Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Scuola Politecnica e delle Scienze di Base

Corso di Laurea Triennale in Ingegneria Informatica

DDPG design for lane keeping problem in TORCS environment

Relatori:
Prof.ssa Santini Stefania
Correlatore:
Dott.Ing. Petrillo Alberto

Candidato:
del Gaudio Raffaele
Matr. N46004100

Anno Accademico
2020/2021

Indice

Elenco delle figure **iv**

Elenco delle tabelle **v**

1 Dal RL al Deep RL **3**
1.1 Reinforcement Learning . . . 3
1.1.1 Il modello del RL . . . 4
1.1.2 Q-Learning . . . 6
1.2 Deep Reinforcement Learning . . . 7
1.2.1 Metodi Value Based . . . 7
1.2.1.1 Deep Q-Network . . . 7
1.2.2 Metodi Policy Based . . . 9
1.2.2.1 REINFORCE . . . 9
1.2.3 Metodi Actor-Critic . . . 11
1.2.3.1 DDPG . . . 12

2 Problem Statement **15**
2.1 Introduzione a TORCS . . . 16
2.2 Segnali di interazione con l'ambiente . . . 18

3 Design del Controllo **19**
3.1 Architettura delle reti . . . 21
3.2 Reward Shaping . . . 23
3.3 Rumore Esplorativo . . . 24
3.3.1 Ornstein-Uhlenbeck Mod . . . 25
3.3.2 Gaussian Noise . . . 25
3.3.3 Time Variant Noise . . . 26

4 Simulazioni e Risultati **27**
4.1 Set-up Numerico . . . 27
4.1.1 Panoramica dei modelli allenati . . . 29

4.2 Scelta del modello DDPG 30
4.2.1 Modalità del confronto tra modelli 30
4.2.2 Test sul tipo di rumore 31
4.2.3 Test sul tipo di reward function 33
4.2.4 Modello finale 34
4.3 Risultati numerici 35

Bibliografia **38**

Elenco delle figure

1.1 Modello del RL 4
1.2 Topologia della Deep Q-Network 7
1.3 Policy Network 9
1.4 Modello dell'Actor-Critic 11
1.5 Algoritmo DDPG 14

2.1 Immagine di una gara in TORCS 16
2.2 Architettura di SCR 17

3.1 Modello del sistema TORCS-Agente 20
3.2 Rete Actor 21
3.3 Rete Critic 22
3.4 Andamento della media Gaussiana 26

4.1 Tracciati di training e validazione 27
4.2 Statistiche del Modello 1 31
4.3 Statistiche del Modello 2 32
4.4 Statistiche del Modello 3 32
4.5 Statistiche del Modello 4 33
4.6 Statistiche del Modello 5 33
4.7 Reward nell'episodio 35
4.8 Velocità nell'episodio 36
4.9 Errore quadratico trackPos nell'episodio 36

Elenco delle tabelle

2.1 Sensori utilizzati nella costruzione del vettore di stato 18
2.2 Azioni eseguibili dall'agente 18

4.1 Parametri del modello 28
4.2 Metriche di giudizio 28
4.3 Valore predefinito dei parametri immutati 29
4.4 Modelli e rispettivi parametri 29
4.5 Confronto Modelli 1, 2 e 3 31
4.6 Confronto Modelli 1, 4 e 5 34

Introduzione

La storia ci insegna che l'uomo si è da sempre impegnato per automatizzare e controllare i processi che fanno parte della sua vita. Si pensi al caso dell'aratro antico che permetteva di smuovere il terreno utilizzando il lavoro di un animale piuttosto che quello del contadino, oppure a quello del mulino che, sfruttando vento o acqua, permetteva la macinazione del grano senza necessità di sforzi fisici.

Nell'era dell'Informatica e dei Big Data questo impegno non si è ridotto, anzi è stato traslato anche verso processi che fino a pochi anni fa si ritenevano ad esclusivo appannaggio degli esseri umani.

Se oggi è possibile automatizzare processi come la classificazione di immagini, convertire testo in codice programmabile o effettuare interpolazione di frame nei video è sopratutto grazie alla grande evoluzione che l'Intelligenza Artificiale sta avendo negli ultimi anni.

Anche in materia di Controlli non mancano esempi di utilizzo delle Neural Network, come nel caso della modellazione del catalizzatore a tre vie in ambito Automotive dove una rete neurale è stata utilizzata per modellare le velocità delle reazioni chimiche che avvengono al suo interno, o del controllo della dinamica non lineare di un missile con una CMAC.

In questa tesi si affronta uno dei principali problemi dell'Autonomous Driving, il lane keeping, ovvero il problema del mantenimento della carreggiata per un'auto a guida autonoma. Il problema viene affrontato nell'ambiente di simulazione di guida TORCS utilizzando una tecnica di Reinforcement Learning.

Il lavoro è strutturato come segue. Nel Capitolo 1 si presenta il Reinforcement Learning e si fa una panoramica del Deep Reinforcement Learning, con una distinzione tra i metodi Value Based, Policy Based e Actor-Critic. Nel Capitolo 2 si definisce il problema del lane keeping nel caso di studio specifico. Nel Capitolo 3 si affronta l'approccio utilizzato, ovvero il design del controllore scelto. Nel Capitolo 4 verranno comparati i risultati ottenuti dalle diverse architetture di controllo sperimentate.

Capitolo 1

Dal RL al Deep RL

1.1 Reinforcement Learning

Gli esseri umani per la maggior parte della loro vita sono immersi in un ambiente facendo esperienza dell'interazione con esso e registrando nel proprio cervello una grande quantità di informazioni rispetto a cause ed effetti, alle conseguenze delle proprie azioni e su come agire per raggiungere degli obiettivi. Questa forma di insegnamento, che è l'interazione con l'ambiente in se, è in assoluto quella predominante nella vita degli individui, solo in piccola parte accompagnata da quello supervisionato. Che si tratti di imparare a guidare o tenere una conversazione, siamo sempre molto attenti a come l'ambiente risponde alle nostre azioni con l'obiettivo di influenzare cosa succede attraverso esse.[6]

Il Reinforcement Learning fonda le proprie basi teoriche ispirandosi proprio a questo aspetto del comportamento umano.

Il Reinforcement Learning è la branca dell'Intelligenza Artificiale che si preoccupa di studiare il problema di un agente che deve apprendere un comportamento attraverso tentativi ed errori, interagendo con l'ambiente in cui è immerso. Il tutto senza che gli venga specificato come agire ma solamente attraverso un meccanismo di ricompense e punizioni.

1.1.1 Il modello del RL

Il modello standard del Reinforcement Learning è composto da un agente che è in comunicazione con un ambiente attraverso percezioni e azioni. Come si vede dalla fig.1.1, in ogni istante di tempo t l'agente riceve un insieme di percezioni s_t e una ricompensa r_t dall'ambiente e sceglie un'azione a_t da eseguire. Tale azione cambia lo stato dell'ambiente, il quale emette una nuova ricompensa per l'agente.

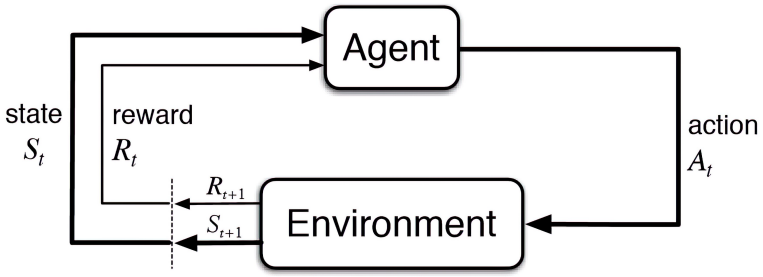

Figura 1.1: Modello del RL

La funzione che in ogni istante di tempo associa ad uno stato una distribuzione di probabilità sull'insieme delle azioni si chiama *policy* e usualmente viene indicata col simbolo π.

$$\pi : s \to \mathscr{D} \tag{1.1}$$

Per misurare la bontà di un'azione a_t eseguita in uno stato s_t viene definita la *action-value function* Q associata alla policy π

$$Q^{\pi}(s_t, a_t) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k}\Big] \tag{1.2}$$

mentre per misurare il valore di uno stato s_t si definisce la *state-value function* V associata alla policy π

$$V^{\pi}(s_t) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k}\Big] \tag{1.3}$$

Dove $\gamma \in [0, 1]$ è chiamato *discount factor* e controlla come l'agente valuta le ricompense future nell'ambito dei problemi ad orizzonte infinito[2]. Bassi valori favoriscono un comportamento teso a massimizzare le ricompense a breve termine mentre alti valori quelle a lungo termine.

Q è calcolata come la media pesata rispetto a potenze di γ delle ricompense future ottenute dall'agente in ottemperanza alla policy π. L'eq.1.2 è comunemente espressa nella forma di Bellman come segue

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma\, Q^{\pi}(s_{t+1}, a_{t+1}) \tag{1.4}$$

Dove $r(s_t, a_t)$ è la ricompensa ottenuta nello stato di arrivo s_{t+1} eseguendo l'azione a_t nello stato s_t. Per tale motivo verrà indicata anche come r_{t+1}.

La tupla $(s_t, a_t, s_{t+1}, r_{t+1})$ viene detta *transizione* e rappresenta l'unità fondamentale dell'apprendimento per gli algoritmi di RL nell'ambito dei *Markov Decision Process*.

L'obiettivo dell'agente è imparare una *policy ottima* π^*, a cui corrisponde una *action-value function* ottima Q^{π^*} che calcola il valore di ogni stato s come la somma della ricompensa istantanea e del valore scontato dello stato successivo eseguendo *la migliore azione possibile*

$$Q^{\pi^*}(s_t, a_t) = r_{t+1} + \gamma \max_a Q^{\pi^*}(s_{t+1}, a_{t+1}),\ \forall s \tag{1.5}$$

Data la 1.5, la policy ottima si determina come segue

$$\pi^*(s_t) = \arg\max_a Q^{\pi^*}(s_t, a_t) \tag{1.6}$$

1.1.2 Q-Learning

Il Q-Learning[10] è uno degli algoritmi model-free[1] più conosciuti del RL e fa parte della famiglia dei Temporal Difference[2].

Il cuore dell'algoritmo, che usa la eq.1.2 come action value function, è la funzione di aggiornamento implementata dall'agente:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{1.7}$$

Dove $\alpha \in [0, 1]$ è il *learning rate*, ovvero il tasso con il quale i valori di Q sono aggiornati ad ogni step, e γ è quello definito in precedenza. Sotto l'ipotesi che tutte le azioni sono campionate un numero sufficiente di volte in tutti gli stati e i valori stato-azione sono rappresentati in modo discreto si dimostra[9] che il Q-Learning converge con *probabilità 1* alla action-value function ottima Q^*.

[1] Classe di algoritmi che non fa utilizzo del modello del processo da ottimizzare ma si basa sul trial-and-error.

[2] Famiglia di algoritmi basati su una ottimizzazione dinamica della stima della action-value function basata sulle osservazioni correnti.

1.2 Deep Reinforcement Learning

Con l'evoluzione delle *Deep Neural Network* è stato possibile rielaborare i classici algoritmi del RL e di formularne di nuovi. I benefici dell'utilizzo delle DNN come approssimatori di funzioni non lineari sono la possibilità di creare agenti più robusti alle variazioni ambientali e di gestire spazi di stato di dimensioni molto maggiori senza tante complicazioni sulla velocità computazionale.

1.2.1 Metodi Value Based

Questa famiglia di metodi si basa sulla stima della action-value function ottima Q^* come base per poi ricavare una policy ottima π^*.

1.2.1.1 Deep Q-Network

Il DQN[5] è una variante del Q-Learning che implementa una DNN per approssimare la action-value function di eq.1.2.

Come si vede dalla fig.1.2 la rete riceve in ingresso lo stato attuale e presenta in uscita il *valore* di ogni possibile azione. Proprio per il fatto che il numero di neuroni in uscita ad una rete neurale non può essere infinito il DQN è utilizzabile solo nel caso di insiemi di azioni numerabili.

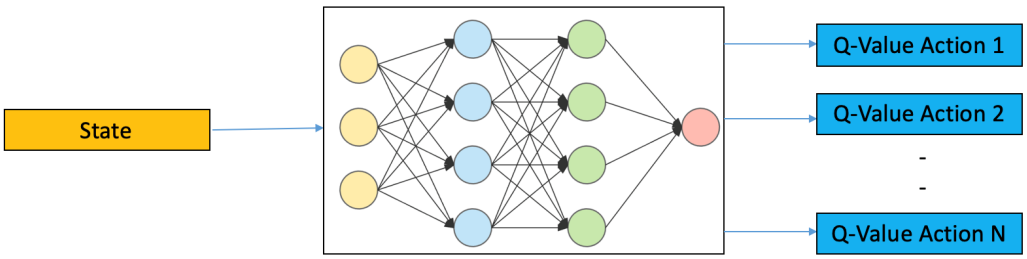

Figura 1.2: Topologia della Deep Q-Network

Per risolvere il problema della divergenza del RL nel caso di utilizzo di approssimatori di funzioni non lineari[8], gli autori del DQN prevedono l'utilizzo di un *experience replay*, ovvero un buffer dove vengono salvate le transizioni. L'obiettivo di tale aggiunta è che prelevando in maniera casuale le transizioni dal replay buffer si risolve il problema della correlazione tra la sequenza di osservazioni che è una delle cause della divergenza.

Ad ogni time step i si prelevano casualmente dall'experience replay un batch D di transizioni e si aggiorna la rete Q seguendo il gradiente della *loss function* di equazione:

$$L(\theta_i) = \mathbb{E}_D \left[\left(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1} \,|\, \theta_i') - Q(s_t, a_t \,|\, \theta_i) \right)^2 \right] \tag{1.8}$$

dove $Q(s_t, a_t \,|\, \theta_i)$ è la rete principale con parametri θ_i e $Q(s_{t+1}, a_{t+1} \,|\, \theta_i')$ è una seconda rete, detta *target network*, che approssima la action-value function dello stato successivo. I suoi parametri θ_i' vengono aggiornati con una copia di quelli della main Q-network solo ogni C iterazioni e mantenuti fissi tra gli aggiornamenti.

1.2.2 Metodi Policy Based

In opposizione ai metodi Value Based, questi mirano a stimare direttamente la policy ottima π^*. Possiamo categorizzarli in *Stochastic Policy Based* o *Deterministic Policy Based*. Il primo tipo di algoritmi produce una distribuzione di probabilità su un insieme numerabile di azioni, mentre il secondo determina direttamente il valore dell'azione da eseguire, e quindi è applicabile a processi con spazi di azioni continui.

1.2.2.1 REINFORCE

REINFORCE[11] è un algoritmo Stochastic Policy Based che utilizza una DNN per implementare una policy π. Come si evince dalla fig.1.3 la rete produce una distribuzione di probabilità sull'insieme di azioni $[a_1, a_2, ..., a_k]$ a partire da una osservazione $[s_1, s_2, ..., s_N]$.

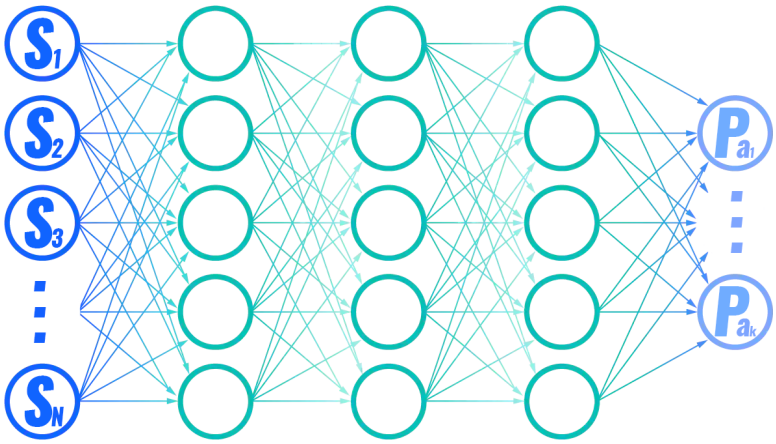

Figura 1.3: Policy Network

Il cuore dell'algoritmo è il processo di aggiornamento dei parametri θ della rete, che viene eseguito a fine di ogni *episodio*[3], nella direzione del gradiente della funzione di performance $J(\theta)$ di eq:

$$J(\theta) = \sum_{t=0}^{T-1} \gamma^t \, r_{t+1} \tag{1.9}$$

[3]Un episodio è un insieme di step eseguiti dall'agente fino al raggiungimento di una delle possibili condizioni di terminazione prefissate.

dove T è il numero di step che compongono l'episodio e il gradiente ha eq:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(t) \tag{1.10}$$

con G funzione di ricompensa cumulativa scontata di eq:

$$G(t) = \sum_{k=0}^{T-t-1} \gamma^k\, r_{t+k+1} \tag{1.11}$$

Ulteriori dettagli sul processo di calcolo della eq.1.10 sono reperibili qui[12].

1.2.3 Metodi Actor-Critic

Questi metodi sono un ibrido che combinano i benefici dei metodi Value Based e di quelli Policy Based. Consistono nell'utilizzo di due DNN, una per la stima della policy π, nominata *Actor* e una per la stima della *state value function* V, nominata *Critic*. Nella fig.1.4 si può osservare il modello.

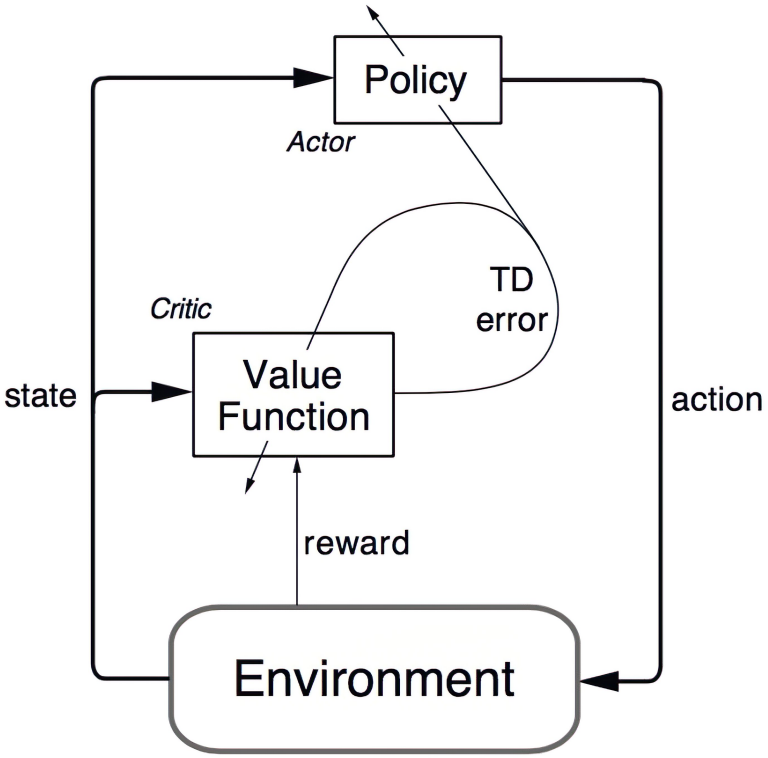

Figura 1.4: Modello dell'Actor-Critic

Ad ogni step t dell'episodio l'Actor produce un'azione a_t che modifica lo stato dell'ambiente. Il nuovo stato s_{t+1} e la ricompensa associata r_{t+1} vengono poi recepiti dal Critic che ne valuta la bontà attraverso la state-vale function $V_\theta(s_{t+1})$ di eq.1.3. Si tenga in considerazione che il Critic mantiene in memoria anche $V_\theta(s_t)$ derivante dallo stato precedente.

L'aggiornamento delle due reti viene fatto ogni step. I parametri θ della rete Critic vengono aggiornati nella direzione della minimizzazione della seguente funzione di costo:

$$\delta^2 = \Big[r_{t+1} + \gamma\, V_\theta(s_{t+1}) - V_\theta(s_t)\Big]^2 \tag{1.12}$$

mentre i parametri ϕ dell'Actor vengono aggiornati per minimizzare la funzione di costo:

$$X = \delta \log \pi_\phi(a_t|s_t) \tag{1.13}$$

1.2.3.1 DDPG

Il Deep Deterministic Policy Gradient (DDPG)[3] è un algoritmo model-free e off-policy[4], facente parte della famiglia degli Actor-Critic, che permette ad un agente di imparare una Q-function e una policy *specificatamente* in uno spazio di stato e di azione continuo.

Questo algoritmo è strettamente connesso al Q-learning ed è motivato allo stesso modo: se si conosce la action-value function ottima $Q^*(s, a)$ allora, in ogni dato stato, la policy ottima può essere trovata risolvendo l'eq.1.6.

Quando c'è un numero finito di azioni, il calcolo del massimo non pone problemi perché si possono calcolare i Q-value per ogni azione separatamente e successivamente confrontarli. Ma quando lo spazio delle azioni è continuo, non è possibile valutare in modo esauriente il massimo tra i Q-value in quanto si sarebbe costretti a campionare lo spazio d'azione e questo porterebbe inevitabilmente ad errori di valutazione. Inoltre l'utilizzo di un normale algoritmo di calcolo del massimo renderebbe la subroutine estremamente inefficiente all'aumentare del numero delle azioni.

Tuttavia poiché lo spazio delle azioni è continuo, si presume che la funzione $Q^*(s, a)$ sia differenziabile rispetto all'azione a. Questo consente di impostare un apprendimento basato sul gradiente per una policy $\mu(s)$. Quindi, invece di eseguire una costosa subroutine di ottimizzazione ogni volta che si desidera calcolare $\max_a Q(s, a)$, è possibile approssimarlo con $Q(s, \mu(s))$.

Il modello prevede l'utilizzo di due coppie di DNN:

- Una coppia di reti principali: $\mu_\phi(s)$, chiamata Actor, che sarà utilizzata per stimare la policy ottima e $Q_\theta(s, a)$, chiamata Critic, che stimerà la action-value function ottima.
- Una seconda coppia di reti: $\mu_{\phi_{targ}}(s)$ e $Q_{\theta_{targ}}(s, a)$, dette *Target Actor* e *Target Critic*, che saranno utilizzate per il calcolo della loss function

[4]Sono off-policy gli algoritmi che aggiornano i loro parametri basandosi su policy vecchie.

che permetterà l'aggiornamento dei parametri delle prime due. Queste reti hanno la stessa struttura di quelle principali e vengono aggiornate periodicamente attraverso un *soft update* in base ai loro parametri.

Le reti target vengono adoperate in modo che il valore atteso target non sia dipendente dai parametri utilizzati nelle reti principali che calcolano il valore atteso attuale. In questo modo si evitano grosse divergenze tra i due valori migliorando la stabilità durante laddestramento. Come nel DQN, il DDPG utilizza un experience replay per campionare le esperienze passate e aggiornare i parametri della rete neurale.

Dato un insieme D di transizioni prelevate dall'experience replay, l'aggiornamento dei parametri della $Q_\theta(s,a)$ viene fatta nella direzione di minimizzazione del mean-squared Bellman error (MSBE), che quantifica in che misura Q_θ si avvicina a soddisfare l'equazione di Bellman:

$$L(\theta) = \mathbb{E}_D \left[\left(Q_\theta(s_t, a_t) - \Big(r_{t+1} + \gamma(1-d) Q_{\theta_{targ}}(s_{t+1}, \mu_{\phi_{targ}}(s_{t+1})) \Big) \right)^2 \right] \quad (1.14)$$

con d un flag booleano posto ad 1 quando s_{t+1} è uno stato terminale, in modo che la Q-function non attribuisca ulteriore valore all'azione se non quella conferita dalla ricompensa immediata r_{t+1}.

L'aggiornamento dei parametri dell'Actor è più intuitivo in quanto viene fatto seguendo la massimizzazione della funzione $Q_\theta(s_t, \mu_\phi(s))$ rispetto i sui parametri ϕ, e quindi nella direzione del gradiente:

$$\nabla_\phi \, Q_\theta(s_t, \mu_\phi(s)) = \frac{\partial}{\partial \mu_\phi} \, Q_\theta(s_t, \mu_\phi(s)) \, \frac{\partial}{\partial \phi} \, \mu_\phi(s) \quad (1.15)$$

I parametri delle reti target vengono aggiornati seguendo un *soft update*:

$$\begin{aligned} \theta_{targ} &\leftarrow \tau\theta + (1-\tau)\theta_{targ} \\ \phi_{targ} &\leftarrow \tau\phi + (1-\tau)\phi_{targ} \end{aligned} \quad (1.16)$$

dove τ è detto *smooth factor* e generalmente è molto piccolo per garantire un aggiornamento lento delle reti target.

Di seguito viene mostrato lo pseudocodice dell'algoritmo DDPG[3]:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figura 1.5: Algoritmo DDPG

Capitolo 2

Problem Statement

Per risolvere il problema dell'Autonomous Driving (AU) di un veicolo terrestre, esso, normalmente, viene scomposto in diversi sotto problemi, ognuno dei quali viene risolto da uno specifico processo. Ad esempio, l'*Obstacle Avoidance* è il processo che risolve la gestione degli ostacoli presenti sul percorso durante la guida, oppure l'*Overtaking* è quello che si occupa di gestire il sorpasso di altri veicoli.

Tutti questi processi, però, fondano il loro corretto funzionamento sul fatto che, in principio, l'auto si trovi correttamente in carreggiata durante tutto il processo di guida. Il processo che risolve questo fondamentale problema viene detto *Lane Keeping*.

I tradizionali sistemi di AU sono basati su regole precostruite e devono creare informazioni sulla struttura dalle scene prima di prendere una decisione. Le informazioni sulla struttura sono molto complicate, il che porta alla costruzione di regole complesse e quindi ad una difficile costruzione di un modello decisionale completo ed efficace. Con gli ultimi sviluppi dell'Intelligenza Artificiale nell'AU è stato possibile affidare la comprensione di scene complesse e il processo decisionale alle reti neurali senza bisogno di regole definite dall'uomo. In queste circostanze è comune parlare di sistema decisionale end-to-end[1].

L'obiettivo di questa tesi è quello di affrontare il problema del lane keeping nell'ambiente avanzato di simulazione di guida TORCS[7], attraverso un sistema decisionale end-to-end basato sul DDPG. Il lavoro si ispira a quello dei ricercatori della *Beijing University of Technology*[1].

[1]Un sistema che produce una decisione direttamente a partire dalla comprensione dell'ambiente, senza passaggi intermedi.

2.1 Introduzione a TORCS

TORCS[7] è un simulatore 3D di gare automobilistiche open source. Esso è progettato per permettere a delle IA pre-programmate di gareggiare le une contro le altre e permette ad un giocatore di utilizzare una tastiera, un mouse, o un volante per controllare un veicolo.

Figura 2.1: Immagine di una gara in TORCS

TORCS si presenta come un'applicazione autonoma in cui i bot vengono compilati come moduli separati che vengono caricati nella memoria principale quando si svolge una gara. Questo presenta diversi svantaggi sul profilo della realizzazione in quanto impone la modalità di interfacciamento col gioco e il linguaggio di programmazione da utilizzare.

Nell'ambito di questa tesi si è utilizzata la versione di TORCS con patch SCR.

Simulated Car Racing (SCR)[4] estende l'architettura TORCS originale sotto tre aspetti. Innanzitutto struttura TORCS come un'applicazione Client-Server: i bot vengono eseguiti come processi esterni connessi al server di gara tramite connessioni UDP.

In secondo luogo, aggiunge il real-time: ad ogni tic di gioco (corrispondente all'incirca a 20 ms di tempo simulato), il server invia gli input sensoriali correnti a ciascun bot e poi attende 10ms (di tempo reale) per ricevere un'a-

zione dal bot. Se non arriva nessuna azione la simulazione continua e viene utilizzata l'ultima azione eseguita.

Infine SCR costruisce un livello d'astrazione che garantisce una separazione fisica tra il codice dell'AI di guida e il server di gara, *un modello di sensori e attuatori*, che lascia completa libertà di scelta in merito al linguaggio di programmazione utilizzato per i bot.

Nella figura seguente viene mostrata l'architettura di SCR.

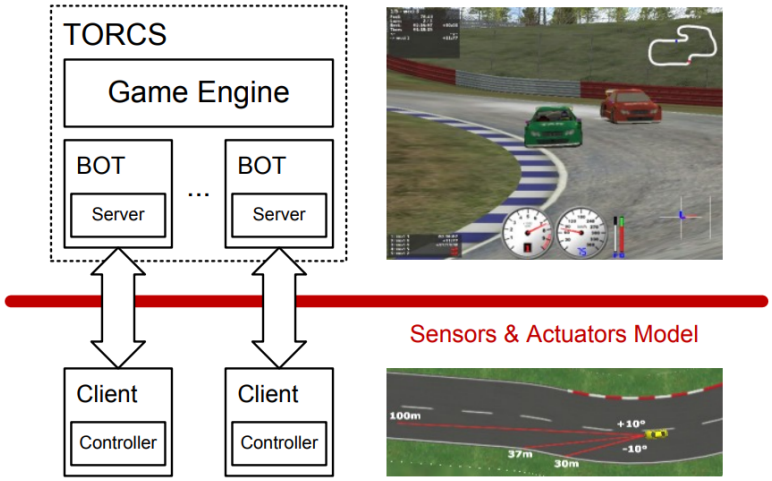

Figura 2.2: Architettura di SCR

La lista completa dei sensori e degli attuatori forniti da SCR non è riportata per brevità ma è consultabile in[4].

2.2 Segnali di interazione con l'ambiente

In questo lavoro si presenta un agente capace di terminare il percorso di gara, rimanendo sempre in carreggiata, utilizzando un sottoinsieme dei sensori forniti da TORCS. Nella tab.2.1 sono definiti i sensori utilizzati per costruire lo stato s mentre nella tab.2.2 gli attuatori, ovvero l'azione a che l'agente può eseguire sull'ambiente.

Nome	**Range (unità)**	**Descrizione**
angle	$[-\pi, +\pi](rad) \in \mathbb{R}$	Angolo tra la direzione dell'auto e l'asse della carreggiata
speedX	$[0, 300](km/h) \in \mathbb{R}$	Velocità dell'auto lungo il suo asse longitudinale
speedY	$[0, 300](km/h) \in \mathbb{R}$	Velocità dell'auto lungo il suo asse trasversale
speedZ	$[0, 300](km/h) \in \mathbb{R}$	Velocità dell'auto lungo il suo asse Z
track	$[0, 200](m) \in \mathbb{R}^{19}$	Vettore di 19 rangefinder disposti a diverse angolazioni davanti l'auto: ogni sensore misura la distanza tra l'auto e il bordo della carreggiata in un range di 200 metri
trackPos	$[-1, +1] \in \mathbb{R}$	Distanza normalizzata tra l'auto e l'asse della carreggiata

Tabella 2.1: Sensori utilizzati nella costruzione del vettore di stato

Nome	**Range (unità)**	**Descrizione**
accel	$[0, 1] \in \mathbb{R}$	Pedale del gas virtuale (0 significa accelerazione nulla e 1 accelerazione massima)
brake	$[0, 1] \in \mathbb{R}$	Pedale del freno virtuale (0 significa frenata nulla e 1 frenata massima)
steer	$[-1, +1] \in \mathbb{R}$	Sterzo virtuale (-1 significa sterzata massima a destra e +1 sterzata massima a sinistra) Sterzata massima di $0,366519\,rad$

Tabella 2.2: Azioni eseguibili dall'agente

Nella modellazione si è scelto di direzionare i 19 rangefinder nelle seguenti angolazioni: (-45, -19, -12, -7, -4, -2.5, -1.7, -1, -0.5, 0, 0.5 , 1, 1.7, 2.5, 4, 7, 12, 19, 45) gradi. In definitiva lo spazio di stato dell'agente è $s \in \mathbb{R}^{24}$ e quello di azione è $a \in \mathbb{R}^3$.

Capitolo 3

Design del Controllo

Il modello del controllo scelto si basa sul DDPG (presentato nella sez. 1.2.3.1). In questo capitolo ci si soffermerà sulla particolarizzazione dell'algoritmo nel caso di applicazione specifico.

L'obiettivo dell'agente, in questo caso, è quello di far rimanere sempre in carreggiata l'auto, mentre cerca di farle completare il percorso più velocemente possibile, in uno scenario ad agente singolo.
Il modello dell'intero sistema di fig.3.1 mostra le interazioni tra agente ed environment TORCS.

Il flusso del sistema può essere diviso in due parti fondamentali che avvengono simultaneamente ad ogni istante di tempo t:

1. *Esplorazione:* A partire da una osservazione s_t la rete Actor μ_ϕ produce un'azione $a_t = \mu_\phi(s_t)$, con $a_t = (a_{accel}, a_{brake}, a_{steer}) \in \mathbb{R}^3$. A questa azione viene aggiunto un *rumore esplorativo* e il risultato viene poi inviato a TORCS che restituisce lo stato successivo s_{t+1}. Dallo stato successivo si calcola la ricompensa immediata r_{t+1}. La transizione del sistema $(s_t, a_t, s_{t+1}, r_{t+1})$ viene infine memorizzata nell'experience replay per l'allenamento dell'agente.

2. *Allenamento:* Viene prelevato un batch di transizioni dal buffer. I parametri della rete Critic Q_θ vengono aggiornati seguendo la loss di eq.1.14. L'Actor, successivamente, viene aggiornato secondo il gradiente di eq.1.15. Infine i parametri delle reti target vengono aggiornati tramite soft update seguendo la eq.1.16.

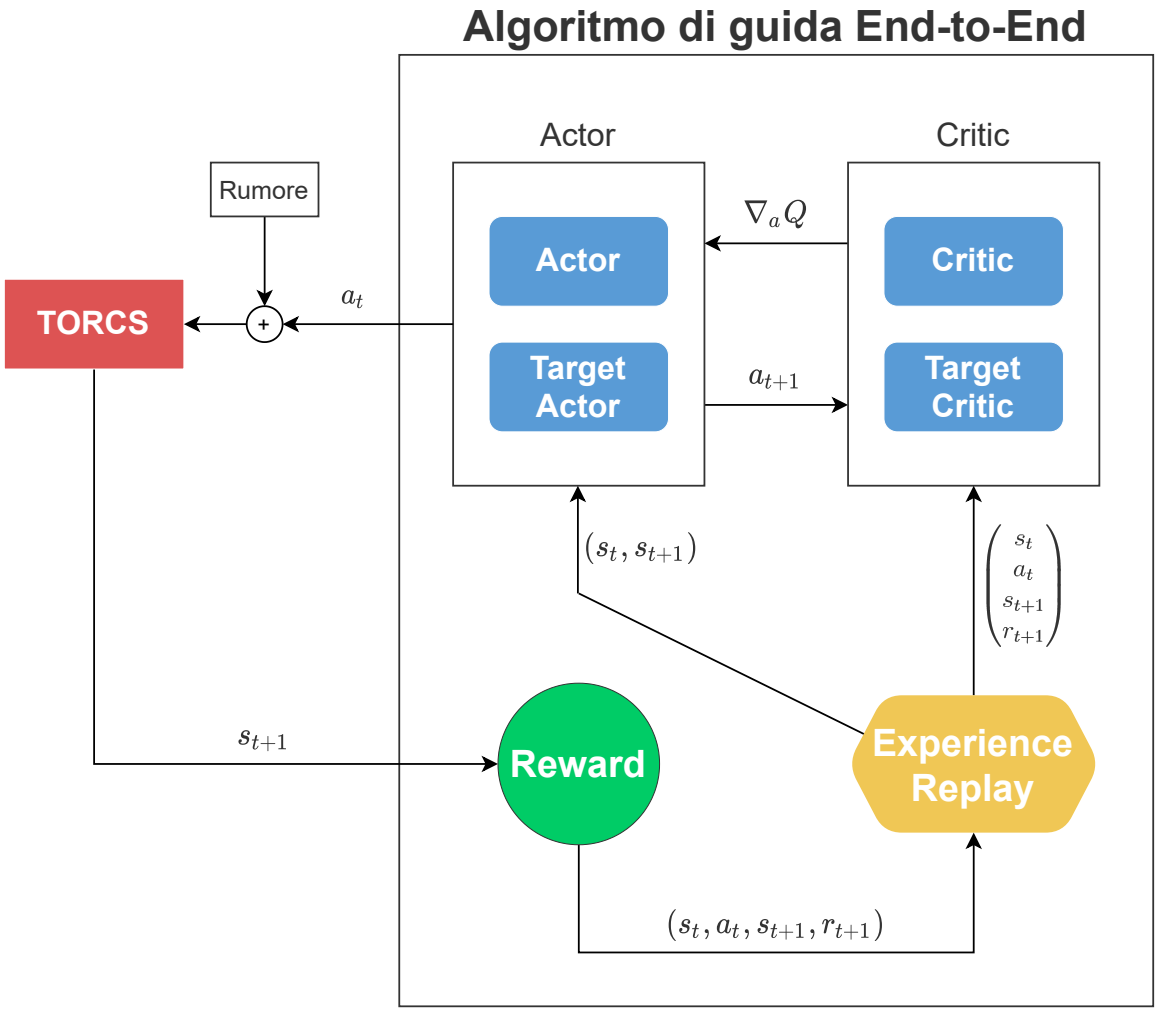

Figura 3.1: Modello del sistema TORCS-Agente

3.1 Architettura delle reti

In questa sezione vengono definite le architetture delle reti utilizzate.

La rete Actor di fig.3.2 è una rete *fully connected* con un input di 24 neuroni, che rappresentano lo stato, e un output di 3 neuroni, ovvero le azioni. I due neuroni che producono l'uscita di *accel* e *brake* sono attivati con *sigmoide* per garantire un output compreso in $[0, 1]$, mentre il neurone che produce l'azione di *steer* viene attivato dalla *tangente iperbolica* in modo da avere un range di uscita compreso in $[-1, +1]$. Frapposti tra i livelli di ingresso e uscita ci sono due livelli nascosti da 300 e 400 neuroni con attivazione relu.

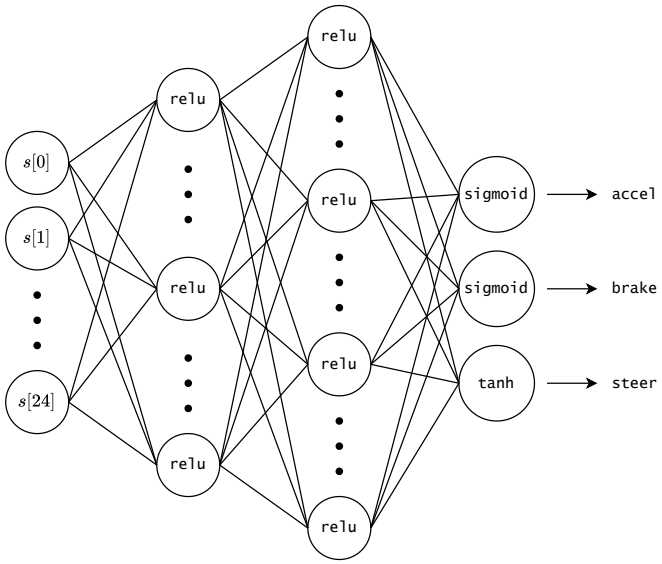

Figura 3.2: Rete Actor

La struttura della rete Critic è mostrata in fig.3.3. L'input è diviso in due parti, una è l'informazione sullo stato dell'ambiente fornita da TORCS e l'altra è il valore dell'azione emesso dalla rete Actor. Le informazioni sullo stato ambientale vengono elaborate dal primo livello nascosto, combinate con il valore dell'azione e inserite nel secondo livello nascosto. I due livelli includono rispettivamente 300 e 400 neuroni e sono attivati da funzione relu. Infine il risultato viene elaborato dal neurone di output per ottenere il Q-value.

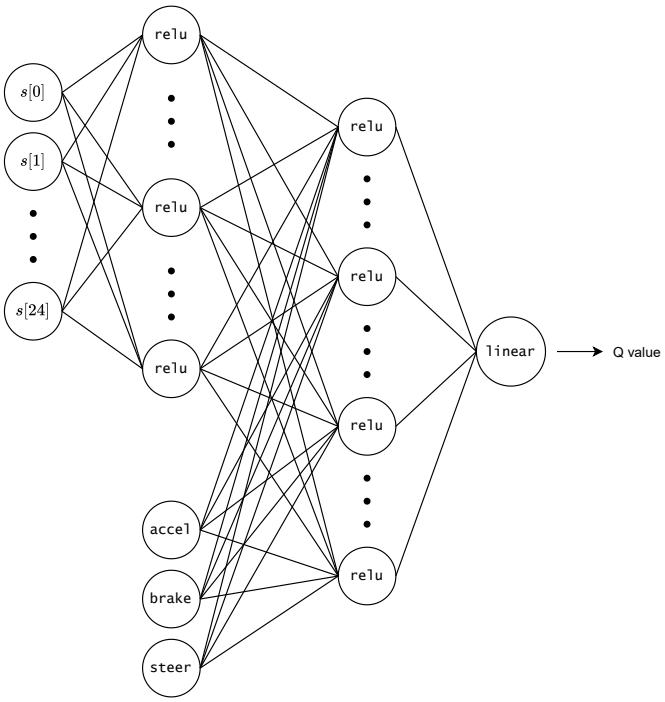

Figura 3.3: Rete Critic

3.2 Reward Shaping

Come viene spiegato nel capitolo introduttivo del RL, la ricompensa è ciò che più di ogni altra cosa caratterizza l'apprendimento dell'agente. Essa infatti definisce *cosa l'agente imparerà e come*.

L'obiettivo in questo caso è mantenere il veicolo al centro della corsia e terminare il percorso più velocemente possibile. La ricompensa sarà quindi funzione della velocità e dell'offset tra l'agente e il centro della corsia. Nella eq.3.1 si definisce la funzione di ricompensa, dove le variabili sono definite nella tab. 2.1.

$$r = \begin{cases} S_x\, cos(\alpha) - |S_x\, sin(\alpha)| - |S_x\, \gamma| & \text{se } |\gamma| < 1 \\ -200 & \text{se } |\gamma| > 1 \end{cases} \tag{3.1}$$

con $S_x = speedX$, $\alpha = angle$ e $\gamma = trackPos$.

Oltre alla reward sopra definita sono state testate anche delle sue varianti. Una prima variante senza il controllo puntuale della trackPos, definita nella eq.3.2, e una sostituendo il controllo della trackPos con un controllo dell'angolo dell'auto rispetto alla carreggiata, definita nella eq.3.3.

$$r = \begin{cases} S_x\, cos(\alpha) - |S_x\, sin(\alpha)| & \text{se } |\gamma| < 1 \\ -200 & \text{se } |\gamma| > 1 \end{cases} \tag{3.2}$$

$$r = \begin{cases} S_x\, cos(\alpha) - |S_x\, sin(\alpha)| - S_x \left|\frac{\alpha}{\pi}\right| & \text{se } |\gamma| < 1 \\ -200 & \text{se } |\gamma| > 1 \end{cases} \tag{3.3}$$

3.3 Rumore Esplorativo

Negli algoritmi di Deep RL devono essere impostate strategie di esplorazione appropriate per evitare che l'algoritmo cada in un ottimo locale. Questo problema è comunemente noto come l'*Explore-Exploit Dilemma*.

L'Explore consiste nella scelta di un'azione casuale. Ciò consente all'agente di migliorare le proprie conoscenze attuali sugli action-value, con un possibile vantaggio a lungo termine. Il miglioramento dell'accuratezza degli action-value stimati gli consentirà di prendere decisioni più informate in futuro. *L'Exploit*, d'altra parte, sceglie l'azione avidamente per ottenere la massima ricompensa sfruttando le attuali stime delle action-value conosciute. Quando un agente esplora tende a costruire stime mano a mano più accurate degli action-value e quando sfrutta mira ad ottenere più ricompense. Non può, tuttavia, scegliere di fare entrambe le cose contemporaneamente. Per tale ragione è necessario utilizzare tecniche per la gestione delle fasi di esplorazione e sfruttamento.

In base all'algoritmo utilizzato esistono tecniche esplorative differenti. Nel caso del DQN, una nota strategia esplorativa è l'ϵ-greedy, che consiste nell'eseguire un'azione casuale con probabilità ϵ e un'azione che massimizzi la Q con probabilità $(1 - \epsilon)$.

Strategie di esplorazione comuni come ϵ-greedy non sono adatte a scenari di guida autonoma come quello esaminato in questa tesi perché si presenterebbero molti casi di azioni invalide, come ad esempio azioni in cui la frenata è maggiore dell'accelerazione.

Per tale ragione l'autore del DDPG propone l'utilizzo di un rumore additivo sull'azione generato dal processo stocastico di Ornstein-Uhlenbeck[3]. Nell'ambito di questo lavoro sono stati testati tre tipi di rumori esplorativi: una variante dell'*Ornstein-Uhlenbeck*, un *Gaussian Noise* e un *Time Variant Noise*. Questi rumori vengono presentati nelle seguenti sottosezioni.

3.3.1 Ornstein-Uhlenbeck Mod

Questo rumore è una variazione dell'*Ornstein-Uhlenbeck* in quanto nel 10% dei casi viene prodotto un rumore gaussiano $N(\mu, \sigma)$ mentre nel 90% dei casi viene prodotto un rumore in base al classico processo stocastico di Ornstein-Uhlenbeck.

Questo serve a fare in modo che l'agente sia sollecitato con due stimoli molto diversi fra loro in misure diverse. Nove volte su dieci si stimola l'agente ad andare molto veloce attraverso un rumore additivo con media alta sull'accelerazione e nulla sulla frenata e, una volta su dieci, invece, lo si stimola a frenare attraverso un rumore con una media nettamente inferiore sull'accelerazione e alta sulla frenata.

3.3.2 Gaussian Noise

Questo rumore è una variazione dell'*ϵ-greedy Gaussian Noise* in quanto nel 90% dei casi viene prodotto un rumore gaussiano $N(\mu_1, \sigma_1)$ mentre nel 10% dei casi uno $N(\mu_2, \sigma_2)$.

Anche in questo caso si utilizza questo stratagemma, con un opportuno tuning dei parametri di medie e std, per sollecitare l'agente con stimoli molto diversi fra loro per garantire che esso impari un più ampio spettro di comportamenti.

La probabilità ϵ viene fatta decadere linearmente rispetto al numero delle iterazioni di training.

3.3.3 Time Variant Noise

Il TVNoise è un rumore gaussiano con media e std variabile nel tempo. La classe restituisce un rumore gaussiano $N(\mu_1, \sigma_1)$ per i primi **`time_step1`** passi.

Successivamente tende linearmente ad un rumore con distribuzione gaussiana $N(\mu_2, \sigma_2)$ in un numero di passi pari a
`(self.time_step2 - self.time_step1)`.

Infine, sempre linearmente, tende a $N(\mu_3, \sigma_3)$ in **`(self.time_step3 - self.time_step2)`** passi. Nel progetto di tesi questo terzo passaggio viene sfruttato per far decadere a zero linearmente il rumore gaussiano aggiunto.

Nella fig.3.4 viene mostrato un esempio di possibile andamento della media del rumore gaussiano. Lo stesso ragionamento si applica alla std.

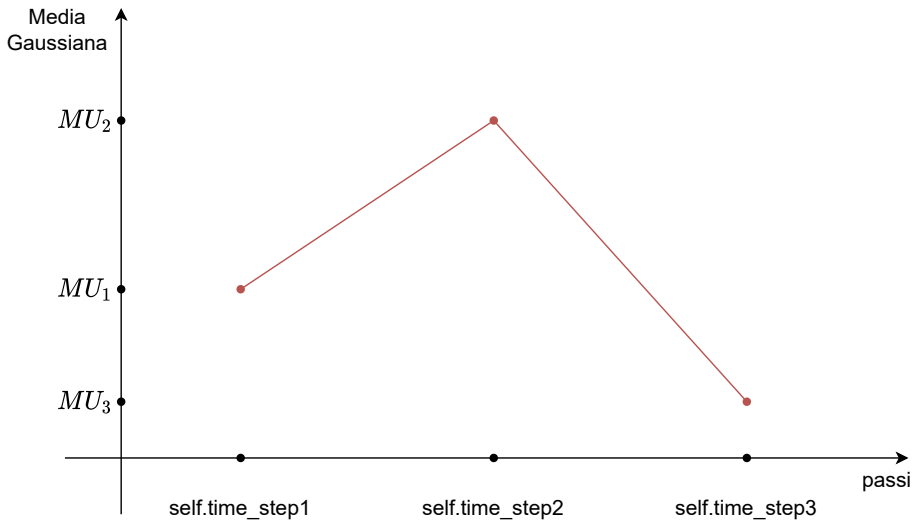

Figura 3.4: Andamento della media Gaussiana

Capitolo 4

Simulazioni e Risultati

4.1 Set-up Numerico

In questo capitolo si discutono le simulazioni effettuate e si analizzano i risultati con l'obiettivo di trovare il miglior modello, sia rispetto agli episodi di allenamento, sia rispetto agli altri modelli che presentano parametri diversi. Per affrontare in maniera consistente questo processo, si definiscono nella tab.4.1 i parametri che caratterizzano nell'interezza un modello DDPG e nella tab.4.2 le metriche utilizzate nel giudizio di questi ultimi.

Nella fig.4.1 si vede a sinistra il tracciato utilizzato per il training dei modelli e a destra quello utilizzato per la validazione.

Figura 4.1: Tracciati di training e validazione

Nome	Descrizione
MEM_SIZE	Dimensione dell'experience replay
BATCH_SIZE	Dimensione del batch di transizioni prelevate contemporaneamente dall'experience replay
DC_FACT	Discount Factor γ del Q Learning
SM_FACT	Fattore di smooth τ utilizzato nel soft update
CLR	Learning rate del Critic e del Target Critic
ALR	Learning rate dell'Actor e del Target Actor
C_1	Numero di neuroni del primo livello denso nascosto del Critic e del Target Critic
C_2	Numero di neuroni del secondo livello denso nascosto del Critic e del Target Critic
A_1	Numero di neuroni del primo livello denso nascosto dell'Actor e del Target Actor
A_2	Numero di neuroni del secondo livello denso nascosto dell'Actor e del Target Actor
NOISE	Rumore esplorativo scelto per il modello
REWARD	Reward scelta per il modello

Tabella 4.1: Parametri del modello

Nome	Descrizione
EPISODIC_REWARD	Reward cumulativa di un episodio
MSE_TRACKPOS	Mean Squared Error della trackPos rispetto al centro della carreggiata (ovvero trackPos = 0) di un episodio
N_STEPS	Numero di step che compongono l'episodio (max 6000)

Tabella 4.2: Metriche di giudizio

4.1.1 Panoramica dei modelli allenati

Nella tab.4.4 sono presentati tutti i modelli che sono stati allenati insieme ai valori dei loro parametri.

Non tutti i parametri hanno subito variazioni durante i test, pertanto nella tab.4.3 sono presentati tali parametri e i loro valori predefiniti.

Parametro	**Valore**
MEM_SIZE	100000
BATCH_SIZE	64
DC_FACT	0,99
SM_FACT	0,001
CLR	0,001
ALR	0,0001
C_1	300
C_2	400
A_1	300
A_2	400

Tabella 4.3: Valore predefinito dei parametri immutati

MODEL_ID	**NOISE**	**REWARD**
1	OU	standard
2	TVNoise	standard
3	GN	standard
4	OU	no trackPos
5	OU	angle

Tabella 4.4: Modelli e rispettivi parametri

4.2 Scelta del modello DDPG

Lo scopo di questa sezione è evidenziare il miglior modello tra quelli presentati, rispetto ai parametri che hanno subito variazioni.

4.2.1 Modalità del confronto tra modelli

Ogni confronto ha lo scopo di evidenziare il miglior episodio di allenamento di un modello, rispetto alla variazione di un parametro, sia tra tutti i modelli sotto test che tra gli episodi di allenamento del modello stesso.

Per la valutazione si procede come segue:

1. Si prelevano i 100 episodi di allenamento di ogni modello che hanno il maggior N_STEPS.
2. Di questi, si selezionano solamente i 30 episodi che hanno il minor MSE_TRACKPOS.
3. Di questi, si seleziona l'episodio di ciascun modello che ha la EPISODIC_REWARD più grande.
4. Infine si seleziona l'episodio del modello che, tra tutti quelli sotto test, ha la maggior EPISODIC_REWARD.

4.2.2 Test sul tipo di rumore

In questo test si mettono a paragone i tre tipi di rumore utilizzati: l'OU Modificato, il GN e il TVNoise. Si testeranno, quindi, il Modello 1, 2 e 3.

In fig.4.2, 4.3, 4.4 vengono mostrati gli andamenti delle metriche di interesse rispetto agli episodi di testing dei tre modelli. Da sinistra verso destra, rispettivamente, EPISODIC_REWARD, N_STEPS, MSE_TRACKPOS.

Alla fine dei primi 3 passi della valutazione rimane da confrontare l'episodio 415 del Modello 1 con quello 493 del Modello 2 e quello 451 del Modello 3. Come si vede in tab.4.5, il Modello 1 presenta una EPISODIC_REWARD maggiore dei Modelli 2 e 3.
Gli MSE_TRACKPOS dei Modelli 1 e 2 sono comparabili mentre il Modello 3 ha performance di tracking notevolmente inferiori ai primi due.

In questo caso, nel rispetto delle modalità di confronto definite, il Modello 1 risulta il migliore.

MODEL_ID	EPISODIO	EPISODIC_REWARD	MSE_TRACKPOS	N_STEPS
1	415	510353	0,022	6000
2	493	483797	0,015	6000
3	451	400165	0,088	6000

Tabella 4.5: Confronto Modelli 1, 2 e 3

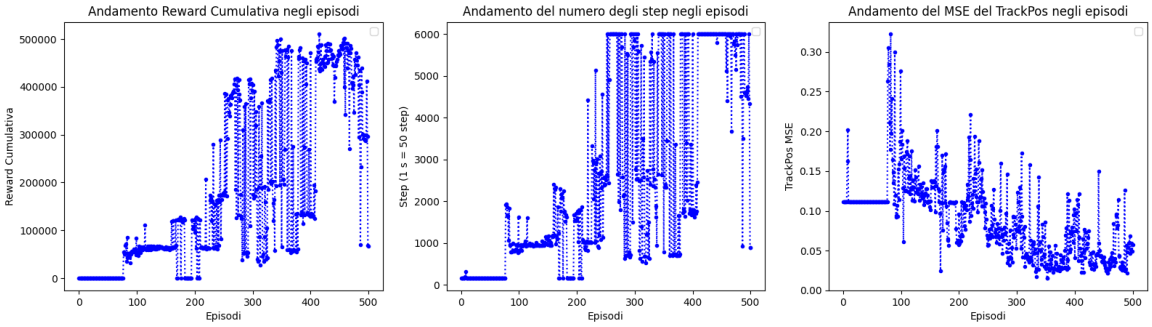

Figura 4.2: Statistiche del Modello 1

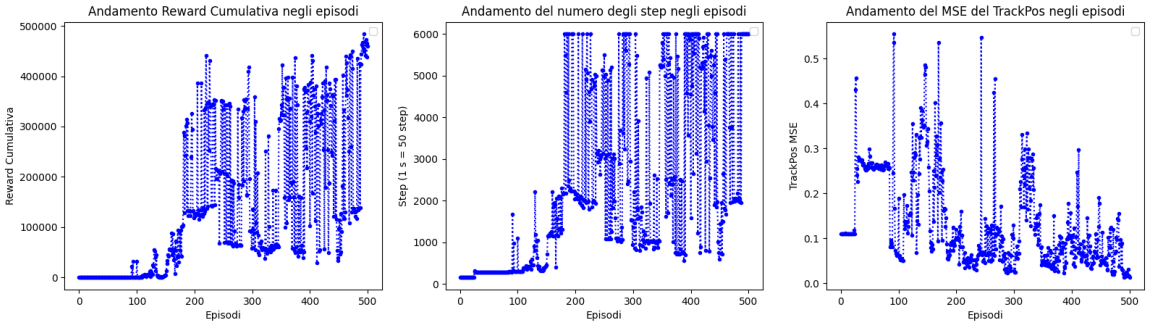

Figura 4.3: Statistiche del Modello 2

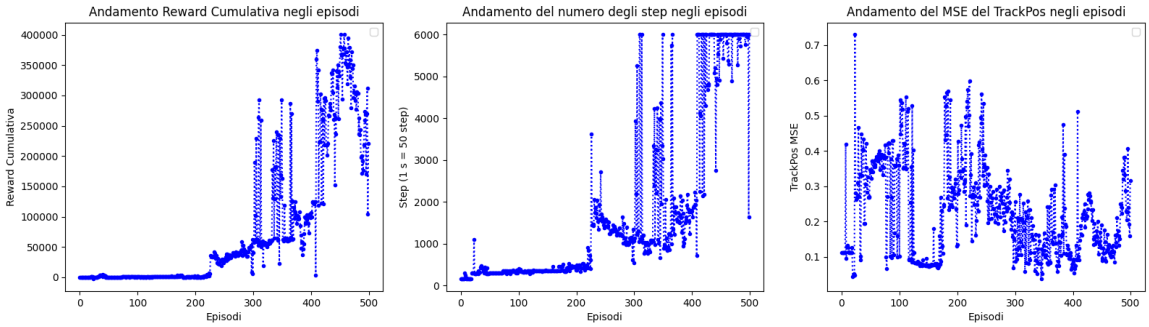

Figura 4.4: Statistiche del Modello 3

4.2.3 Test sul tipo di reward function

Per verificare quale reward function porti risultati migliori, si mettono a paragone i Modelli 1, 4 e 5. Nella fig.4.5 si mostrano gli andamenti delle metriche del Modello 4, mentre nella fig.4.6 quelli del Modello 5.

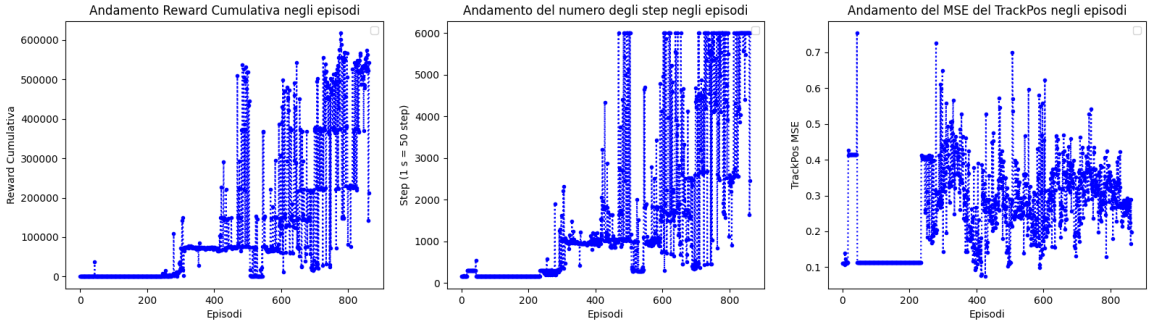

Figura 4.5: Statistiche del Modello 4

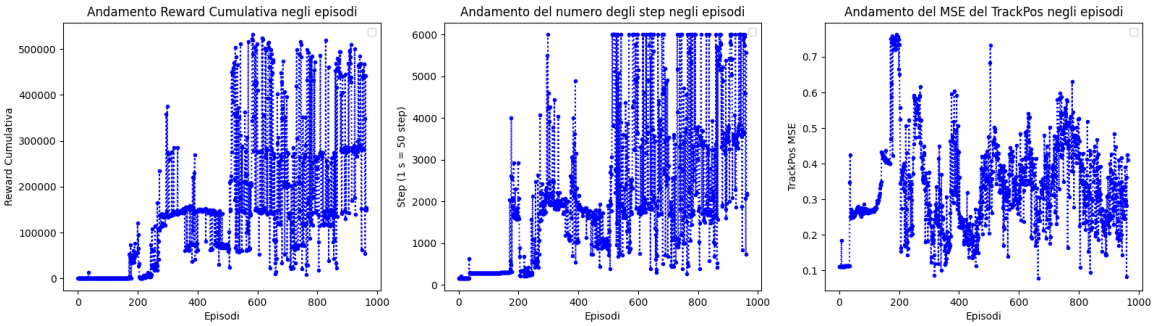

Figura 4.6: Statistiche del Modello 5

Alla fine dei primi 3 passi della valutazione rimangono da confrontare l'episodio 415 del Modello 1, l'episodio 778 del Modello 4 e l'episodio 583 del Modello 5.

Come si nota dalla tab.4.6, i Modelli 4 e 5 hanno una EPISODIC_REWARD maggiore di quella del Modello 1. Tuttavia si riconosce che il Modello 1 riesce a mantenere un MSE_TRACKPOS dieci volte più basso rispetto a quello degli altri due modelli.

Riassumendo, quindi, l'utilizzo di una reward che include il controllo puntuale della trackPos, come quella definita in 3.1, favorisce l'allenamento di un modello che segue pedissequamente il centro della carreggiata, mentre l'utilizzo di una reward come quelle definite in 3.2 e 3.3 porta all'allenamento di un agente che, seppur mantenendosi sempre entro i limiti della carreggiata, riesce ad elaborare strategie di guida più audaci che portano a ricompense più alte perché è in grado di sfruttatare meglio la conformazione della pista.

In ultima analisi si osserva che il Modello 5, il quale implementa una reward con controllo sull'angolo del veicolo, ha un MSE_TRACKPOS maggiore di quello del Modello 4. Questo ci fa concludere che la reward più funzionale alla risoluzione del problema proposto è quella del Modello 4.

MODEL_ID	EPISODIO	EPISODIC_REWARD	MSE_TRACKPOS	N_STEPS
1	415	510353	0,022	6000
4	778	601018	0,276	6000
5	583	531486	0,284	6000

Tabella 4.6: Confronto Modelli 1, 4 e 5

4.2.4 Modello finale

Il Modello 4 e il Modello 5 risultano essere, nel complesso, i due miglior modelli secondo le modalità di confronto definite. Il Modello 4, però, riesce ad ottenere performance molto più alte in termini di EPISODIC_REWARD, quindi è scelto come modello definitivo nell'episodio 778.

4.3 Risultati numerici

In questa sezione vengono presentati i risultati numerici ottenuti sul tracciato di validazione dal Modello 4 nell'episodio 778 e confrontati con quelli del Modello 1 nell'episodio 415.

Nella fig.4.7, 4.8 e 4.9 vengono mostrati, rispettivamente, l'andamento della reward, l'andamento della velocità e l'andamento dell'errore quadratico della trackPos dei due modelli nei due rispettivi episodi migliori. Nell'analisi della 4.9 si tenga presente che il Modello 4 implementa una reward che non impone il mantenimento del centro della carreggiata bensì solo di rimanere all'interno di quest'ultima. Per tale ragione si considera buono ogni risultato minore di 1.

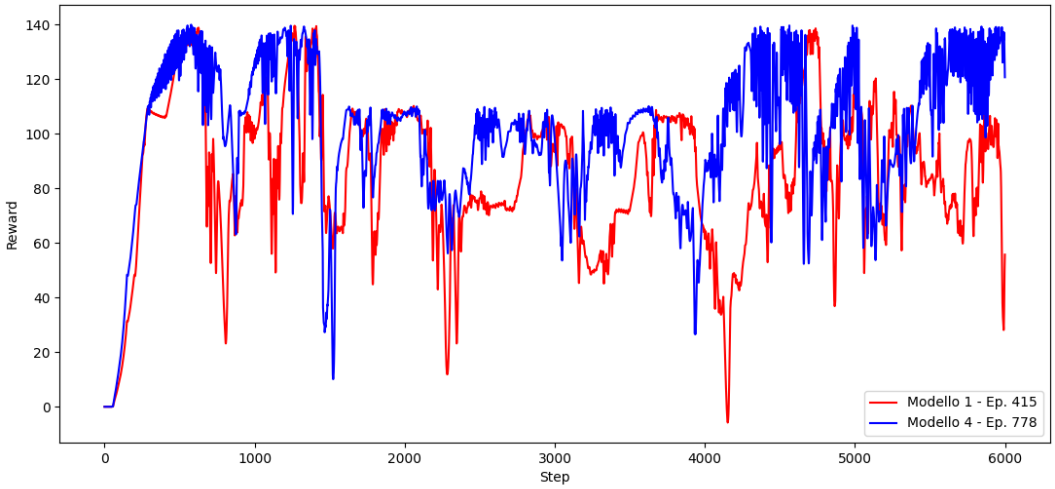

Figura 4.7: Reward nell'episodio

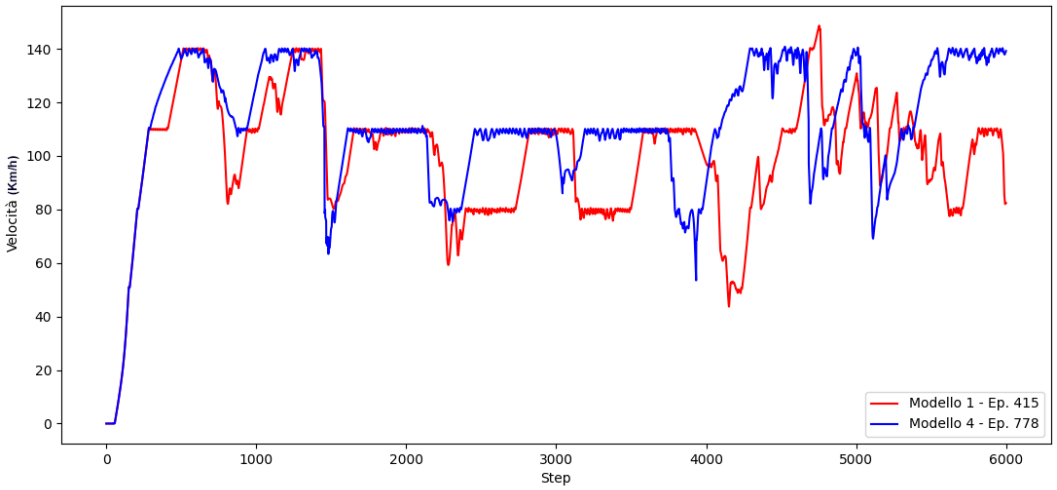

Figura 4.8: Velocità nell'episodio

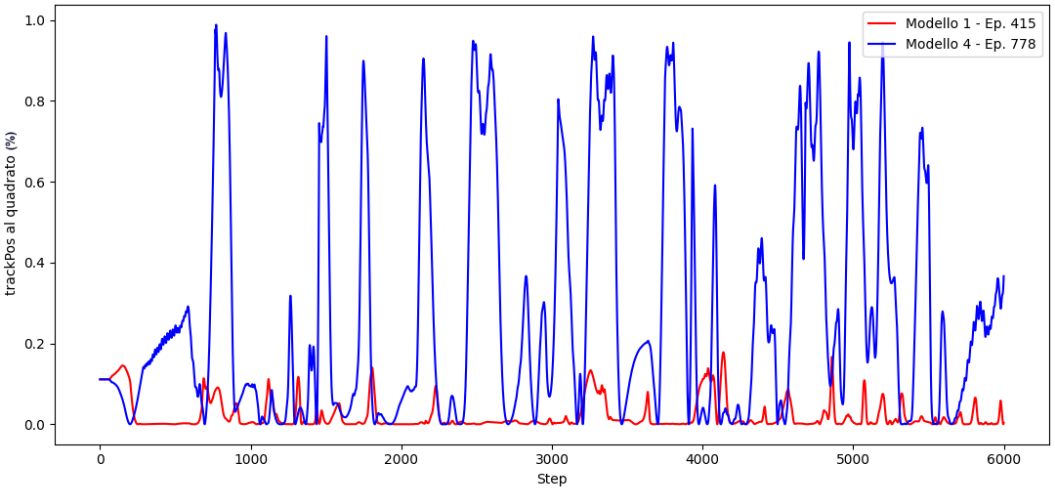

Figura 4.9: Errore quadratico trackPos nell'episodio

Conclusioni

In questa tesi ci si è approcciati al problema del lane keeping nell'ambiente di simulazione di guida TORCS utilizzando un approccio di Reinforcement Learning: il DDPG.

Si è partiti facendo un'introduzione al Reinforcement Learning per poi approdare alle definizioni dei più comuni algoritmi di Deep RL, in cui ricade anche il DDPG. Si è poi presentato nel dettaglio l'ambiente di simulazione TORCS, evidenziandone lati positivi e criticità, le quali hanno avuto bisogno di ulteriori attenzioni sul piano implementativo. Si è spiegata l'importanza del problema del lane keeping nel campo dell'AU e si è utilizzato il DDPG proprio per la creazione di un sistema decisionale end-to-end per la gestione di tale processo nel caso di un veicolo terrestre.

Infine si sono allenati molteplici modelli e si sono effettuate svariate simulazioni su diverse combinazioni di variazioni parametriche, come il rumore esplorativo e la reward function. Dato che questi test sono stati svolti in un simulatore di gare automobilistiche, ci si è posti il problema di identificare un modello che avesse ottime performance nel mantenimento di carreggiata, senza però mettere in secondo piano la metrica della velocità dell'auto. Per tale ragione il migliore è risultato essere il Modello 4, con reward senza controllo puntuale sulla trackPos e rumore esplorativo Ornstein-Uhlenbeck Mod.

L'OU si è dimostrato avere performance migliori rispetto alle altre due classi di rumore esplorativo mentre la reward function, caratterizzata dall'avere un maggiore grado di libertà sulla posizione dell'auto in carreggiata, ha permesso di ottenere ottime performance nel caso di applicazione studiato.

In defintiva il DDPG si è dimostrato essere un validissimo algoritmo per la creazione di un sistema decisionale end-to-end per il lane keeping di un veicolo terrestre.

Bibliografia

[1] Zhiqing Huang, Ji Zhang, Rui Tian, and Yanxin Zhang. End-to-end autonomous driving decision based on deep reinforcement learning. In *2019 5th International Conference on Control, Automation and Robotics (ICCAR)*, pages 658–662, 2019.

[2] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[3] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[4] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *arXiv preprint arXiv:1304.1672*, 2013.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

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

[7] TORCS. `http://torcs.sourceforge.net/`.

[8] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[10] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[11] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[12] Chris Yoon. `https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63`.