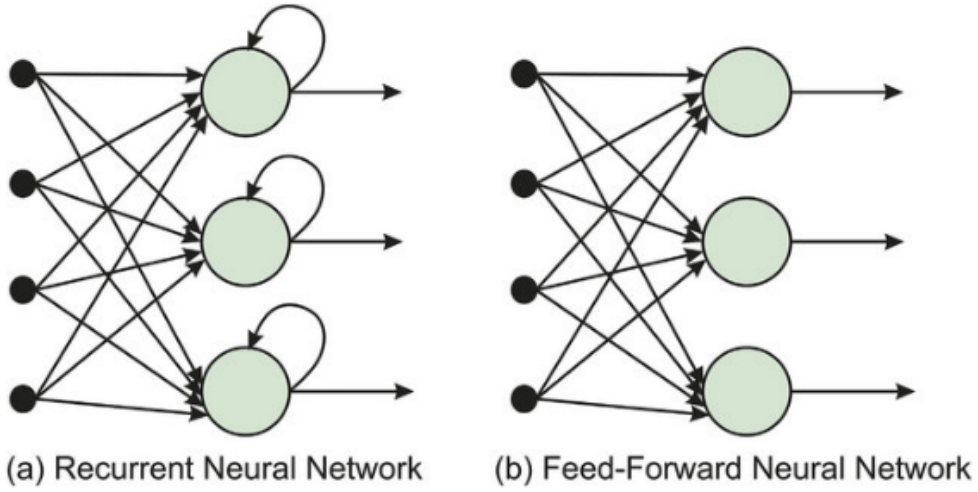

Figura 1.4: Esempi di Reti Neurali [7]

1.2.3 Reti Neurali Feed Forward

Le reti Neurali Feed Forward sono tipicamente organizzate a livelli. Il layer che prende i dati in ingresso viene chiamato input layer mentre il livello che restituisce il risultato viene chiamato output layer. Fra questi due livelli esterni è possibile inserire uno o più livelli, detti nascosti (o hidden). Se è presente un solo livello nascosto la rete è considerata di tipo Shallow; se sono presenti più di un livello nascosto, la rete viene chiamata Deep.

Una ulteriore suddivisione può essere fatta distinguendo vari tipi di layer che si possono trovare in una rete Feed Forward: livelli densi e livelli convoluzionali.

Livelli Densi

I livelli densi possono essere rappresentati tramite un grafo fortemente connesso, quindi tutti i neuroni presenti al k-esimo livello sono collegati al $k+1$-esimo livello.

Livelli Convoluzionali

I livelli convoluzionali sono una tipologia di layer ideale per l'analisi di dati strutturati a griglia, o bidimensionali. L'utilizzo di reti contenenti livelli convoluzionali per l'estrazione di pattern si è rivelato estremamente efficiente per problemi come il riconoscimento vocale, il face e object detection [2] e per questo motivo sono diventati molto popolari.

Un livello convoluzionale esegue tre operazioni fondamentali:

1. Applica una operazione di convoluzione sull'input, ovvero opera una trasformazione su una matrice in input applicandovi un'altra matrice detta filtro. Invece di dare in input a un neurone la totalità dei neuroni del livello precedente, viene considerata una regione specifica nella matrice input I alla quale verrà applicata un filtro, chiamato anche kernel. L'output di questa operazione andrà al livello successivo. In

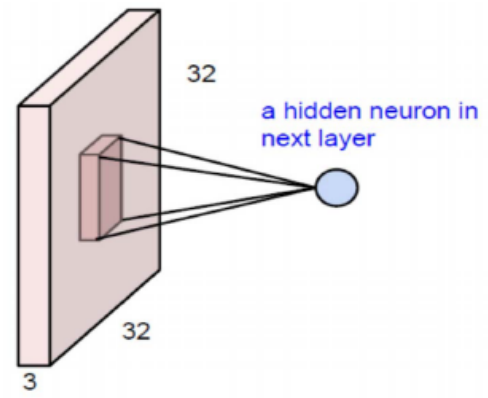

Figura 1.5: Esempio di livello convoluzionale [2]

questo modo la convoluzione permette di ridurre drasticamente il numero di parametri utilizzati fra un layer e quello successivo [2]. Questo numero può essere ridotto ancora se il filtro resta invariato per ogni regione del layer analizzato. Il numero di operazioni che devono essere svolte può scendere dall'ordine del miliardo al migliaio, diminuendo lo spazio necessario per la memorizzazione dei pesi e riducendo i tempi di computazione.

2. Applica una funzione di attivazione, tipicamente una ReLU.

3. Applica una funzione di pooling tale da far diventare la rappresentazione invariante rispetto alle trasformazioni e alle perturbazioni che l'input può subire [25]. Si può considerare l'esempio di una feature da riconoscere in un'immagine: la rete deve essere in grado di riconoscere una particolare feature indipendentemente dalla posizione, che sia in alto a sinistra, in basso a destra o ruotata. [11].

Caratteristiche come la *connettività sparsa*, i *parametri condivisi* e l'*invarianza rispetto alla traslazione* sono elementi chiave che hanno permesso alle reti convoluzionali di migliorare i modelli di apprendimento Deep.

1.2.4 Come addestrare una Rete Neurale

Nella programmazione tradizionale è presente un algoritmo che prende un input, lo processa e restituisce l'output in modo programmatico. Nel Machine Learning l'approccio è completamente diverso in quanto si dà in pasto a un modello sia l'input che l'output e questo deve imparare a restituire l'output desiderato. Il risultato del Machine Learning è un programma al quale viene insegnato a risolvere un certo tipo di problemi la cui soluzione non può essere codificata in modo fisso (hard-coded) dal programmatore [1]. Nel Deep Learning il modello che deve essere addestrato è la rete neurale.

Il processo di addestramento è fondamentalmente un problema di ottimizzazione. Generalmente è presente una funzione obiettivo (loss function o objective function) che deve essere minimizzata nel corso di varie iterazioni tramite la tecnica della discesa del gradiente (o gradient descent).

Loss Function

I due elementi di una rete neurale che influenzano l'output sono i pesi della rete e i numeri di attivazione. Quest'ultimi non possono essere modificati in quanto vengono calcolati in ogni livello della rete dai neuroni. Quindi gli unici elementi che possono essere modificati sono i pesi della rete. La funzione obiettivo $J(\theta)$, definita come $J : \mathbb{R}^n \to \mathbb{R}$, prende in input il vettore contenente tutti i pesi della rete e restituisce un numero scalare che deve essere minimizzato. La minimizzazione è possibile tramite la modifica dell'input della loss function: è necessario modificare i pesi della rete di iterazione in iterazione in modo che la loss function calcoli un numero inferiore al risultato calcolato nell'iterazione precedente.

Gradient Descent

Per minimizzare la loss function si potrebbe pensare di trovare i suoi punti critici, e di conseguenza un minimo locale, ponendo la derivata della funzione uguale a 0. Non sempre questo calcolo è immediato, quindi si può

utilizzare un altro procedimento: si calcola la derivata $f'(x)$ che permette di trovare la pendenza della retta tangente passante nel punto x e se è positiva, si decrementa l'input shiftandolo verso sinistra, se è negativa si incrementa l'input shiftandolo verso destra. Viene modificato l'input x della funzione f per avere un nuovo output che scenda gradualmente verso il minimo.

Questo concetto viene applicato anche nel caso di una funzione con input multidimensionale, come la loss function. Infatti, trovare gli zeri di una funzione di questo tipo risulta molto difficile e costoso. Viene quindi calcolato il gradiente della loss function, $\nabla_{\mathbf{x}} f(\mathbf{x})$, un vettore nel quale l'i-esimo elemento corrisponde alla derivata parziale di f rispetto a x_i.

I gradienti, in realtà, non indirizzano verso il minimo della funzione, ma verso il massimo. Per questo motivo vengono sommati i negativi dei gradienti per trovare la direzione di discesa "più ripida". Questo metodo viene chiamato gradient descent.

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

Con questa formula troviamo il nuovo vettore di pesi x' con cui aggiornare la rete dove x è il vettore dei pesi e ϵ rappresenta il learning rate, uno scalare positivo che determina quanto grande deve essere il passo che viene fatto nella nuova direzione [11] [20].

Backpropagation

La retropropagazione dell'errore è l'algoritmo attraverso il quale viene implementato il metodo della discesa del gradiente. La backpropagation è il punto focale del deep learning in quanto è attraverso questo algoritmo che la rete impara.

Questo processo può essere suddiviso in due parti fondamentali e viene ripetuto fino alla convergenza del modello:

1. **Forward propagation** o propagazione in avanti del segnale in input: il dato viene preso in input dalla rete neurale e viene processato. Questa

produce un certo output che può essere vicino o meno alla soluzione desiderata. Durante questa fase i pesi della rete sono costanti.

2. **Back propagation** o retropropagazione del segnale d'errore: si calcola la differenza fra l'output effettivo della rete e l'output atteso tramite la loss function. Il risultato è il segnale d'errore che deve essere retropropagato nella rete. Sono poi calcolate le derivate parziali della loss function, ovvero il vettore dei gradienti, rispetto ai valori di attivazione presenti nell'output layer della rete. Il segnale d'errore viene propagato all'indietro fino all'input layer grazie alla regola della catena (chain-rule), che, nel passaggio da layer a layer, aggiorna i pesi di tutta la rete [17] [28].

1.2.5 Supervised vs Unsupervised vs Reinforcement

Nel Machine Learning sono presenti vari paradigmi che possono essere utilizzati per addestrare un modello e il loro utilizzo varia in base al problema che deve essere risolto e al tipo di dati a disposizione. I tre principali sono:

Apprendimento supervisionato (supervised learning): il modello impara prendendo in input un insieme di dati etichettati (structured data). In questo modo la rete sa qual è il risultato verso il quale deve convergere e impara a generalizzare l'output durante la fase di training.

Apprendimento non supervisionato (unsupervised learning): al modello non vengono fornite etichette con le quali classificare i propri input. Questo deve "scoprire" la struttura nascosta dei dati riconoscendo pattern frequenti. I dati, quindi, vengono suddivisi in gruppi senza che l'algoritmo sia a conoscenza delle loro etichette, ma solamente in base alle feature che hanno in comune.

Apprendimento per rinforzo (reinforcement learning): il modello deve imparare interagendo con l'ambiente che lo circonda compiendo delle

azioni, inizialmente randomiche, in base allo stato in cui si trova. Queste possono essere ricompensate o meno, quindi l'obiettivo è quello di massimizzare il premio cercando di capire quali siano le associazioni stato-azione che restituiscono il reward più alto [11].

Capitolo 2

Deep Reinforcement Learning

Il Reinforcement Learning si basa sull'idea che un agente possa imparare attraverso l'interazione con l'ambiente circostante. Questo tipo di approccio può ricordare come le persone o gli animali apprendano in certe situazioni, interagendo con l'ambiente circostante e, ricevendo dei feedback positivi o negativi da esso, aggiustano il loro comportamento per ottenere il risultato desiderato [26]. L'agente riceve delle ricompense, o reward, in base alle azioni compiute con lo scopo di massimizzarne la somma sul lungo periodo, utilizzando l'esperienza accumulata nella fase di addestramento.

2.1 Processi decisionali di Markov

Il reinforcement learning si occupa di processi decisionali sequenziali [9]. Questo tipo di problema può essere modellizzato matematicamente tramite i processi decisionali di Markov (MDP). Grazie a questa formalizzazione è possibile esprimere come le azioni compiute in un certo momento non influenzino solamente gli stati e i reward immediati, ma anche le azioni, gli stati e i reward sul lungo periodo [26]. Allo stesso tempo è caratterizzata dalla proprietà di Markov: la decisione sulla scelta di un'azione in un certo momento viene presa solamente in base all'osservazione attuale fatta dall'agente. Non

c'è nessun interesse nel guardare l'intera sequenza di azioni prese fino a quel determinato istante [9].

Il processo decisionale di Markov è una tupla (S, A, T, R, γ) dove:

- S è l'insieme degli stati dell'ambiente.
- A è l'insieme delle azioni che l'agente può scegliere.
- $T : S \times A \times S \to [0, 1]$ è la funzione di transizione $T(s, a, s')$, restituisce la probabilità di arrivare in un certo stato s' partendo da uno stato s compiendo una azione a.
- $R : S \times A \to \mathbb{R}$ è la funzione di reward $R(s, a)$, restituisce all'agente una ricompensa immediata per aver compiuto un'azione a partendo da uno stato s.
- $\gamma \in [0, 1]$ è il discount factor

E' presente un agente (o agent) che in un certo momento t si trova in uno stato s_t. L'agente decide quale azione a_t compiere in base alla policy π, un'associazione fra stati e azioni. L'ambiente (o environment), come risposta all'azione dell'agente, genera una ricompensa locale r_t (o reward) ed è grazie alla reward function che se ne decide il valore in base alla bontà dell'azione presa. L'agente entra successivamente in un nuovo stato s_{t+1} e ripete que-

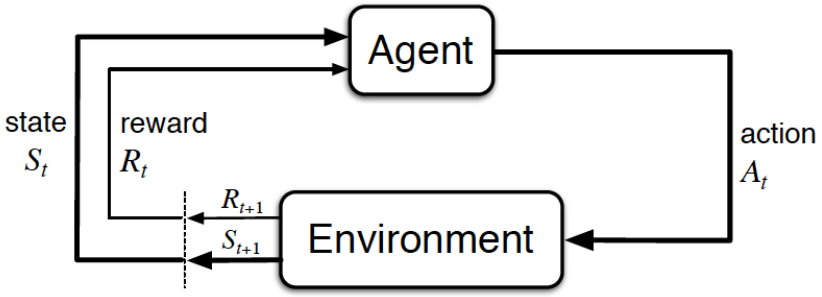

Figura 2.1: Interazione Agente-Ambiente [26]

sto procedimento. L'obiettivo risulta essere la massimizzazione del reward cumulativo (o expected reward) di ogni episodio.

Con il termine episodio (o task) si intende la sequenza di stati, azioni e ricompense che si susseguono nell'interazione agent-environment. L'episodio può essere discreto, quindi finire nel momento in cui si arriva su uno stato terminale, o continuo, proseguendo senza un limite temporale. Nella figura 2.2 è possibile vedere la schematizzazione di un episodio discreto.

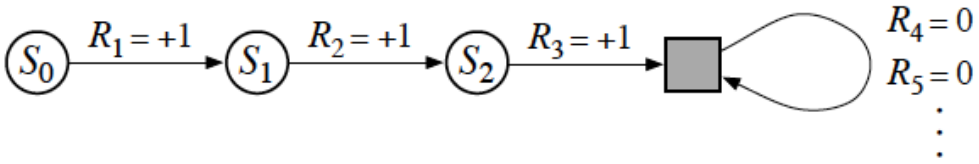

Figura 2.2: State Transition [26]

Per massimizzare i reward attesi si sommano tutte le ricompense locali ricevute dall'ambiente dopo ogni transizione di stato nel corso di un episodio, come mostrato nella formula:

$$R = \sum_{i=\geq 1} \gamma^i r_i$$

Gamma γ, il discount factor, è compreso fra $[0, 1]$ e permette di definire l'importanza che i reward locali futuri hanno sull'expected reward. Se il valore del discount factor si avvicina a 0, l'agente sarà interessato a massimizzare solamente i reward immediati, mentre se si avvicina a 1 l'agente terrà in considerazione i valori dei reward futuri [26].

Il discount factor diventa particolarmente importante nei task continui poiché i reward potrebbero sommarsi per un infinito numero di step. Permette infatti di dare importanza a reward più vicini temporalmente, in modo da evitare di massimizzare l'infinito.

2.2 Elementi del Reinforcement Learning

In questa sezione vengono analizzati i componenti principali del Reinforcement Learning, le loro funzionalità primarie e peculiarità.

2.2.1 Policy

Il componente del RL che definisce quali azioni debbano essere prese dall'agente è la policy, indicata con il simbolo π. L'agente, che si trova in un certo stato s, deve compiere un'azione a e la policy ha il compito di mappare gli stati con le azioni che restituiscono reward più alti.

$$a = \pi_\theta(s)$$

Le policy possono essere suddivise in due categorie:

- Policy deterministiche: dato un determinato stato in input restituiscono una azione specifica, $\pi(s) : S \to A$
- Policy stocastiche: si considera la probabilità che una certa azione possa essere eseguita dall'agente. La policy stocastica viene definita come $\pi(s, a) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ con $\pi(s, a)$ che corrisponde per l'appunto alla probabilità dell'azione a di essere eseguita [9].

La policy può essere implementata tramite una look-up table o una funzione, ma nel deep reinforcement learning viene utilizzata una rete neurale.

Optimal Policy

Una policy π^* viene definita ottimale se il reward atteso è maggiore o uguale a tutte le policy π tenendo in considerazione i reward ottenuti nell'intera durata della funzione [27].

$$\pi^* = argmax_\pi \; E \sum_{t \geq 0} \gamma^t r_t$$

2.2.2 Value Function

Per stabilire se una policy sia efficace o meno, deve essere valutata tramite la value function. Questa indica la somma totale della ricompensa che l'agente può aspettarsi di accumulare sul lungo periodo partendo da un determinato stato [26]. Nonostante sia importante trovare il reward r_i più alto per ogni i-esimo stato di transizione, l'obiettivo del reinforcement learning è quello di massimizzare il value, ovvero la somma di questi reward nel tempo.

Un'analogia concreta può essere quella del gioco degli scacchi nel quale sono sì presenti obiettivi immediati, come mangiare certi pezzi o giocare alcune mosse strategiche che portano un vantaggio nell'immediato, ma il giocatore deve comunque perseguire e raggiungere l'obiettivo finale, ovvero la vittoria.

La stima del value è il problema fondamentale del RL. Se infatti ricevere un reward dall'envirorment dopo aver compiuto un'azione è immediato, la stima del value nel tempo deve essere aggiornata in base a ciò che l'agente osserva nel corso delle varie iterazioni [26].

Vengono definite due funzioni di valutazione: la value function e la state-value function.

- Value function: il value atteso partendo da uno stato s tramite una policy π

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s,\ \pi \right]$$

- State-value function: il value atteso partendo da uno stato s e compiendo un'azione a tramite una policy π

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s,\ a_t = a,\ \pi \right]$$

2.2.3 Modello

L'ultimo elemento del reinforcement learning è il modello che descrive l'ambiente, ovvero imita il suo comportamento [26]. Per massimizzare il

reward, l'agente interagisce con l'environment e questo restituisce una ricompensa. E' possibile adottare due possibili approcci nei confronti dell'ambiente che cambiano radicalmente la strategia di apprendimento: model-based e model-free.

Model-based vs model-free

Si parla di approccio model-based nel caso in cui si utilizzi un modello dell'ambiente che possa simulare il suo comportamento. Si cerca di sfruttare questo tipo di conoscenza per fare deduzioni riguardo agli stati futuri che può assumere [26].

A questo tipo di approccio si contrappone il model-free, che non conosce in modo esplicito la struttura dell'ambiente. Si basa sul metodo euristico del trial and error, prova e sbaglia, in quanto procedendo a tentativi si cerca di capire quale sia la strada più corretta da intraprendere in base all'esperienza accumulata nelle varie iterazioni. Data la sua natura, nel caso di problemi basati sul mondo reale, il model-free ha bisogno di una grande quantità di interazioni con l'ambiente [8].

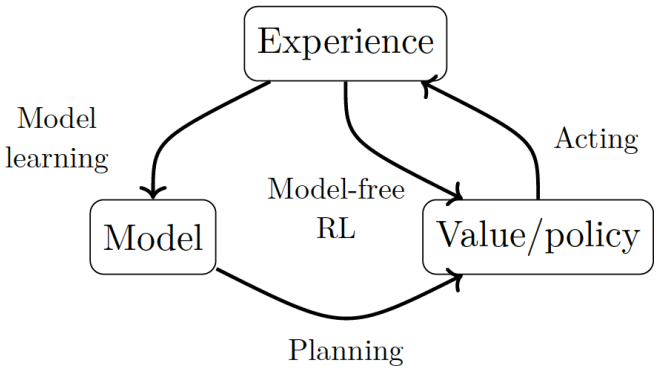

Figura 2.3: Metodi per il RL [9]

Exploration vs Exploitation

Nel contesto di un approcccio model-free, il modello che viene addestrato tramite RL deve scoprire quali siano le azioni più remunerative corrispondenti a certi stati. Viene immediato pensare che sia presente una qualche componente esplorativa.

Il modello, appena inizia l'addestramento, deve esplorare (explore) le varie possibilità che vengono valutate e che, potenzialmente, possono restituire reward maggiori rispetto a quelle scelte in precedenza. Allo stesso tempo si deve considerare che a un certo punto dell'addestramento si saranno raccolte abbastanza informazioni sulle possibili azioni da compiere. Viene naturale pensare che queste possano essere sfruttate (exploitation). Bisogna però operare un trade-off fra esplorazione e sfruttamento trovando un equilibrio fra questi due concetti.

Questo dilemma viene presentato nel "Problema del bandito multi-armato" (Multi-armed Bandit Problem). Si consideri una persona che deve scegliere fra n possibili azioni. A ogni azione corrisponde una ricompensa, chiamata value dell'azione, presa da una distribuzione di probabilità stazionaria sconosciuta. Nel momento in cui viene scelta una certa azione si scopre anche il value corrispondente. Il problema sta nel capire per quanto tempo si debba esplorare il parco delle azioni in riferimento agli stati che vengono sottoposti rispetto allo sfruttare le conoscenze accumulate per ottenere un reward molto alto.

L'exploitation è utile nel momento in cui si vuole massimizzare la ricompensa, ma l'exploration permette di trovare nuove azioni che possono produrre un reward più alto nel lungo periodo, evitando che l'agente si blocchi in un massimo locale.

Nonostante siano stati proposti numerosi approcci, questo dilemma rimane un problema irrisolto [26].

Value-based vs Policy-based

Fra gli approcci model-free sono presenti due metodi per la ricerca della optimal policy: il value-based e il policy-based.

- I metodi policy-based, detti anche actor-only, cercano di approssimare la policy nella forma di una rete neurale parametrizzata, agendo in modo diretto su di essa. Viene ottimizzata nel corso delle iterazioni con la tecnica della discesa del gradiente. Un problema che può emergere con l'utilizzo di questo metodo è che il policy gradient è soggetto a un'alta varianza [15]: la direzione scelta ad ogni iterazione può variare in base agli stati visitati e di conseguenza il costo computazionale della ricerca rischia di diventare molto alto.

- I metodi value-based, al contrario dei policy-based, non cercano di approssimare direttamente la policy, che diventa implicita. L'azione successiva viene individuata grazie alla value function che calcola l'azione più remunerativa. Questo metodo porta a una buona approssimazione della policy, ma non è detto che la soluzione trovata sia necessariamente subottimale [15].

Esiste un terzo approccio, chiamato actor-critic, che permette di prendere i punti di forza delle due tecniche. La critic usa una struttura per imparare la value function, ovvero stima quale sia la direzione migliore da prendere, mentre l'actor aggiorna i parametri della policy secondo la direzione suggeritagli dalla critic [15]. Sia critic che actor sono due reti neurali nel deep reinforcement learning.

Capitolo 3

AWS DeepRacer

AWS DeepRacer è un progetto sviluppato da Amazon Web Services per introdurre sviluppatori esperti e non ai concetti del deep reinforcement learning [23]. Il progetto si basa sulla simulazione di un'automobile da corsa che ha come obiettivo quello di imparare a guidare autonomamente su una vasta gamma di circuiti grazie alle tecniche di deep reinforcement learning. Inoltre, Amazon organizza mensilmente qualifiche e campionati grazie ai quali è possibile gareggiare contro modelli addestrati da altri utenti. Usando le parole degli sviluppatori del progetto "DeepRacer è la prima implementazione di successo su larga scala di deep reinforcement learning di un agente di controllo robotico che utilizza solamente immagini raw come osservazioni e un metodo di apprendimento model-free per eseguire un piano di sviluppo robusto" [5].

3.1 Introduzione

DeepRacer si tratta di un progetto sim2real [5]. Il modello viene addestrato in un ambiente simulato, nel quale acquisisce la conoscenza che verrà poi sfruttata nel momento in cui viene testato su un'auto reale.

Lo sviluppo di un modello si suddivide in due parti, l'addestramento e la valutazione della performance.

L'addestramente può essere solamente simulato in un ambiente virtuale che emula la fisica del mondo reale, mentre la valutazione viene sia simulata che effettuata nel mondo reale.

L'obiettivo è tagliare la linea del traguardo senza uscire dal circuito. Sono presenti tre tipologie di gare differenti:

- gara a tempo
- gara a ostacoli, o object avoidance
- gara testa a testa con altri modelli

In base alla gara scelta, varierà anche l'approccio che l'utente dovrà tenere nei confronti dell'addestramento.

Per svolgere il training, AWS mette a disposizione una console tramite la quale si possono sfruttare i servizi di cloud computing offerti da Amazon. Nonostante ciò, grazie al fatto che si sia sviluppata una community abbastanza attiva e alla presenza del codice di DeepRacer sviluppato da AWS su GitHub, sono presenti degli ambienti che permettono sia la simulazione su una macchina in locale, sia l'utilizzo di altri servizi cloud come Google Cloud e Microsoft Azure [6].

3.2 Caratteristiche dell'agente

L'agente corrisponde alla versione in scala 1:18 di una automobile da corsa, chiamata DeepRacer, che tramite l'interazione con l'ambiente, ovvero il circuito, addestra il modello.

L'automobile deve compiere un'azione in base allo stato in cui si trova in un certo momento. Per osservare lo spazio che la circonda, l'auto può utilizzare 3 diversi sensori [23]:

- una videocamera frontale grandangolare a 120° che cattura immagini RGB a 15 fps in seguito convertite in bianco e nero e ridotte a una dimensione di 160x120px.

- una videocamera stereoscopica che cattura due immagini in input con la stessa risoluzione e frequenza. Questa tipologia di videocamera permette di percepire la profondità degli oggetti catturati. Dovrebbe essere utilizzata nelle gare di object avoidance e nelle gare testa a testa. Anche queste immagini vengono convertite in una scala di grigi.

- un sensore LiDAR che attraverso degli impulsi laser permette permette all'agente di rilevare oggetti al di fuori del campo visivo della videocamera. Anche questo sensore, come la videocamera stereoscopica, viene utilizzato per l'object avoidance e per le gare testa a testa.

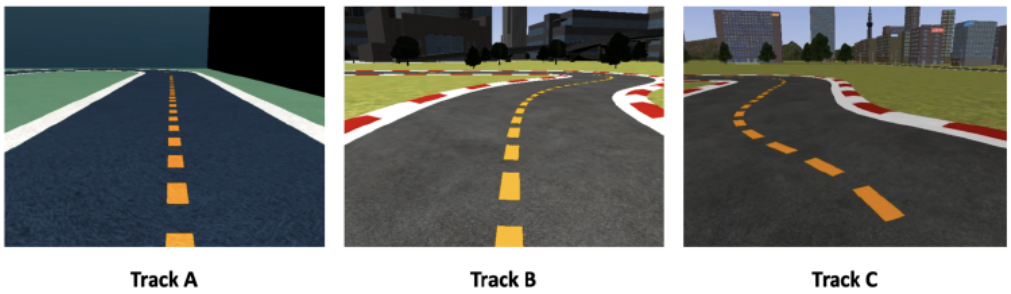

Figura 3.1: Immagini raccolte dalla videocamera frontale in un ambiente simulato [5]

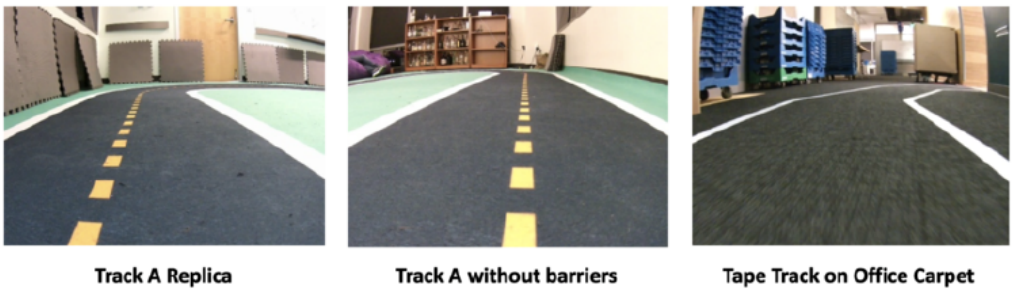

Figura 3.2: Immagini raccolte dalla videocamera frontale in un ambiente reale [5]

Le immagini e le rilevazioni tramite LiDAR che l'agente raccoglie sono le osservazioni parziali dello stato. Il modello infatti non conosce il layout totale del circuito [5]. I dati raccolti in input dai sensori vengono processati dalla rete neurale convoluzionale. E' possibile scegliere fra due tipi di CNN: una shallow con 3 layer e una deep con 5 layer. Una volta processati, la rete neurale sceglie l'azione che l'agente dovrà compiere. L'azione è una coppia che comprende velocità e angolo di sterzo; viene individuata in un insieme discreto di azioni (anche chiamato discrete action space). Avere un action space discreto implica che ci siano un numero finito sia di velocità n sia di angoli di sterzo m possibili, così da avere un numero di azioni pari a $n * m$. Nel caso di DeepRacer il range dell'angolo di sterzo è di $[-30°, +30°]$ e la velocità arriva ad un massimo di $0.8m/s$ [23]. Una volta che la rete neurale sceglie l'azione da compiere, l'agente si trova in un nuovo stato dell'ambiente, riceve una ricompensa calcolata dalla funzione di reward ed è pronto a fornire nuove osservazioni in input alla rete neurale.

3.3 PPO

Per l'addestramento del modello, DeepRacer utilizza PPO (Proximal Policy Optimization). Questo algoritmo fa parte della famiglia dei metodi policy-gradient, nei quali si ottimizza il modello intervenendo direttamente sulla policy [22]. PPO viene sfruttato attraverso un approccio actor-critic e vengono utilizzate due reti neurali durante l'addestramento: la policy network, o actor network, e la value network, anche nota come critic network [23]. La policy network prende le decisioni sulle azioni che l'agente deve compiere in base alle immagini in input, mentre la value network stima il reward cumulativo che si dovrebbe raggiungere data un'immagine in input [23].

Una caratteristica fondamentale di PPO è l'alternanza fra il prelievo di dati campione durante una quantità finita di episodi e l'ottimizzazione della policy grazie ai campioni di dati prelevati nel corso di un numero finito di epochs [22].

L'agente inizializza la policy network e dà in input alla rete i dati che raccoglie. La rete sceglie delle azioni inizialmente randomiche che vengono valutate dalla funzione di reward. Questo processo viene ripetuto per un numero di episodi scelto dall'utente. Nel corso degli episodi le osservazioni computate dal modello vengono salvate in un replay buffer. Una volta terminati gli episodi si passa all'aggiornamento della policy.

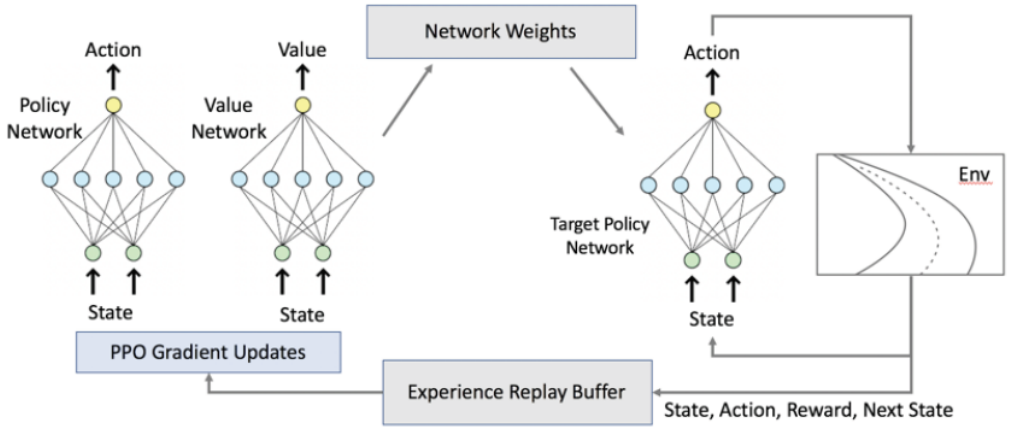

Figura 3.3: Schema di PPO [23]

La policy scorre per un numero prestabilito di volte, chiamato epoch, su un insieme di osservazioni, chiamato batch, scelte randomicamente dall'experience replay. Dopo aver analizzato più volte la batch, i pesi della rete vengono aggiornati. Essendo PPO un algoritmo actor-critic, l'aggiornamento della policy tiene conto dei valori calcolati dalla value function nel corso dell'analisi della batch. Una volta che l'aggiornamento della policy si conclude, le osservazioni presenti nell'experience buffer vengono scartate e sostituite con le nuove esperienze fatte dalla policy aggiornata nel corso dei nuovi episodi [23].

L'algoritmo per l'addestramento del modello è allo stato dell'arte e viene implementato direttamente dal team di AWS. Non è pertanto possibile modificarlo o sostituirlo da algoritmi implementati dagli utenti.

3.4 Hyperparameters

Uno dei due componenti grazie ai quali l'utente può influenzare l'addestramento del modello sono gli iperparametri (o hyperparameters) [23]. La scelta di iperparametri adeguati può influire fortemente sulla performance finale del modello [18].

Di seguito vengono illustrati quelli presenti nel contesto di PPO [23].

Batch size $\{32, 62, 128, 256, 512\}$ Rappresenta il numero di esperienze prese randomicamente dal buffer replay che devono essere esaminate prima dell'aggiornamento della policy. Se la dimensione del batch size è maggiore si ha una discesa del gradiente meno ripida, ma allo stesso tempo il modello potrebbe impiegare più tempo a convergere.

Numero di epochs $[3 - 10]$ Rappresenta quante volte il modello deve ciclare sul batch prima di aggiornare i pesi della rete.

Learning rate $[1e - 3, 1e - 8]$ Controlla la velocità di apprendimento del modello, quindi quanto il calcolo del gradiente debba influenzare l'aggiornameno dei pesi. Se il learning rate è troppo basso il modello rischia di impiegare troppo tempo per arrivare alla convergenza. Al contrario se viene settato troppo alto si rischia di non arrivare alla convergenza su una soluzione ottima.

Exploration $\{CateghoricalParameters, EpsilonGreedy\}$ Si può scegliere fra due metodi per affrontare il dilemma Exploration vs Exploitation. L'esplorazione categorica sceglie le azioni in base alla distribuzione di probabilità dell'action space. L'esplorazione epsilon sceglie fra l'esplorazione e lo sfruttamento in modo randomico.

Entropy $[0, 1]$ Rappresenta il grado di randomicità presente nel momento in cui l'agente sceglie un'azione in una data situazione. Se ha un valore alto, è più probabile che l'agente scelga di intraprendere azioni randomiche e quindi esplorare l'action space. Diventa utile nel momento in

cui il modello si ritrova bloccato sullo stesso percorso, in quanto incentiva la scoperta di nuove azioni che possono restituire dei reward maggiori.

Dicount factor $[0, 1]$ Già discusso nel Capitolo 2, permette di definire il peso dei reward locali sul reward cumulativo ricevuto al termine di ogni episodio.

Loss type $\{MeanSquaredError, HubelLoss\}$ Tramite questo hyperparameter si può scegliere il tipo di loss function che verrà utilizzata durante l'update della policy. Si differenziano nel momento in cui gli aggiornamenti diventano più grandi, infatti in questo caso Huber Loss tende a fare incrementi inferiori rispetto a Mean Squared Error Loss. Per questo motivo se ci sono problemi di convergenza è consigliato usare Huber Loss, mentre se l'obiettivo è avere un training più veloce, Mean squared error diventa la scelta più adeguata.

Numero di episodi $[5, 100]$ Rappresenta il numero di episodi da compiere tra le iterazioni di update della policy. Nel contesto di DeepRacer ogni episodio può terminare su due stati terminali: l'arrivo al traguardo o l'uscita fuori pista. In entrambi i casi la macchina viene riposizionata al centro del circuito e ricomincia l'addestramento con l'episodio successivo. Ogni episodio viene suddiviso in osservazioni, ovvero gli stati di transizione, per ognuno dei quali la funzione di reward restituisce un valore più o meno alto in base alla bontà dell'azione scelta.

3.5 Reward function

Il secondo componente col quale l'utente può influenzare l'addestramento del modello è attraverso la progettazione della reward function.

Ogni osservazione viene valutata attraverso la funzione di ricompensa che prende in input dei parametri ricevuti dall'ambiente simulato. Questi si rife-

riscono al posizionamento della macchina nello stato preso in considerazione e devono essere sfruttati per la sua implementazione.

Qui sotto vengono descritti i parametri presi in input dalla reward function nel contesto di DeepRacer, che l'utente può utilizzare per la creazione della funzione [23]:

`"all_wheels_on_track": Boolean` un flag che indica se le ruote dell'agente sono all'interno del circuito

`"x": float` coordinata sull'asse x dell'agente

`"y": float` coordinata sull'asse y dell'agente

`"closest_objects": [int, int]` vettore di due coppie (x,y) che indicano la posizione dei due oggetti più vicini all'agente

`"distance_from_center": float` distanza in metri dell'agente dal centro del circuito

`"is_crashed": Boolean` flag che indica se l'agente si è arrestato in modo anomalo a causa di un altro oggetto

`"is_left_of_center": Boolean` flag che indica se l'agente è sul lato sinistro rispetto al centro del circuito

`"is_offtrack": Boolean` flag che indica se lo stato terminale dell'agente è fuori dal circuito

`"is_reversed": Boolean` flag che indica se l'agente procede in senso orario o anti-orario

`"heading": float` imbardata dell'agente in gradi

`"objects_distance": [float, ]` lista delle distanze di tutti oggetti sul percorso dalla linea di partenza del circuito

`"objects_heading": [float, ]` lista dell'heading degli oggetti sul circuito in gradi

`"objects_left_of_center": [Boolean, ]` lista di flag che indica se l'iesimo ogetto è posizionato a destra o a sinistra della linea centrale del circuito

`"objects_location": [(float, float),]` lista di coppie che indicano le coordinate degli oggetti

`"objects_speed": [float, ]` lista delle velocità m/s degli oggetti su circuito

`"progress": float` percentuale del circuito completato dall'agente

`"speed": float` velocità dell'agente in m/s

`"steering_angle": float` angolo di sterzo dell'agente

`"steps": int` numero di azioni prese dall'agente in seguito a ogni stato

`"track_length": float` lunghezza totale del circuito

`"track_width": float` larghezza del circuito

`"waypoints": [(float, float), ]` lista delle coordinate (x,y) dei waypoint. Il circuito infatti viene suddiviso in un numero finito di waypoint equidistanti fra loro

`"closest_waypoints": [int, int]` vettore di due coppie (x,y) che indicano i due waypoint più vicini all'agente

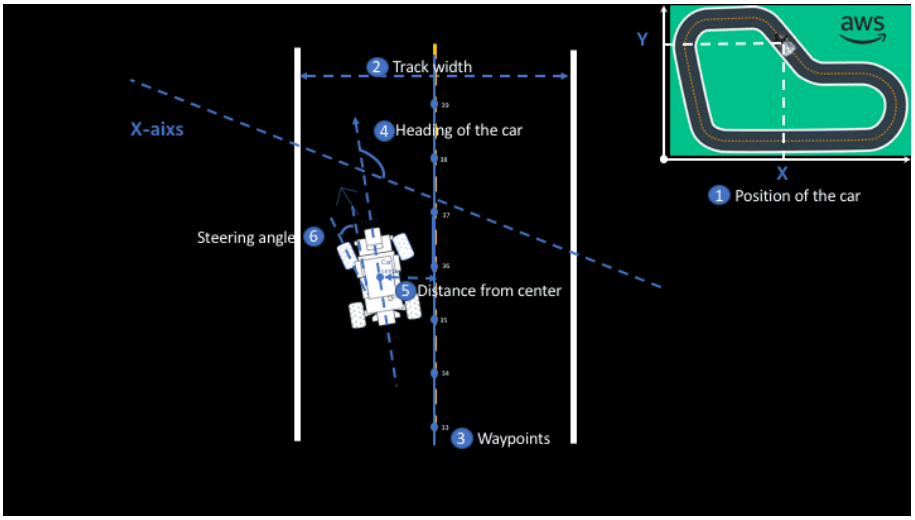

Figura 3.4: Esempio di alcuni parametri in AWS DeepRacer [23]

3.6 Addestramento in locale

La simulazione in locale permette di sfruttare la capacità computazionale del proprio pc evitando di doversi affidare a servizi di cloud computing a paga-

mento. Nonostante ciò, sono state riscontrate forti differenze a livello di velocità di convergenza del modello se lo si paragona a quello dell'addestramento tramite i servizi offerti da AWS.

Anche nell'addestramento in locale è possibile modificare solamente la reward function, gli hyperparameter, il circuito di addestramento e le reti neurali utilizzate. Avendo a disposizione il codice completo della simulazione è stato possibile tentare di modificarlo con l'obiettivo di inserire un altro algoritmo RL.

DeepRacer utilizza localmente una libreria che implementa algoritmi allo stato dell'arte, RL Coach, sviluppata da InteLabs [13]. Questi algoritmi sono integrati con Amazon SageMaker, la piattaforma messa a disposizione da AWS per lo sviluppo e l'addestramento di modelli di machine learning [4].

Le prime difficoltà sono state riscontrate fin da subito nello studio del codice in locale. A causa della struttura complessa del codice questo risulta poco leggibile. Inoltre, l'integrazione fra la libreria RL Coach e l'ambiente non permette modifiche sostanziali. E' stata tentata la sostituzione di PPO con l'implementazione di un algoritmo RL non facente parte della collezione sviluppata da RL Coach, ma la presenza di elementi hard coded non hanno permesso la buona riuscita di questo tentativo.

Un secondo problema affrontato riguarda la potenza di calcolo che il computer dell'utente deve essere in grado di generare. Infatti per la simulazione locale è fortemente consigliata una scheda grafica NVIDIA, dato che l'utilizzo di una GPU risulta più efficiente grazie alle librerie di supporto sviluppate da NVIDIA: CUDA/CuDNN [19]. Questo implica che la mancanza di questo componente può rallentare la simulazione fino a renderla impraticabile [6].

Nel corso di questa esperienza sono stati riscontrati problemi nell'utilizzo della GPU dovuti alla presenza di leak in memoria nella simulazione in locale. Infatti, dopo pochi episodi dall'inzio dell'addestramento, la GPU esauriva lo spazio disponibile andando in out of memory. Dopo vari tentativi il leak è stato trovato, probabilmente causato dalla libreria Tensorflow. Questa generava la richiesta di memoria fino all'esaurimento completo e al conseguente

fallimento della simulazione.

E' stato inoltre presentato un nuovo ambiente di simulazione dalla community, ma i requisiti minimi, fra cui la presenza di un totale fra RAM e GPU di 32GB, rendono l'utilizzo di questo nuovo ambiente non adeguato a chi non dispone di mezzi particolarmente potenti.

3.6.1 Studio delle Reward Function

Di seguito sono riportati tre esempi di reward function fra quelle utilizzate durante la sperimentazione in locale di DeepRacer. Se ne discutono gli effetti sulla simulazione anche in relazione alla modifica degli iperparametri.

Esempio reward function 1:

```
def reward_function1(params):
    # parametri in input
    track_width = params['track_width']
    distance_from_center = params['distance_from_center']
    # Calcola 3 indicatori al variare della distanza dalla linea centrale
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width
    # Viene restituito un reward maggiore se l'agente si trova vicino alla linea centrale
    if distance_from_center <= marker_1:
        reward = 1.0
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3  # agente uscito dal circuito

```

```
19      return float(reward)
```

Questo tipo di funzione è stata la prima utilizzata per testare un veicolo. L'obiettivo è dare un reward più alto nel caso in cui l'agente si posizioni nel centro della corsia del circuito. Dopo 19 ore di addestramento e 3240 episodi, l'agente usciva dal circuito meno frequentemente. Durante la fase di valutazione riusciva a completare un giro senza uscire dal circuito.

La stessa funzione è stata usata nell'addestramento di un modello tramite i servizi di cloud computing messi a disposizione da Amazon e nell'arco di un'ora, utilizzando gli stessi hyperparameter, è stato possibile ottenere risultati migliori rispetto al modello sopra descritto. Infatti durante la fase di valutazione, che nella simulazione offerta da AWS consiste nel tagliare il traguardo per tre volte consecutive, l'agente è uscito solo 2 volte dal circuito.

Esempio reward function 2:

```
20  import math
21  def reward_function2(params):
22      all_wheels_on_track = params['all_wheels_on_track']
23      #ritorna gli indici dei due waypoint vicini (il
            precedente 0 e il successivo 1)
24      closest_waypoints = params['closest_waypoints']
25      heading = params['heading']
26      speed = params['speed']
27      waypoints = params['waypoints']
28      # REWARD # In questa fase della funzione do una
            ricompensa se la macchina segue il percorso
29
30      # Inizializzo il reward con un valore standard
31      reward = 1
32
33      # recupero i waypoint vicini, sono coordinate
            spaziali x,y
34      next_waypoint = waypoints[closest_waypoints[1]] #
            successivo rispetto a dove mi trovo
```

```
35      prev_waypoint = waypoints[closest_waypoints[0]] #
            precedente rispetto a dove mi trovo
36
37      # Calcolo l'angolo della retta che passa fra i due
            waipoint
38      y_input = next_waypoint[1] - prev_waypoint[1]
39      x_input = next_waypoint[0] - next_waypoint[0]
40
41      # ritorna l'arctan di y/x, in radianti
42      track_direction = math.atan2(y_input, x_input)
43      # converto in gradi
44      track_direction = math.degrees(track_direction)
45
46      # calcolo la differenza fra la direzione del track (
            trovata con i waypoint) e l'heading della
            macchina
47      direzione_effettiva = abs(track_direction - heading)
48
49      # il valore deve essere compreso fra 0 e 180 gradi
50      if direzione_effettiva > 180:
51          direzione_effettiva = 360 - direzione_effettiva
52
53      # assegnamento del reward
54      threshold = 10.0
55      if direzione_effettiva > threshold:
56          reward = 0.5 #dimezzo il reward, non sta
                seguendo il percorso
57
58      # PENALITA' # riassegno il reward molto basso se l'
            agente fuori dal track
59      if not all_wheels_on_track:
60          reward = 1e-3
61
```

```
62      return float(reward)
```

Questo secondo esempio cerca di evitare il fenomeno zig-zag che si viene a creare nel momento in cui l'agente viene premiato solamente se rimane nella zona centrale della corsia del circuito. Ad ogni step si considerano i due waypoint più vicini e se ne calcola la traiettoria. Questa viene confrontata con l'heading dell'agente. Se la differenza fra la direzione verso cui punta l'agente e la direzione appena trovata supera un certo threshold allora l'agente viene penalizzato. Inoltre nel caso in cui l'agente esca dal circuito, gli viene assegnato un reward molto basso.

Questo tipo di reward function non indirizza verso il percorso più veloce, ma punta a mantenere l'agente al centro della pista.

Applicandola al modello addestrato localmente di cui si è discusso precedentemente, si è potuto notare inizialmente un peggioramento della performance, che però è stato seguito da un netto miglioramento del modello. L'andamento era più lineare e usciva meno frequentemente dal circuito.

Esempio reward function 3:

```
64  import math
65  old_progress = 0
66
67  def reward_function3(params):
68      global old_progress
69      #parametri in input
70      all_wheels_on_track = params['all_wheels_on_track']
71      distance_from_center = params['distance_from_center'
            ]
72      track_width = params['track_width']
73      new_progress = params['progress']
74
75      # Inizializzo il reward con un valore standard
76      reward = 0
77      #DISTANZA DAI BORDI E DAL CENTRO
78      # Penalizza l'agente se si allontana dal centro
```

```
79      marker_1 = 0.1 * track_width
80      marker_2 = 0.5 * track_width
81
82      if distance_from_center <= marker_1:
83          reward += 0.5
84      elif distance_from_center <= marker_2:
85          reward += 0.1
86      else:
87          reward = 1e-3
88      #PROGRESSO
89      # Viene considerato il nuovo progresso
90      progress = new_progress - old_progress
91
92      if(progress > 0):
93          reward *= (1 + progress)
94      else:
95          reward = 1e-3   # no progresso
96
97      old_progress = new_progress
98      #DENTRO AL CIRCUITO
99      # In questa fase della rew fun controllo se la
           macchina si trova dentro al circuito
100     if all_wheels_on_track:
101         reward += 0.2
102     else:
103         reward = 0
104
105     return float(reward)
```

Questa funzione penalizza sia la distanza dell'agente dal centro sia l'uscita dal circuito, ma considera il progresso che viene fatto di stato in stato dall'agente, ovvero la percentuale di circuito completata da esso. Viene definita una variabile globale alla quale viene assegnata la differenza fra il progresso

dello stato attuale e quello precedente. Se nell'iterazione successiva, il progresso aumenta più velocemente si moltiplica il reward locale alla differenza fra progresso attuale e progresso precedente.

Questa funzione è stata applicata sia a un modello già addestrato che si manteneva in corsia, sia a un modello nuovo. Nel primo caso il modello non ha apportato particolari migliorie alla performance. Nel secondo caso invece il modello non ha dato buoni risultati. Concentrarsi subito sulla velocità potrebbe aver influito sul modello in modo che venisse data priorità alla velocità, tralasciando la ricerca di un cammino accurato che lo potesse mantenere sul percorso il più a lungo possibile.

Conclusioni

DeepRacer è un progetto sviluppato da Amazon Web Services per introdurre gli sviluppatori al mondo del deep reinforcement learning. Nonostante questa sua prerogativa risulta essere un ambiente decisamente limitante se si considera come obiettivo finale lo sviluppo di un algoritmo che sfrutti il deep reinforcement learning. Infatti il team di AWS ha già creato un ambiente ottimale per l'addestramento di modelli di auto da corsa che sfruttano algoritmi allo stato dell'arte, come PPO [5], lasciando agli utenti come unico compito quello di personalizzare componenti come gli iperparametri e la reward function. Nonostante ciò possa aiutare a comprendere i principi fondamentali e i meccanismi attorno a cui ruota il deep reinforcement learning, l'interazione tende a essere trattata più come gioco piuttosto che come strumento utile a studenti o sviluppatori intenzionati a cimentarsi nell'implementazione di algoritmi RL.

DeepRacer è un progetto ancora giovane, lanciato verso la fine del 2018 [23] e non è detto che non verrà implementata anche questa feature in futuro dal momento che esiste una community crescente e che nell'ultimo periodo sono state aggiunte due nuove feature, un altro algoritmo SAC e un nuovo modello che utilizza action space continuo [23].

Però, tenendo in considerazione le ultime osservazioni, il livello di complessità del codice per la simulazione in locale rimane tale da rendere questa opzione poco probabile.

Esistono altre alternative per lo sviluppo di algoritmi RL molto più versatili. Uno fra tutti è TORCS, The Open Racing Car Simulator, un simulatore

3D di gare automobilistiche open source per il quale sono stati sviluppati plug-in che permettono la completa personalizzazione di tutti i componenti, in modo da avere un maggiore controllo sulla simulazione.

Sicuramente AWS DeepRacer è un progetto interessante, ma nello stato in cui si trova attualmente non soddisfa le caratteristiche che potrebbero renderlo un ambiente di sviluppo e testing adeguatodi algoritmi per deep reinforcement learning.

Bibliografia

[1] Abdulrahman Alassadi and Tadas Ivanauskas. Classification performance between machine learning and traditional programming in java, 2019.

[2] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[3] A. Asperti. Deep learning. *PDF Lesson 1*, 2019/2020. Università di Bologna.

[4] AWS. *Amazon SageMaker*. https://aws.amazon.com/it/sagemaker/.

[5] B. Balaji, S. Mallya, S. Genc, S. Gupta, L. Dirac, V. Khare, G. Roy, T. Sun, Y. Tao, B. Townsend, E. Calleja, S. Muralidhara, and D. Karuppasamy. Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2746–2754, 2020.

[6] AWS DeepRacer Community. *AWS DeepRacer Wiki*. https://wiki.deepracing.io/Main_Page.

[7] Ashkan Eliasy and Justyna Przychodzen. The role of ai in capital structure to enhance corporate funding strategies. *Array*, 6:100017, 2020.

[8] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. 2018.

[9] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] ICHI.PRO. *Funzioni di attivazione*. https://ichi.pro/it/funzioni-di-attivazione-99956793768697.

[13] IntelLabs. *coach*. https://github.com/IntelLabs/coach.

[14] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009.

[15] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014. Citeseer, 2000.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[17] Jing Li, Ji-hang Cheng, Jing-yuan Shi, and Fei Huang. Brief introduction of back propagation (bp) neural network algorithm and its improvement. In David Jin and Sally Lin, editors, *Advances in Computer Science and Information Engineering*, pages 553–558, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] Gang Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):1–16, 2016.

[19] NVIDIA. *NVIDIA cuDNN*. https://developer.nvidia.com/cudnn.

[20] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[21] Warren S. Sarle. Neural networks and statistical models, 1994.

[22] John Schulman, F. Wolski, Prafulla Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[23] Amazon Web Services. *AWS DeepRacer Developer Guide*, 2021. https://docs.aws.amazon.com/deepracer/latest/developerguide/.

[24] Marco Somalvico et al. *Intelligenza artificiale*. Scienza & vita nuova, 1987.

[25] Manli Sun, Zhanjie Song, Xiaoheng Jiang, Jing Pan, and Yanwei Pang. Learning pooling for convolutional neural network. *Neurocomputing*, 224:96–104, 2017.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[27] Shiyin Wei, Yuequan Bao, and Hui Li. Optimal policy for structure maintenance: A deep reinforcement learning framework. *Structural Safety*, 83:101906, 2020.

[28] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Ringraziamenti

Vorrei ringraziare il professore Asperti che mi ha dato la possibilità di affrontare un argomento al quale ero completamente estranea ma che mi ha appassionata molto fin da subito. Spero di continuare questo percorso che ho cominciato tramite tirocinio e tesi, nel mondo del machine learning.

Vorrei ringraziare mia sorella Anna, che nonostante lo stress che continuo a procurarle, mi supporta sempre nei momenti cruciali e i miei genitori, che mi hanno permesso di intraprendere questo percorso.

Una speciale menzione per Angelica che malgrado i chilometri di distanza è sempre stata una costante. E soprattutto, dopo aver studiato cozze e ingegnerie improbabili, abbiamo finalmente trovato la nostra strada.