Progetto Data-Driven
Progettazione dei Sistemi di Controllo

Claudio Ciavola
Mat. 0622700726
Corso di Laurea in Ingegneria Informatica Magistrale
Università degli Studi di Salerno

15 giugno 2020

Sommario

Questo elaborato ha lo scopo di descrivere l'esperienza di progetto avutasi durante l'insegnamento di Progettazione dei Sistemi di Controllo del corso di laurea magistrale in Ingegneria Informatica, presso l'Università degli Studi di Salerno.
L'attività di progetto prevede la descrizione di alcuni algoritmi, esistenti in letteratura, di reinforcement learning e la loro applicazione sull'enviroment openAI chiamato BipedalWalker. Sono stati implementati alcuni degli ultimi approcci di RL per la locomozione robotica, inizialmente sono stati considerati algoritmi semplici per poi passare ad algoritmi più complessi ed evoluti.

1 INTRODUZIONE

L'ambiente BipedalWalker-v3 è fornito dall'organizzazione OpenAI, e può essere importato in Python come frame-work attraverso il modulo `openAi.gym`.
Questo framework fornisce delle API, che permettono il test degli algoritmi AI. L'ambiente specifico invece, illustra un robot bipede in 2D, composto da uno scafo con due gambe e ognuna delle quali è costituita da due articolazione (anca e ginocchio). Il robot è l'agente, deve camminare senza cadere su un percorso piano, quindi l'obiettivo principale è quello di costruire un algoritmo che allena il robot a camminare.
Questo è fondamentalmente un processo decisionale di Markov (MDP) in cui l'azione appropriata deve essere scelta ad ogni passo sotto la politica corrente. Con il termine politica, ci riferiamo alle azioni che il robot sceglie da una serie di possibili azioni A per ogni stato $s \in S$. Formalmente la politica è la probabilità che l'agente scelga un azione a quando si trova in uno stato s.
Il robot deve essere addestrato attraverso una procedura "trial and error" in accordo all'algoritmo scelto.

2 OBIETTIVO

L'obiettivo principale di questo progetto è quello di risolvere il problema caratterizzato dall'environment BipedalWalker-v3. Il problema si pone come concluso quando il robot sceglie azioni corrette per non cadere e camminare fino alla fine del percorso. In particolare, la ricompensa restituita dall'ambiente è un valore dello spazio continuo che dipende dalla distanza che il robot percorre. Formalmente la risoluzione del problema si ha quando il robot ottiene una ricompensa media maggiore di 300 (percorso completato) per 100 episodi consecutivi:

$$risoluzione \implies \frac{1}{100}\sum_{i=1}^{100} r_i \geq 300$$

Si vuole trovare quindi un algoritmo di reinforcement learning in grado di raggiungere l'obiettivo suddetto in un tempo minore possibile.

3 AMBIENTE

L'ambiente genera osservazioni dello stato come conseguenza all'azione esercitata dal robot. Più

precisamente, per ogni step $t \in T$ nello stato $s_t \in S$, il robot seleziona l'azione $a_t \in A$. Successivamente l'ambiente ritorna una ricompensa $R(s_t) \in \mathbb{R}$ e un nuovo stato $s_{t+1} \in S$. Il valore di uno stato, invece, è la quantità totale di ricompensa che l'agente può aspettarsi di accumulare in futuro a partire da quello stato. Tutti gli stati e le ricompense vengono generati dall'ambiente dopo ogni azione.

Num	Observation	Min	Max
0	Angolo scafo	0	2π
1	Velocità angolare scafo	$-\infty$	$+\infty$
2	Velocità scafo lungo x	-1	+1
3	Velocità scafo lungo x	-1	+1
4	Angolo anca gamba 1	$-\infty$	$+\infty$
5	Velocità angolare anca gamba 1	$-\infty$	$+\infty$
6	Angolo ginocchio gamba 1	$-\infty$	$+\infty$
7	Velocità angolare ginocchio gamba 1	$-\infty$	$+\infty$
8	Flag contatto gamba 1 - terreno	0	1
9	Angolo anca gamba 2	$-\infty$	$+\infty$
10	Velocità angolare anca gamba 2	$-\infty$	$+\infty$
11	Angolo ginocchio gamba 2	$-\infty$	$+\infty$
12	Velocità angolare ginocchio gamba 2	$-\infty$	$+\infty$
13	Flag contatto gamba 2 - terreno	0	1
14-23	10 parametri lidar	$-\infty$	$+\infty$

Tabella 1: Composizione stato

Le ricompense definiscono quanto è stata buona l'azione a_t allo step t, dallo stato corrente $s_t \rightarrow s_{t+1}$. Intuitivamente, lo scopo è quello di costringere il robot a selezionare azioni, a seconda dello stato, che massimizzano le ricompense (reward).
In un MDP la politica dipende solo dallo stato corrente e non dalla storia passata. Tutti i MDP finiti hanno almeno una politica ottimale, che può dare la massima ricompensa, e tra tutte le politiche ottimali almeno una è stazionaria e non varia nel tempo.

Num	Action	Min	Max
0	Coppia anca gamba 1	-1	+1
1	Coppia ginocchio gamba 1	-1	+1
2	Coppia anca gamba 2	-1	+1
3	Coppia ginocchio gamba 2	-1	+1

Tabella 2: Composizione azione

Un'osservazione dello stato $s_t \in S$, è rappresentata da un vettore di 24 elementi nello spazio continuo, nella tabella 1 è riportata la composizione dello stato e il dominio di ogni parametro.

Le azioni $a_t \in A$, sono rappresentate da un vettore di 4 elementi nello spazio continuo, nella tabella 2 è riportata la composizione delle azioni e il loro dominio.
L'episodio termina quando il percorso viene completato e si ha la ricompensa di 300, oppure quando lo scafo del robot tocca il suolo con una ricompensa di -100.
Formalmente il problema si considera concluso quando si ha per 100 prove consecutive una ricompensa media maggiore o uguale di 300.

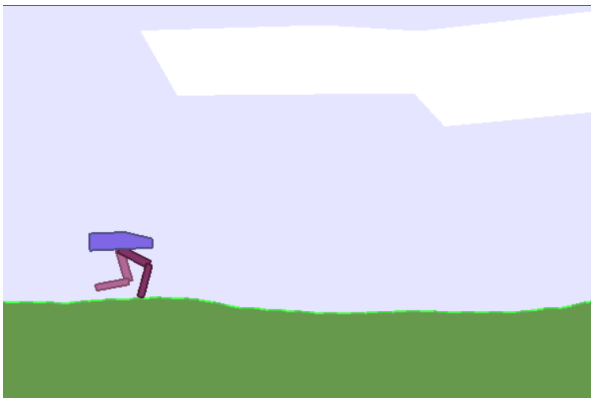


Figura 1: BipedalWalker screenshot

La figura 2 e l'algoritmo 1 descrivono un semplice esempio di ciclo agente-ambiente. Si fornisce un'azione in input e si ottiene immediatamente la nuova osservazione dello stato e della ricompensa dall'ambiente Tuttavia, prima di ciò è fondamentale fornire una formulazione matematica dei vettori, azione e stato, precedentemente citati.

$$state = \langle h_0, \frac{\partial h_0}{\partial t}, \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, h_1, \frac{\partial h_1}{\partial t}, k_1, \frac{\partial k_1}{\partial t}, f_{leg_1}, h_2, \frac{\partial h_2}{\partial t}, k_2, \frac{\partial k_2}{\partial t}, f_{leg_2}, lidar \rangle \tag{1}$$

$$action = \langle H_1, K_1, H_2, K_2, \rangle \tag{2}$$

Le informazioni sopra riportate sono fornite dal repository github di openAI-wiki, dell'ambiente BipedalWalker-v2, ma queste informazioni sono identiche per la v3.

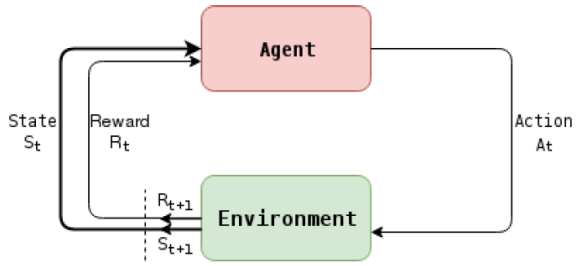


Figura 2: Ciclo Agente-Ambiente

Algorithm 1 Esempio interazione ambiente

```
import environment OpenAi.gym
env = OpenAiLoad('BipedalWalker-v3')
for i in range(n.episodes) do
    state = resetEnv()
    while True do
        renderEnv()
        action = randomAction()
        observation, reward, done, info = stepEnv(action)
        if done then
            print("Episode finished")
            break
        end if
    end while
end for
closeEnv()
```

4 PROCESSO DI PROGETTAZIONE E CENNI TEORICI

Questo progetto nasce nell'ambito del reinforcement learning, e prevede diverse fasi. La fase iniziale aveva lo scopo di prendere confidenza con gli aspetti teorici e i meccanismi fondamentali del reinforcement learning e formalizzare una prima semplice soluzione utilizzando concetti alla base di ogni algoritmo di reinforcement learning.
Dopodiché si è passato a considerare algoritmi più evoluti, innanzitutto individuando le differenza e le similitudini con gli algoritmi già testati e poi applicarli sull'ambiente in esame. Successivamente c'è stata una fase in cui scegliere opportunamente tra le varie configurazioni e parametri possibili per poter migliorare le performance dell'algoritmo.
Infine, l'ultima fase prevede un confronto tra tutti i vari approcci utilizzati, analizzando pregi e difetti e infine scegliendo quello che raggiunge lo scopo e che risolve il problema.

4.1 Equazioni di Bellman

L'equazione 3 è detta equazione di Bellman per funzione azione-valore, e mette in evidenza che il valore di uno stato precedente è determinato da due parti: la ricompensa immediata e il valore degli stati successivi pesato per il discount rate γ.

$$q_\pi = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad (3)$$

Il parametro $\gamma \in [0, 1]$ definisce il tasso di decadimento, è costante e generalmente è posto pari a $0,99$. Il suo ruolo è quello di manipolare il modo in cui l'agente considera le ricompense future. Se γ è prossimo a 0 l'agente viene detto miope e massimizza maggiormente i premi immediati, invece se è prossimo a 1 l'agente viene detto lungimirante poiché massimizza maggiormente le ricompense future.
L'algoritmo Q-learning apprende una politica, che fa scegliere ad un agente le azioni da intraprendere in determinati stati.
Per ogni MDP finito, Q-Learning trova una politica che massimizza il valore atteso della ricompensa totale su tutti i passaggi successivi, a partire dallo stato attuale. Quindi quando il robot si trova in uno stato s_t, sceglie un'azione a_t, si muove nel nuovo stato $s_{t+1} \in S$. Al nuovo stato s_{t+1}, il robot deve selezionare l'azione successiva che produce la più alta ricompensa-valore e aggiornare in modo iterativo la seguente equazione:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)] \quad (4)$$

L'equazione 4 è l'equazione di Bellman utilizzata in Q-Learning, dove $R \in \mathbb{R}$ definisce la ricompensa ottenuta dallo stato attuale scegliendo l'azione a.
Il parametro α è il tasso di apprendimento e solitamente ha un valore di circa 10^{-4}. Esso controlla la velocità con cui l'agente dimentica gli stati precedenti e apprende dalle nuove esperienze.
Q-Learning è un metodo off-policy cioè utilizza una policy totalmente greedy che viene utilizzata per l'esplorazione (behavior policy) delle coppie stato-azione in modo tale da non ricadere in ottimi locali,

e una policy di apprendimento vera e propria con politica ϵgreedy chiamata anche target policy. Attraverso un fattore di esplorazione ϵ, si spinge l'algoritmo verso una scelta esplorativa piuttosto che di apprendimento. con l'avanzare degli episodi ϵ viene decrementato per dare più peso alla policy di apprendimento. Nella pratica quindi si cerca di mantenere una politica maggiormente esplorativa all'inizio del training per poi spostare l'attenzione sul fattore di apprendimento

Algorithm 2 Q-Learning (off-policy TD)

inizializza i parametri:$\alpha \in (0, 1]$, $\epsilon > 0$
inizializza $Q(s, a) \forall s \in S^+, a \in A(s)$
for all episode **do**
    $S \leftarrow$ stato iniziale
    for all steps in episode **do**
        scegli A da S usando la politica derivata da Q (ϵ-greedy)
        seleziona A, osserva R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    end for
end for

L'obiettivo è trovare una politica ottimale di azione previsionali, che massimizzano la ricompensa presente e futura utilizzando l'algoritmo 2 di aggiornamento della funzione Q.

5 ALGORITMO Q-LEARNING

Q-Learning è un algoritmo di reinforcement learning basato sul valore che viene utilizzato per trovare l'azione ottimale attraverso una politica che utilizza una funzione Q.
Nelle sezioni seguenti viene illustrato l'algoritmo, poiché esso è un punto di partenza basilare per il reinforcement learning.
Questo algoritmo è un algoritmo model-free ovvero non sono conosciute le probabilità di transizione di stato e quindi bisogna stimarle. Esso si basa sul fatto di apprendere la funzione valore e da essa inferire la policy ottimale.
Lavorano sull'idea della General Policy Iteration (GPI) cioè uno schema iterativo composto da due processi. Il primo prova a costruire un'approssimazione della funzione valore basandoci sulla policy corrente (policy evaluetion step) e nel secondo step, la policy viene migliorata (policy improvement step).
Q-Learning inoltre, è un algoritmo temporal difference, poiché calcola l'errore temporale ovvero la differenza tra la nuova e la vecchia stima della funzione valore, considerando la ricompensa ricevuta allo step corrente e utilizzandola per aggiornare la funzione valore.

5.1 Configurazioni

In questa sezione vedremo come l'algoritmo Q-Learning è stato applicato all'ambiente in esame. In questo algoritmo il metodo utilizzato per salvare i valori della funzione valore Q è la forma tabellare essendo questo il metodo più semplice per poter provare questo tipo di algoritmo che è alla base di tutti gli altri.
Un'altra scelta importante è quella relativa allo spazio degli stati e delle azioni. Il nostro ambiente presenta, come abbiamo visto dalle tabelle 1 e 2, uno spazio dello stato e delle azioni continuo, quindi per poter applicare l'algoritmo Q-Learning è necessaria una discretizzazione degli spazi. Questo è necessario perché in questa fase iniziale si è scelto di memorizzare i valori della funzione Q in forma tabellare, la tecnica utilizzata è quella a bucket.
Fissato il numero di bucket per ogni parametro dello stato si sfrutta una funzione che, preso in input lo stato in forma continua, permette la discretizzazione restituendo il bucket a cui viene assegnato. Per le azioni invece, il procedimento è l'opposto, cioè una volta trovata l'azione che massimizza Q in un certo stato, questa è già discreta essendo in forma tabellare bisogna renderla continua per poterla somministrarla all'agente.

Parametro	**Valore**
Dimensione Bucket Stato	(20,20,20,20,20,20,20,20,2,20,20,20,20,20,2)
Dimensione Bucket Azione	$(5, 5, 5, 5)$
Numero Episodi	100000
Learning Rate	0.01
Discount Rate	0.11
Exploration Rate	0.99..0

Tabella 3: Q-Learning Configurazione 1

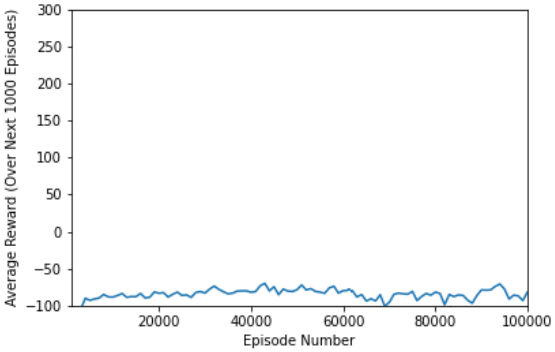
Parametro	Valore
Dimensione Bucket Stato	(4,5,5,5,4,5,4,5,2,4,5,4,5,2)
Dimensione Bucket Azione	(20,20,20,20)
Numero Episodi	100000
Learning Rate	0.01
Discount Rate	0.11
Exploration Rate	0.99..0

Tabella 4: Q-learning Configurazione 2

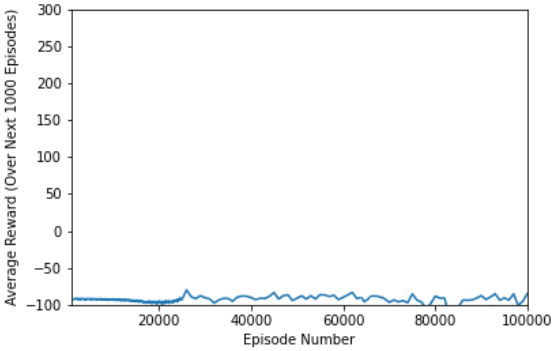
Per questo algoritmo sono state considerate due configurazioni, dove la sostanziale differenza è quella di privilegiare nella configurazione 1 (tabella 3) la dimensionalità dello stato, mentre nella configurazione 2 (tabella 4) la dimensionalità delle azioni.

5.2 Risultati

Come vediamo dalla figura 3 applicando questo tipo di algoritmo al nostro ambiente abbiamo delle prestazioni pessime anche con un numero di episodi cosi elevato.
Questi risultati sono dovuti al fatto che gli spazi degli stati e delle azioni sono stati discretizzati, una possibile soluzione era quella di aumentare la dimensione del bucket, ma poiché lo spazio degli stati è molto grande risulterebbe impossibile salvare tutti questi dati in forma tabellare. La memoria richiesta per salvare tutti questi dati è troppo vasta. Inoltre, anche la sola ricerca nella tabella di un valore in un particolare stato potrebbe risultare computazionalmente proibitiva.
Per superare questi problemi si adotteranno degli approssimatori di funzione per salvare la funzione valore. Esistono vari metodi in reinforcement learning per l'approssimazione di funzione. successivamente verranno trattate le Deep Network per risolvere questo problema.



(a) Configurazione 1



(b) Configurazione 2

Figura 3: Ricompense medie ogni 1000 episodi

6 APPROCCIO DEEP - DQN

Con il termine Deep Reinforcement Learning ci riferiamo semplicemente all'uso di Deep Neural Network (DNN) come approssimatori di funzione per la funzione valore o policy, in algoritmi di Reinforcement Learning.
Il Deep Q-Learning [1] è un evoluzione del metodo Q-Learning dove però la tabella stato-azione viene sostituita da una rete neurale.
In questo algoritmo quindi l'apprendimento non consiste nell'aggiornare la tabella ma consiste nell'aggiustamento dei pesi dei neuroni che compongono la rete attraverso backpropagation.
Come per il Q-Learning il comportamento della policy è dato da un approccio ϵ-greedy per garantire un esplorazione sufficiente.
Questa fase di progettazione quindi si è proposta di riuscire a trovare dei modelli di rete neurale utilizzando l'algoritmo DQN. L'algoritmo DQN ha due differenze importanti rispetto al Q-Learning, oltre ad utilizzare una DNN al posto di una tabella, esso prevede una riproduzione dell'esperienza per addestrare la rete, ovvero vengono memorizzate in un buffer un certo numero di esperienze e si prelevano alcune di esse in modo random in un batch di di-

mensione fissata, quest'ultimo verrà poi utilizzato per addestrare la DNN. Questo meccanismo costituisce un set di dati di input alla rete che è abbastanza stabile per l'addestramento. Campionando casualmente dal buffer delle esperienze, i dati sono indipendenti l'uno dall'altro e più vicini ad essere identicamente distribuiti, escludendo il fatto che alcuni campioni possano soffrire di correlazione.

Algorithm 3 DQN con experience replay

inizializza il buffer B di capacità N
inizializza la rete azione-valore Q con i pesi casuali θ
inizializza la rete target azione-valore $\hat{Q}$ con i pesi $\hat{\theta} = \theta$
for $episodio = 1, M$ **do**
    inizializza lo stato iniziale s_1
    for t = 1,T **do**
        if random(0,1) $\leq \epsilon$ **then**
            seleziona un'azione casuale a_t
        else
            seleziona $a_t = \arg\max_a Q(s_t, a; \theta)$
        end if
        esegui l'azione a_t e osserva la ricompensa r_t e lo stato s_{t+1}
        memorizza l'esperienza (s_t, a_t, r_t, s_{t+1}) in B
        estrai un minibatch di esperienze (s_j, a_j, r_j, s_{j+1}) da B
        $y_t = r_t +$ (1 - done) $\gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \hat{\theta})$
        esegui il gradiente discendente della MSE $(y_j - Q(s_j, a_j; \theta))^2$ rispetto a θ
        if t mod C = 0 **then**
            reset $\hat{Q} = \tau Q + (1 - \tau)\hat{Q}$
        end if
        $s_t = s_{t+1}$
        if done **then**
            print("Episode finished")
            break
        end if
    end for
end for

Un altro aspetto nuovo è quello di utilizzare una rete neurale come approssimatore di funzione al posto di una funzione valore esatta. Con Q-Learning si aggiorna esattamente un valore di stato, mentre con DQN se ne aggiornano molti poiché bisogna aggiornare i pesi della rete. Questo fatto causa una forte instabilità sui valori di azione approssimati per lo stato successivo cosa che non accadeva in Q-Learning.
Questo problema viene risolto introducendo una seconda rete neurale detta target che ha lo scopo di misurare l'errore. Questa rete è una rete che rispetto all'originale è ritardata nel tempo, dando alla rete originale più tempo per considerare altre azioni che hanno avuto luogo di recente invece di aggiornarle continuamente.
Lo pseudo codice di DQN è illustrato dall'algoritmo 3. Si noti che a ciascun timestep il robot ha due possibilità: selezionare un azione casuale o selezionare l'azione generata dal modello. In questo algoritmo non è più necessario discretizzare lo stato perché viene passato direttamente in input alla rete neurale, mentre per quanto riguarda le azioni è necessaria sempre una sorta di discretizzazione in quanto gli output della rete sono comunque dei valori della funzione Q, che vengono poi confrontati per determinare l'uscita più alta e da qui si ricava il numero dell'azione che poi dovrà essere convertita nel formato accessibile all'agente.

6.1 Loss Function

Questa sezione spiega la funzione di loss della rete neurale. Questa funzione è definita come lo scarto quadratico medio, è sostanzialmente usata in problemi di regressione poiché misura la distanza quadratica media tra i valori target e i valori stimato.

$$L_t(\theta) = \frac{1}{n}\sum_{i=1}^{n}[r + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \hat{\theta}) - Q(s_t, a; \theta)]^2 \tag{5}$$

L'apprendimento della funzione valore in DQN è basato quindi sulla modifica dei in funzione della loss function indicata dall'equazione 5, dove $r + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \hat{\theta})$ rappresenta l'expected return ottimo (target) mentre $Q(s_t, a; \theta)$ è il valore stimato dalla rete.
Combinando tutto quanto descritto dall'algoritmo 3, l'obiettivo è ridurre la deviazione tra il valore target e il valore stimato derivata dall'equazione di Bellman. Gli errori calcolati dalla loss function saranno propagati all'indietro nella rete mediane un passo backword (backpropagation), seguendo la logica di discesa del gradiente. Infatti il gradiente indica la direzione di maggior crescita di una funzione e muovendoci in direzione opposta riduciamo

al massimo l'errore.

6.2 Modello DQN

Il modello di rete consiste di quattro livelli hidden tutti con trasformazione lineare.

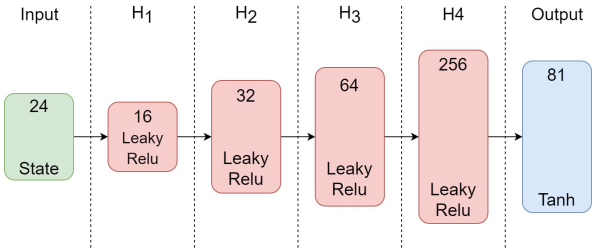


Figura 4: Architettura rete DQN

In particolare, abbiamo in input lo stato quindi 24 valori, il primo livello hidden ha 16 nodi, il secondo ha 32 nodi, il terzo 64 nodi e il quarto 256, in output invece presenta 3^4 poiché si è deciso di mappare ogni dimensione dello spazio delle azioni con tre valori $(-1, 0, 1)$ ottenendo cosi un numero di azioni possibili da far intraprendere all'agente pari a 81. La funzione di attivazione scelta per i primi quattro livelli è Leaky Relu con $negative_{slope} = 0.2$, mentre per il livello di uscita si è scelta la funzione di attivazione *tanh*. La figura 4 mostra l'architettura della rete utilizzata.

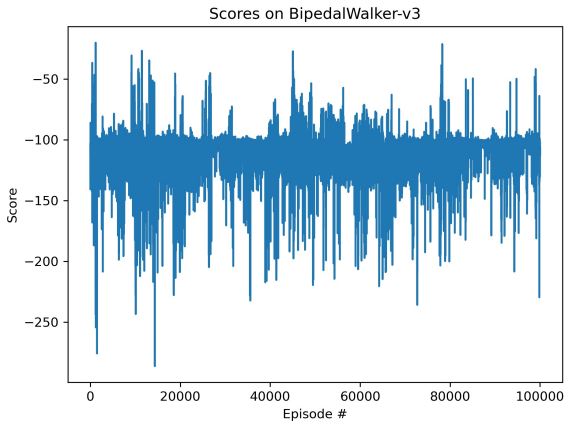
6.3 Risultati

La configurazione riportata nella tabella 5 è risultata essere la migliore per l'algoritmo DQN.

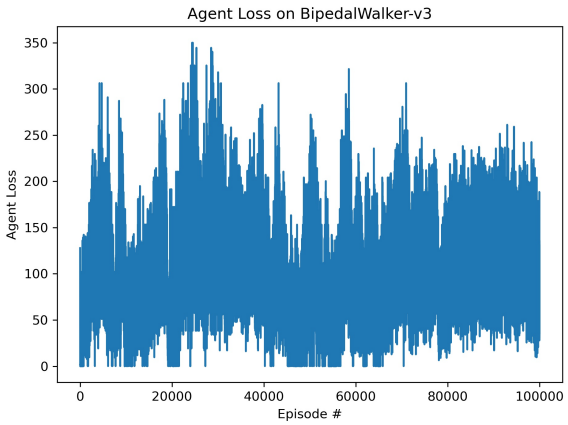
Parametro	**Valore**
Dimensione Replay Buffer	100000
Dimensione Mini Batch	64
Discount Rate γ	0.99
Interpolation Parameter τ	0.01
Learning Rate	0.05
Exploration Rate ϵ	0.99..0.01
ϵ Decay	0.01
Numero Step per Aggiornamento C	4
Numero Azioni	81

Tabella 5: Configurazione DQN

I grafici seguenti illustrano le performance di questo algoritmo applicato all'ambiente BipedalWalker-v3 con la configurazione precedente. Sono stati eseguiti 10000 episodi in circa 8 ore, come si può notare questo tipo di algoritmo, sulla configurazione migliore tra gli esperimenti fatti, non produce il risultato desiderabile. SI può notare che il modello in questione raggiunge una ricompensa media di circa -100.



(a) Ricompensa



(b) Funzione di Loss

Figura 5: Performance DQN

In conclusione, con l'algoritmo DQN non si riesca ad ottenere risultati affidabili. Quindi si è ritenuto necessario adottare metodi più avanzati e architetture di rete più complesse per risolvere questo compito specifico. Inoltre, sembra che tutte le migliorie apportate dal passaggio dal Q-Learning a

DQN non abbiano portato a grandi vantaggi, ma ciò può essere considerato come delle informazioni preliminari e basilari su metodi semplificati che però successivamente sono di notevole importanza per comprendere gli algoritmi più avanzati e complessi.
Le scarse performance ottenute con il DQN possono essere attribuite al fatto che la funzione Q in esame è molto complessa, e questo genera un apprendimento molto complicato aumentando notevolmente i tempi di convergenza dell'algoritmo. Inoltre, con il DQN siamo riusciti ad ovviare alla discretizzazione degli stati, ma comunque una discretizzazione sulle azioni è rimasta, infatti per poter applicare l'algoritmo si è dovuto rendere finito il numero di azioni. Vediamo ora nelle successive due sezioni due modi diversi per poter fare in modo di non discretizzare le azioni e ottenere in uscita dalla rete azioni direttamente somministrabili all'agente.

7 DDPG

Il Deep Deterministic Policy Gradient [5] è una tecnica di reinforcement learning che combina il DQN ai policy gradient PG. Inoltre, questa tecnica prevede l'utilizzo dell'architettura Actor-Critic. Nelle seguenti due sottosezioni vedremo queste due sostanziali caratteristiche del DDPG.

7.1 Policy-Gradient

L'obiettivo di un agente per il reinforcement learning è quello di massimizzare la ricompensa attesa seguendo una politica π. Se rappresentiamo la ricompensa totale per una data traiettoria τ come $r(\tau)$, arriviamo alla definizione espressa dall'equazione 6.

$$J(\theta) = \mathbb{E}_\pi[r(\tau)] \tag{6}$$

L'algoritmo Policy-Gradient ottimizza una politica π, calcolando il gradiente delle stime della ricompensa attesa per l'azione prevista a' seguendo la politica corrente. Secondo il seguente teorema:

Teorema 1 *La derivata della ricompensa attesa è il valore atteso del prodotto tra la ricompensa e il gradiente del logaritmo della policy π.*

$$\nabla\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi[r(\tau) \cdot \nabla \log \pi(\tau)] \tag{7}$$

Espandendo la definizione della politica π nel modo seguente

$$\pi(\tau) = \mathcal{P}(s_0) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t) \tag{8}$$

dove $\mathcal{P}(s_0)$ rappresenta la distribuzione ergodica dell'avvio in uno stato s_0. Applicando la regola del prodotto di probabilità, poiché sono eventi indipendenti, grazie al fatto che sono MDP otteniamo l'equazione seguente.

$$\nabla\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[r(\tau) \cdot \left(\textstyle\sum_{t=1}^{T} \nabla \log \pi(a_t|s_t)\right)\right] \tag{9}$$

L'equazione 9 è il gradiente della politica classico in cui τ indica la traiettoria. Il risultato dell'equazione mostra che non abbiamo bisogno di conoscere la distribuzione ergodica dello stato iniziale $\mathcal{P}$ e né le dinamiche ambientale p. Questo è importante perché nella maggior parte degli scopi pratici, è difficile modellare entrambe queste variabili. L'equazione 9 è il risultato della procedura nota come *Model-Free Algorithm*, a causa del fatto che non si modella l'ambiente.
Un termine che rimane intatto è la ricompensa della traiettoria $r(\tau)$. Tuttavia, è possibile utilizzare la funzione di sconto G che restituisce la ricompensa di ritorno $G_t = r_{t+1} + r_{t+2} + \cdots + r_T$. Quindi sostituendo $r(\tau)$ con G_t possiamo arrivare ad ottenere l'equazione finale dei metodi Policy-Gradient.

$$\nabla\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[G_t \cdot \left(\textstyle\sum_{t=1}^{T} \nabla \log \pi(a_t|s_t)\right)\right] \tag{10}$$

7.2 Approccio Actor-Critic

Dall'equazione 10 possiamo scomporre il valore atteso di G_t in questo modo

$$\nabla\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[\textstyle\sum_{t=1}^{T} \nabla \log \pi(a_t|s_t)\right]\mathbb{E}_\pi[G_t] \tag{11}$$

ricordando che $\mathbb{E}_\pi[G_t] = Q(s_t, a_t)$ possiamo riscrivere l'equazione 11 come segue:

$$\nabla\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[\textstyle\sum_{t=1}^{T} \nabla \log \pi(a_t|s_t) Q(s_t, a_t)\right] \tag{12}$$

Il valore di Q può essere appreso parametrizzando la funzione Q con una rete neurale. Questo porta a metodi actor-critic in cui abbiamo due reti neurali. La rete critic che stima la funzione valore-azione (funzione Q) e la rete actor che aggiorna la distribuzione della politica nella direzione suggerita dalla rete critic (come con il metodo Policy Gradient).
Vediamo ora come tutte queste caratteristiche si inglobano nell'algoritmo DDPG

7.3 Algoritmo DDPG

Nell'algoritmo DDPG l'attore è una rete politica ovvero prende lo stato come input e produce l'azione esatta già nello spazio continuo, a differenza del DQN che da in output il valore della funzione Q, e a differenza del Policy Gradient classico che da output una distribuzione di probabilità delle azioni.
La rete critic (rete Q) invece, è una rete con valori della funzione Q che prende in input lo stato e l'azione e genera il valore Q similmente al DQN.
DDPG è quindi adatto per i problemi in cui l'azione è continua e il "determinismo" [4] si riferisce al fatto che l'attore calcola l'azione direttamente anziché una distribuzione di probabilità sulle azioni.
In DQN l'azione ottimale viene intrapresa selezionando l'azione che massimizza i valori di Q. In DDPG l'attore è una rete politica che fa esattamente questo secondo l'equazione seguente dove μ indica l'azione con il valore Q più alto per lo stato corrente s.

$$\mu(s) = \arg\max_a Q(s, a) \tag{13}$$

Una differenza rispetto agli algoritmi precedenti in cui l'esplorazione era garantita attraverso il meccanismo ϵ-greedy, in DDPG vine promossa attraverso l'aggiunta del rumore gaussiano all'azione determinata dalla politica stessa.
Per stabilizzare l'apprendimento anche in questo tipo di algoritmo si sfrutta la presenza di reti target, sia per la rete critic che per quella actor. Così come succedeva in DQN, queste reti avranno aggiornamenti "soft" basati sempre sulle reti originali. Con la differenza che nel DDPG l'aggiornamento soft avviene ad ogni step, al contrario del DQN.
Anche in questo algoritmo il processo di learning è effettuato attraverso il prelevamento di un minibatch dal replay buffer contenente le esperienze passate.

Algorithm 4 DDPG

1: inizializza il buffer R di capacità M
2: inizializza la rete critic $Q(s, a|\theta^Q)$ con pesi casuali θ^Q
3: inizializza la rete actor $\mu(s|\theta^\mu)$ con pesi casuali θ^μ
4: inizializza le reti target Q' e μ' con i pesi $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
5: **for** *episodio* $= 1, M$ **do**
6: inizializza un processo random $\mathcal{N}$ per l'esplorazione delle azioni
7: ricevi lo stato iniziale s_1
8: **for** t = 1,T **do**
9: seleziona l'azione $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
10: esegui l'azione a_t e osserva la ricompensa r_t e il nuovo stato s_{t+1}
11: memorizza l'esperienza (s_t, a_t, r_t, s_{t+1}) in R
12: estrai un minibatch di N esperienze (s_i, a_i, r_i, s_{i+1}) da R
13: $y_i = r_i + \gamma(1 - d)Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
14: aggiorna la rete critic-Q minimizzando la loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
15: aggiorna la rete actor-μ usando il gradiente della politica deterministica
$\nabla_{\theta^\mu} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$
16: aggiorna le reti target:
$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$
$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$
17: $s_t = s_{t+1}$
18: **if** d **then**
19: print("Episode finished")
20: break
21: **end if**
22: **end for**
23: **end for**

Dall'algoritmo 4 notiamo che per calcolare il valore Q di uno stato (linea 13) l'uscita della rete actor-μ vine immessa nella rete critic-Q. Questo viene fatto solo durante il calcolo della loss che vedremo in seguito. Questo meccanismo è mostrato nella figura 6.

7.4 Loss Function

La funzione di loss per le rete actor-μ è definita dall'equazione 14.

$$J_\mu = \frac{1}{N}\sum_{i=1}^{N} Q(s_i, \mu(s_i)) \tag{14}$$

Essa è definita semplicemente come la somma dei valori Q per gli stati del minibatch. Per calcola-

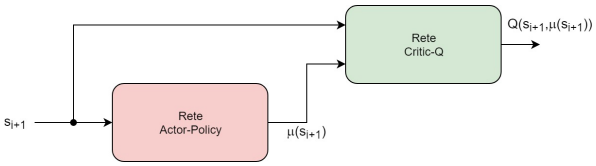


Figura 6: Composizione Actor-μ e Critic-Q

re i valori Q utilizziamo la rete critic e passiamo l'azione calcolata dalla rete actor. Vogliamo massimizzare questo valore poiché desideriamo avere i valori Q massimi.
Invece la funzione di loss per la rete critic-Q è definita dall'equazione 15

$$J_Q = \frac{1}{N}\sum_i (r_i + \gamma(1-d)Q'(s_{i+1}, \mu'(s_{i+1})) - Q(s_i, a_i))^2 \quad (15)$$

dove la perdita della rete critic è un errore TD in cui utilizziamo la rete target per calcolare il valore Q obiettivo per lo stato successivo e le reti originali per calcolare il calore attuale. Dobbiamo minimizzare questa perdita.
Per propagare l'errore all'indietro nella nere abbiamo bisogno della derivata della funzione Q. Quindi partendo dall'equazione 14 di J_μ e trattando μ come costante ma ricordando che per la loss della rete critic la μ è contenuta nel valore Q, utilizziamo la regola della catena, ottenendo il risultato già espresso nell'algoritmo 4 alla linea 15, in particolare otteniamo l'equazione seguente.

$$\nabla_{\theta^\mu} \approx \frac{1}{N}\sum_i (\nabla_\mu Q(s_i, \mu(s_i)) \nabla_{\theta^\mu} \mu(s_i)) \quad (16)$$

7.5 Modello reti neurali

Nel DDPG si è visto che abbiamo 4 reti neurali due actor e due critic di cui una ciascuna è una rete target. L'architettura delle reti target è identica a quelle delle reti originali. Anche per questo algoritmo si è presentato un'architettura presente in letteratura che ha ottenuto le performance migliori.

Come si può vedere dalla figura 7, per la rete actor abbiamo 4 livelli hidden dove i primi due hanno 600 nodi e gli ultimi due 300. La rete actor presenta il livello di input in cui vengono inseriti gli stati (vettore di 24 elementi) e il livello di output

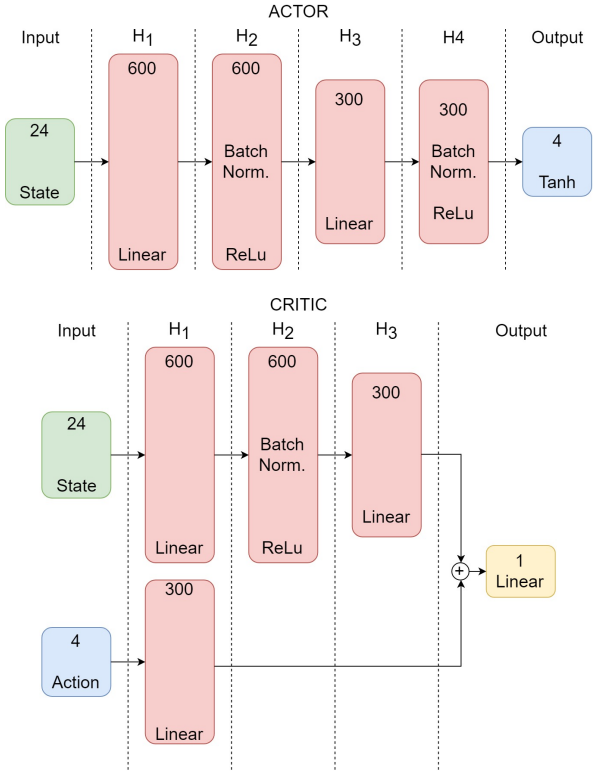
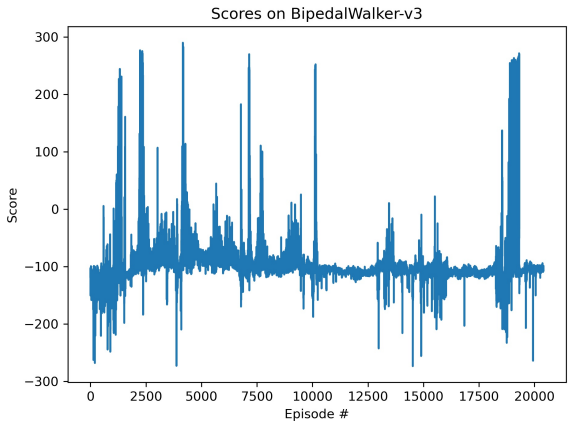


Figura 7: Architettura rete DDPG

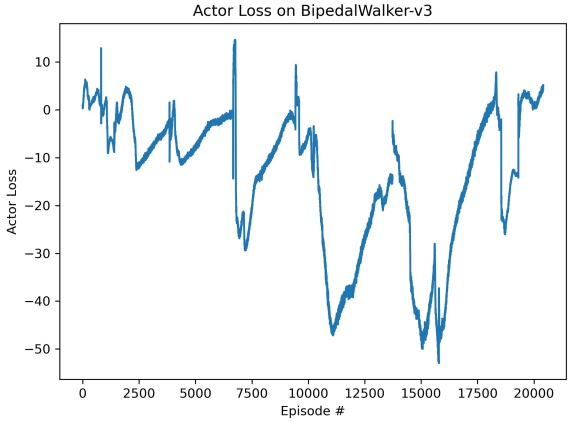
che fornisce le azioni (vettore di 4 elementi).
Invece, per la rete critic è composta da due rami separati inizialmente che poi confluiscono nel livello di output che fornisce il valore Q. Il primo ramo prende in ingresso lo stato ed è composto da 3 livelli hidden rispettivamente formati da 600, 600 e 300 neuroni. Il secondo ramo invece prende in ingresso il vettore delle azioni e presenta un solo livello hidden con 300 neuroni. I livelli indicato con "Linear" applicano la trasformazione lineare del tipo $y = xA^T + b$, mentre quelli con "Batch Normalization" applicano la trasformazione batch del tipo $\frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$.

7.6 Risultati

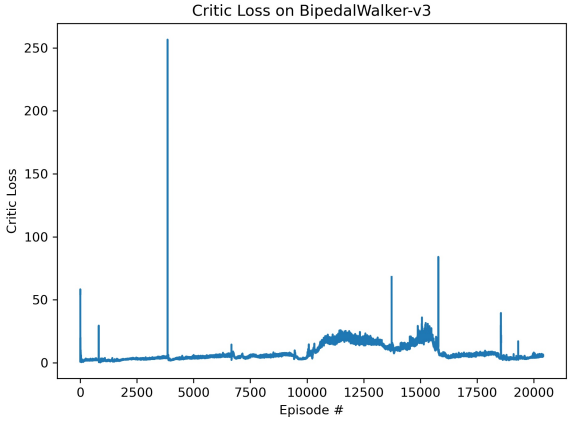
La configurazione riportata nella tabella 6 è risultata essere la migliore per l'algoritmo DDPG. I grafici seguenti mostrano le performance dell'algoritmo DDPG applicato all'ambiente in esame utilizzando la configurazione suddetta.



(a) Ricompensa



(b) Actor Loss



(c) Critic Loss

Figura 8: Performance DDPG

Parametro	**Valore**
Dimensione Replay Buffer	1000000
Dimensione Mini Batch	100
Discount Rate γ	0.99
Interpolation Parameter τ	0.001
Learning Rate Actor	0.0001
Learning Rate Critic	0.001
Weitgth Decay Critic	0.001

Tabella 6: Configurazione DDPG

Il training è stato eseguito per 20000 episodi in un tempo di circa 100 ore. Confrontando il grafico della ricompensa 8(a) con quello 5 dell'algoritmo DQN notiamo che la ricompensa è sempre addensata intorno al valore -100, ma in questo caso raggiungiamo picchi di valori molto vicini alla ricompensa massima. Nonostante ciò l'algoritmo non riesce a convergere.
Per quanto riguarda la funzione di loss delle rete actor bisogna precisare che nella traduzione dalle formule analitiche al codice da implementare la funzione di actor loss è stata cambiata di segno in questo modo anche questa funzione di loss dovrebbe essere minimizzata. Vediamo però che non converge ovvero l'algoritmo non riesce a minimizzare il valore di $-Q$. Mentre per quanto riguarda la critic loss i valori tendono a zero e ciò significa che il valore target è molto prossimo a quello attuale, ma ciò non basta.
Notiamo che anche il DDPG non riesce a risolvere il problema, nonostante un tempo di training molto elevato, probabilmente aumentando gli episodi del training le prestazioni migliorerebbero o addirittura potrebbe raggiungere l'obiettivo.
Quindi il DDPG ha bisogno di un tempo molto elevato per poter vedere dei miglioramenti selle performance su questo tipo di environment.
Il training nel DDPG è instabile come visto nella funzione di loss, inoltre esso è fortemente dipendente dalla ricerca dei parametri ipertestuali[8]. Ciò è dipeso dal fatto che l'algoritmo stima continuamente i valori della rete critic-Q. Questi errori di stima si accumulano nel tempo e possono portare l'agente in un ottimo locale o addirittura ricadere in prestazioni pessime.
Per questi motivi si è scelto di superare l'algoritmo DDPG per una sua versione più avanzata e

sofisticata che risolve le problematiche precedenti.

8 Twin Delayed DDPG

Twin Delayed DDPG[6] (TD3) è l'algoritmo "successore" del DDPG La scelta di adottare l'algoritmo TD3 è motivata dal fatto che quest'ultimo risolve quei problemi che affliggevano il DDPG. In particolare, TD3 si concentra sulla riduzione della distorsione da sovrastima riscontrata nell'algoritmo precedente. Questo problema viene risolto con l'aggiunta di tre funzioni chiave.
La prima caratteristica aggiuntiva di TD3 è l'uso di due reti critiche. Ciò è stato ispirato dalla tecnica vista in Double Q-Learning[2]. TD3 usa una doppia funzione di apprendimento Q dove si considera poi il valore *più piccolo* delle due reti critiche quando si generano gli obiettivi. La minore delle due stime causerà meno danni agli aggiornamenti della politica. In questo modo si favorisce la sottostima dei valori Q. La distorsione generata da questa sotto stima non è un problema in quanto i valori bassi non verranno propagati attraverso l'algoritmo, a differenza dei valori alti. Inoltre, si fornisce un'approssimazione più stabile, migliorando così la stabilità dell'intero algoritmo.
Le reti target sono un ottimo strumento per introdurre stabilità nella formazione dell'agente, tuttavia nel caso dei metodi actor critic ci sono alcuni problemi. Ciò è dovuto dall'interazione tra le reti actor-policy e critic-value. Il training dell'agente diverge quando una politica scarsa viene sovrastimata. La politica dell'agente continuerà a peggiorare poiché si sta aggiornando sugli stati con molti errori. Per risolvere questo problema, gli aggiornamenti sulle reti politiche si eseguono meno frequentemente della rete dei valori. Ciò consente alle rete di valori di diventare più stabile e ridurre gli errori prima di essere utilizzata per aggiornare la rete delle politiche. In pratica, la rete delle politiche viene aggiornata dopo un determinato periodo di fasi temporali, mentre la rete dei valori continua ad aggiornarsi dopo ciascuna fase temporale. Questi aggiornamenti delle politiche meno frequenti avranno una stima del valore con una varianza inferiore e pertanto dovrebbero tradursi in una politica migliore.

Algorithm 5 TD3

inizializza il buffer R di capacità M
inizializza le reti critic $Q_{\theta_1}, Q_{\theta_2}$ con pesi casuali θ_1, θ_2
inizializza la rete actor μ_ϕ con pesi casuali ϕ
inizializza le reti target $Q'_{\theta'_1}, Q'_{\theta'_2}$ e $\mu'_{\phi'}$ con i pesi $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$ e $\phi' \leftarrow \phi$
for $episodio = 1, M$ **do**
    inizializza un processo random $\epsilon \sim \mathcal{N}(0, \sigma)$ per l'esplorazione delle azioni
    ricevi lo stato iniziale s_1
    for t = 1,T **do**
        seleziona l'azione $a_t = \mu_\phi(s_t) + \epsilon$
        esegui l'azione a_t e osserva la ricompensa r_t e il nuovo stato s_{t+1}
        memorizza l'esperienza (s_t, a_t, r_t, s_{t+1}) in R
        $s_t = s_{t+1}$
        if d **then**
            print("Episode finished")
            break
        end if
    end for
    for $n = 1, t$ **do**
        estrai un minibatch di N esperienze (s_i, a_i, r_i, s_{i+1}) da R
        aggiungi rumore all'azione in input alle reti critiche $\tilde{a} = \mu'_{\phi'}(s_{t+1}) + \epsilon$
        $y_i = r_i + \gamma(1 - d) \min_{j=1,2} Q'_{\theta'_j}(s_{i+1}, \tilde{a})$
        aggiorna la reti critic Q_{θ_j} $\forall j = 1, 2$ minimizzando le loss: $L_j = \frac{1}{N} \sum_i (y_i - Q_{\theta_j}(s_i, a_i))^2$
        if n mod $policydelay = 0$ **then**
            aggiorna la rete actor-μ usando il gradiente della politica deterministica $\nabla_\phi \approx \frac{1}{N} \sum_i \nabla_a Q_{\theta_1}(s, a)|_{s=s_i, a=\mu_\phi(s_i)} \nabla_\phi \mu_\phi(s)|_{s=s_i}$
            aggiorna le reti target:
            $\theta'_j \leftarrow \tau\theta_j + (1 - \tau)\theta'_j$ $\forall j = 1, 2$
            $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
        end if
    end for
end for

La parte finale di TD3 esamina l'appiattimento della politica di destinazione. I metodi di politica deterministica hanno la tendenza a produrre valori target con elevata varianza durante l'aggiornamento della rete critic. Ciò è causato da un overfitting di adattamento ai massimi locali nella stima del valore. Al fine di ridurre questa varianza, TD3 utilizza una tecnica di regolarizzazione delle politiche di destinazione nota come smoothing. Idealmente vorremmo che non ci sia alcuna varianza tra i valori target, cioè con azioni simili vorremo valori simili.

TD3 riduce questa varianza aggiungendo una piccola quantità di rumore casuale ai valori target e calcolando la media su mini batch. L'intervallo di rumore viene limitato per mantenere il valore target vicino all'azione originale. In questo modo il training con il rumore aggiunto fa in modo di regolarizzare le azioni degli agenti favorendo una politica più solida.
Abbiamo visto le migliorie di TD3 rispetto al suo predecessore DDPG, ora vediamo come questi aspetti vengono implementati. A questo scopo l'algoritmo 5 mostra lo pseudo codice di TD3.

8.1 Funzione di Loss

Descriviamo ora come vengono costruite le funzioni di loss per le reti in questione che poi serviranno all'aggiornamento dei pesi delle stesse. La funzione di loss per le rete actor-μ è esattamente uguale al caso di DDPG (equazione 14). Ricordiamo che vogliamo massimizzare la actor-loss questo valore poiché desideriamo avere i valori Q massimi.
Stessa cosa per le funzione di loss per le reti critic-Q, (equazione 15) l'unica differenza è quella che nel calcolo del valore Q obiettivo si utilizza la rete critic che ha il Q-value minore.

8.2 Modello reti neurali

Anche in questo caso abbiamo 2 architetture diverse per le 6 reti in gioco, una per tutte le reti actor e una per tutte le reti critic. Sono state analizzate due diverse scelte di architetture, dove sostanzialmente il numero di livelli è lo stesso varia nel numero di neuroni presenti in ogni livello hidden. La configurazione 1 è rappresentata in figura 9, è una struttura molto semplice in cui i livelli hidden di actor e critic hanno lo stesso numero di neuroni.
La versione 2 invece prevede prevede per la rete actor 480 nodi in H_1 e 240 in H_2, mentre per la rete critic ci sono 560 nodi in H_1 e 280 in H_2.
Tutti i livelli hidden sono composti da nodi che applicano la trasformazione lineare con funzione di attivazione di tipo Relu.

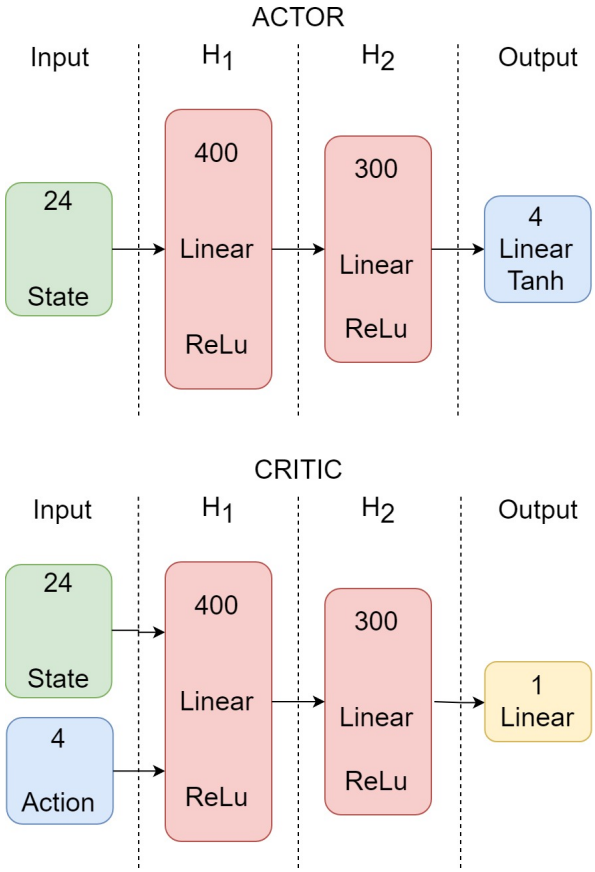


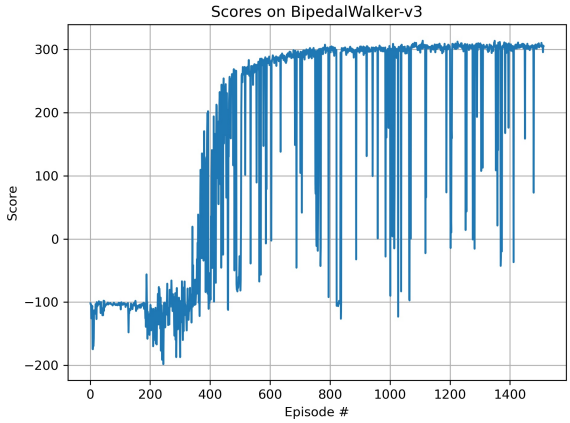
Figura 9: Architettura rete TD3

8.3 Risultati

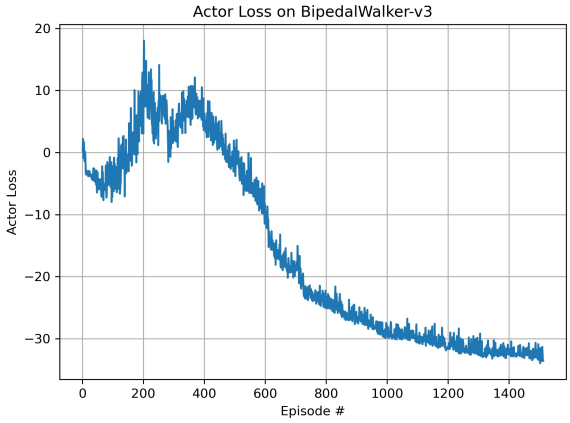
Parametro	Valore
Dimensione Replay Buffer	500000
Dimensione Mini Batch	100
Discount Rate γ	0.99
Interpolation Parameter τ	0.005
Learning Rate Actor	0.001
Learning Rate Critic	0.001
Policy Delay	2
Policy Noise	(-0.5,0.5)

Tabella 7: Configurazione TD3

Il settaggio degli iperparametri è stato identico per le due versioni della rete, è illustrato nella tabella seguente. I grafici seguenti mostrano le performance dell'algoritmo TD3 applicato all'ambiente

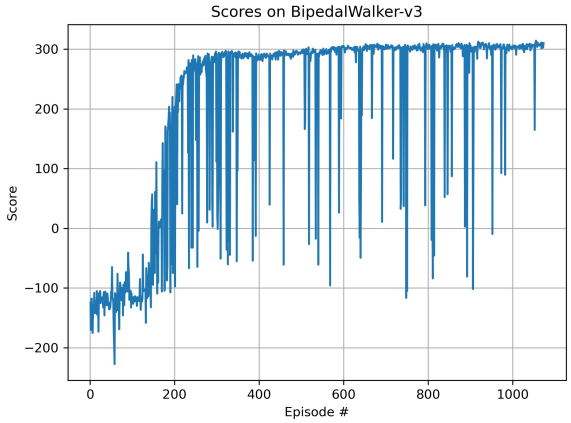


(a) Ricompensa

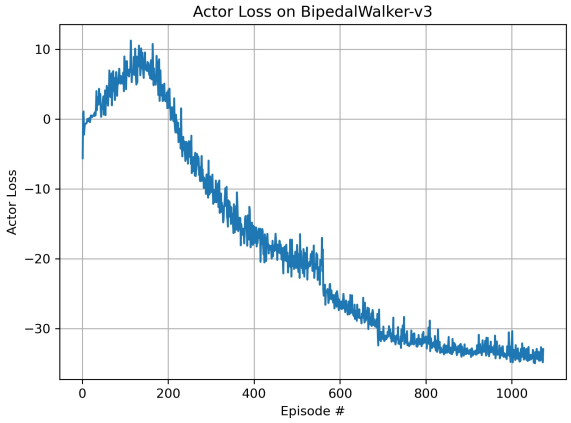


(b) Funzione di Loss

Figura 10: Performance TD3 Architettura 1



(a) Ricompensa



(b) Funzione di Loss

Figura 11: Performance TD3 Architettura 2

BipedalWalker-v3 Notiamo che già con questa prima configurazione l'algoritmo TD3 risolve il problema all'episodio 1512 in circa 12 ore di training, ovvero ha una media della ricompensa maggiore o uguale di 300 per 100 episodi consecutivi. Quindi l'algoritmo riesce a trovare una politica ottimale che consente di raggiungere l'obiettivo. La seconda configurazione presenta le performance mostrate in figura 11, rispetto alla precedente quest'ultima raggiunge l'obiettivo in 1073 episodi in un training durato circa 9 ore. Si nota quindi che la seconda configurazione sebbene leggermente più pesante dal punto di vista computazionale raggiunge la convergenza in minor tempo e in minor episodi. Queste due simulazioni hanno raggiunto i massimi risultati e di gran lunga superiori a tutti gli altri esperimenti. Dai due grafici sulla ricompensa vediamo che la seconda soluzione riesce ad assestarsi intorno ai 300 step già ai 300 episodi, mentre la configurazione 1 li raggiunge soltanto intorno ai 700. Inoltre, è possibile osservare che nella soluzione 1 ci sono molti più picchi negativi rispetto alla soluzione 2. Per quanto riguarda la loss invece, la seconda configurazione ha un picco iniziale più basso e un valore finale più basso della configurazione 2.

9 CONCLUSIONI

La figura 11 visualizza il risultato di quella che risulta la politica ottimale. Al contrario di tutti gli altri algoritmo il TD3 riesce a raggiungere l'obiettivo a differenza degli altri considerati, infatti il DDPG riesce in alcuni episodi a raggiungere la ricompensa massima ma non riesce a raggiungere la convergenza. Questo è attribuibile al fatto che DDPG presenta molte limitazioni sul compito specifico, tuttavia una suo miglioramento il TD3 è stato in grado di riuscire ad addestrare con successo l'agente. Il successo finale è dovuto a quelle differenze specifiche tra il TD3 e DDPG, presentate nel paragrafo 8 che hanno permesso di stabilizzare l'apprendimento riducendo la varianza. Inoltre, TD3 risolve il problema degli errori di stima che si accumulano nel tempo e possono portare l'agente in ottimi locali. Nella tabella seguente sono riportati i risultati finali che riassumono gli esperimenti effettuati con tutti gli algoritmi.

Algoritmo	**Score Medio Max**	**Score Massima**	**Episodi**	**Tempo in ore**
Q-Learning	-76	-50	10000	12
DQN	-100	-20	10000	8
DDPG	43	280	2000	100
TD3	301	305	1073	9

Tabella 8: Configurazione TD3

Tutte le simulazione sono state eseguite su macchine *Colab*. Il codice sorgente degli esperimenti riportati in questo documento è disponibile alla repository GitHub [9].

Riferimenti bibliografici

[1] C.J.C.H Watkins. «Learning from Delayed Reward». PhD thesis. Cambridge University, 1989.

[2] Hado V. Hasselt. «Double Q-learning». In: *Advances in Neural Information Processing Systems 23*. A cura di J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: `http://papers.nips.cc/paper/3964-double-q-learning.pdf`.

[3] Andrew G Sutton Richard S.; Barto. *Reinforcement Learning: An Introduction*. 2013.

[4] Silver et al. «Deterministic Policy Gradient Algorithms». In: *proceedings.mlr.press* (2014).

[5] Lillicrap et al. «Continuous control with deep reinforcement learning». In: *arXiv.org* (2015).

[6] Fujimoto et al. «Addressing Function Approximation Error in Actor-Critic Methods». In: *arXiv.org* (2018).

[7] OpenAI. *Deep Deterministic Policy Gradient*. 2018. URL: `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[8] OpenAi. *Twin Delayed DDPG*. 2018. URL: `https://spinningup.openai.com/en/latest/algorithms/td3.html`.

[9] Ciavola C. *BipedalWalker-v3*. 2020. URL: `https://github.com/claudeHifly/BipedalWalker-v3`.