Multiagent Systems: Implementazione ed analisi dell'algoritmo Experience Replay

Alessandro Arezzo
E-mail address
alessandro.arezzo@stud.unifi.it

Abstract

L'elaborato descritto nel merito della presente relazione è stato realizzato come progetto per l'insegnamento Multiagent Systems previsto dal corso di laurea magistrale in ingegneria informatica dell'Università degli studi di Firenze. Il progetto prevede l'implementazione in linguaggio di programmazione Python del lavoro riassunto nell'articolo Experience Replay for Real-Time Reinforcement Learning Control [1]. L'algoritmo Experience Replay descritto nel paper è stato applicato nello specifico per la risoluzione dei due problemi di controllo real-time relativi alla stabilizzazione di un pendolo e di un manipolatore robotico a due collegamenti in una posizione di equilibrio. Sono quindi stati effettuati dei test attraverso i quali poter comprendere i benefici apportati dall'applicazione dell'algoritmo ai due contesti considerati.

1. Introduzione

L'elaborato oggetto di questa relazione consiste nell'implementazione in linguaggio di programmazione Python del lavoro descritto all'interno dell'articolo *Experience Replay for Real-Time Reinforcement Learning Control* [1].

Nello specifico, il progetto ha previsto in primo luogo l'implementazione di un particolare algoritmo di Reinforcement Learning noto come **Experience Replay**, riportato nel suddetto paper ed illustrato nel dettaglio nel corso del successivo paragrafo della presente relazione.
Successivamente a ciò, tale metodologia implementata è stata applicata a due particolari problemi di controllo real-time, anch'essi considerati tra quelli in analisi nell'articolo in questione. Il più banale tra i due consiste nella stabilizzazione di un pendolo in una posizione di equilibrio, mentre l'altro, il quale prevede invece la stabilizzazione di un manipolatore robotico a due collegamenti, ne rappresenta fondamentalmente una variante su più dimensioni.

Terminata infine la fase relativa all'implementazione, per ciascuna delle due dinamiche considerate, si è eseguita una fase di testing atta a valutare le performance rilevate dall'applicazione dell'algoritmo Experience Replay implementato in linguaggio Python.

2. Reinforcement Learning ed Experience Replay

L'obiettivo del lavoro svolto è stato quello di analizzare ed implementare l' algoritmo **Experience Replay (ER)** descritto all'interno dell'articolo citato in precedenza [1]. Il metodo in questione, appartiene nello specifico ad una particolare branchia dell'intelligenza artificiale che risponde al nome di **Reinforcement Learning (RL)**, la quale rappresenta a sua volta una tecnica di machine learning che consente di realizzare agenti autonomi, capaci cioè di scegliere ad ogni istante di tempo la *miglior* azione tra tutte quelle possibili.

Prendendo in considerazione un agente definito da un *modello di decisione di Markov*, si ha che questo è in primo luogo caratterizzato da uno spazio degli stati X, da uno spazio delle azioni U e da una funzione di transizione di probabilità $f\colon \mathrm{X}\times\mathrm{U}\times\mathrm{X}\longrightarrow[0,\infty]$ atta a fornirgli informazione, generalmente non deterministica, nei confronti dell'ambiente col quale interagisce. Obiettivo di un agente autonomo è quindi quello di scegliere ad ogni istante di tempo t, a seconda dello stato $x(t) \in X$ in cui questi si trova, l'azione $u(t)$ *migliore* tra quelle appartenenti allo spazio delle azioni.
La misura relativa alla bontà correlata alla scelta di un'azione piuttosto che di un'altra, è stabilita da una funzione detta *reward*, la quale è definita come $r\colon \mathrm{X}\times\mathrm{U}\times\mathrm{X}\longrightarrow\mathrm{R}$.
Tenendo conto di ciò, l'agente è in grado di definire una *policy* $h\colon \mathrm{X}\longrightarrow\mathrm{U}$ che definisce l'azione applicata dall'agente a seconda dello stato in cui questi si trova.
Se il modello di decisione di Markov è deterministico, come nei casi analizzati per il lavoro qua illustrato, la funzione di transizione è definita semplicemente come $f\colon \mathrm{X}\times\mathrm{U}\longrightarrow[0,\infty]$, la quale stabilisce lo stato in cui

l'agente si troverà una volta applicata una certa azione a partire da un determinato stato di partenza. Il reward di un agente in un sistema deterministico sarà invece definito come r: X×U⟶R, il che ci consente di comprendere come questo sia in tal caso funzione solamente della coppia di valori composta dallo stato di partenza e dall'azione applicata a partire da tale stato. Applicando l'azione u_k nello stato x_k, l'agente otterrà quindi un reward pari a $r_{k+1} = r(x_k, u_k)$.

L'obiettivo del RL è quindi nello specifico quello di permettere ad un agente autonomo, per mezzo dell'apprendimento ricavato dalla sua conoscenza passata, di definire una policy h che gli garantisca ad ogni istante di tempo di prendere una decisione ottima in termini di massimizzazione del reward futuro. Prendendo in considerazione esclusivamente **problemi deterministici ad orizzonte infinito**, ai quali si era interessati per lo svolgimento del presente lavoro, per far si che ciò avvenga, ad ogni possibile policy h, viene associata una funzione detta *value function*, la quale restituisce per ogni stato ad ogni istante di tempo il reward futuro ottenuto prendendo ad ogni istante t la decisione $u(t)$ per mezzo dell'applicazione della policy h in questione. Tale funzione è quindi definita come $V^h(x(t), t) = r(x(t), u(t)) + \gamma V^h(x(t+1), t+1)$ con $\gamma \in [0, 1)$ parametro detto *fattore di sconto*, non presente per problemi ad orizzonte finito ma la cui introduzione nell'espressione risulta invece necessaria ad orizzonte infinito in quanto garantisce in primo luogo che la serie non diverga, ed inoltre consente di attribuire peso via via minore ai reward ottenuti più avanti nel futuro. L'obiettivo delle tecniche di RL si riduce quindi nel ricavare tra tutte le policy possibili, quella che massimizza la value function, in quanto questa se applicata consentirà ovviamente all'agente di ottenere una maggiore ricompensa totale. Per far ciò, le tecniche studiate definiscono quindi la value function ottima V^o risolvendo l'equazione $V^o(x(t), t) = \max_{u(t)}\{r(x(t), u(t)) + \gamma V^o(x(t+1), t+1)\}$ detta *equazione di Bellman della programmazione dinamica* . Una volta ricavato il valore V^o per ogni stato, è sufficiente quindi definire la policy ottima come

$$h^o(x, t) = \underset{u(t)}{\operatorname{argmax}}\{r(x(t), u(t)) + \gamma V^o(x(t+1), t+1)\}.$$

Il punto cruciale quindi si pone proprio nella metodologia utilizzata per il calcolo della soluzione stazionaria $V^o(x(t))$, la quale può essere ricavata utilizzando due approcci alternativi: la **value iteration** o la **policy iteration**.

2.1. Q-Learning

Il presente lavoro, si concentra nel dettaglio su di una tecnica detta **Q-learning** facente parte della prima categoria citata: la value iteration. Quest'ultima prevede innanzitutto l'introduzione della funzione

$$Q^o(x(t), u(t)) = r(x(t), u(t)) + \gamma V^o(x(t+1))$$

detta *action value function*, la quale fornisce il reward ottenuto applicando le decisioni migliori dal tempo $t+1$ in poi e l'azione $u(t)$ al tempo t in cui l'agente si trova nello stato $x(t)$. Una volta definita la action value function quindi, sia V^o che h^o possono essere riformulate in funzione di tale Q^o come $V^o(x(t)) = \max_{u(t)} Q^o(x(t), u(t))$ e $h^o(x(t) = \underset{u(t)}{\operatorname{argmax}} Q^o(x(t), u(t))$.

Riuscendo a calcolare il valore di $Q^o(x(t), u(t))$ per ogni possibile coppia stato/azione, è possibile perciò determinare la policy ottima senza conoscere nè la funzione di reward nè quella di transizione (si tratta difatti di una tecnica model free). E' a tal punto sufficiente determinare come tale policy ottima, una dinamica che ad ogni istante di tempo sceglie di applicare l'azione $u(t)$ che massimizza il valore di $Q^o(x(t), u(t))$ con $x(t)$ stato corrente.

Per quanto concerne il calcolo di Q^o, questo può essere effettuato per mezzo di una procedura di *value iteration asincrona*, la quale iterativamente aggiorna il valore di una singola coppia x_k,u_k. Partendo quindi da una $Q_0(x, u)$ iniziale, per $k = 0, 1, ...$ applica la seguente procedura:

- Genera una coppia (x_k, u_k)
- Effettua un passo di programmazione dinamica: $Q_{k+1}(x_k, u_k) = r(x_k, u_k) + \gamma \max_{u(t+1)} Q_k(x(t+1), u(t+1))$ con $x(t+1) = f(x_k, u_k)$
- Lascia invariati i valori relativi alle altre coppie stato/azione della action value function successiva: $Q_{k+1}(x, u) = Q_k(x, u) \forall (x, u) \neq (x_k, u_k)$

Purchè tutti gli stati (x, u) in $X \times U$ siano visitati un numero sufficiente di volte, l'algoritmo converge e si ha quindi che

$$\lim_{k \to \infty} Q_k(x, u) = Q^o(x, u)$$

Nel caso in cui il prodotto cartesiano $X \times U$ sia un insieme discreto, si ha quindi semplicemente che dopo un sufficiente numero di iterazioni la procedura sopra riportata restituirà una matrice Q^o la quale associa ad ogni $x \in X$, il reward ottimo futuro ottenibile a seconda dell'azione u applicata all'istante corrente. In tal caso per definire una policy ottima è quindi sufficiente definire $h^o(t)$ in modo che ad ogni istante t, questa selezioni l'azione $u(t)$ a cui è associato il valore maggiore nella matrice Q^o in corrispondenza della riga relativa allo stato corrente $x(t)$.

Ovviamente ciò non può essere valido nei casi per i quali $X \times U$ non risulti essere di cardinalità trattabile, casi questi ai quali siamo interessati per il presente lavoro. Per tali problemi, occorre perciò ricorrere a tecniche di *Q-learning approssimato*, le quali prevedono la rappresentazione della action value function per mezzo dell'introduzione di una funzione approssimante: $Q^o(x(t), u(t)) \simeq \widetilde{Q}(x(t), u(t), \theta)$. Al tempo t quindi con

tali tecniche viene minimizzata la variazione tra la corretta action value function e la funzione approssimante per mezzo di un passo di discesa del gradiente che vada a modificare i valori dei parametri $\theta(t+1)$ sulla base di $x(t), u(t), r(t), x(t+1)$.
Nello specifico, per l'elaborato presentato, si è optato per la soluzione riportata nell'articolo [1], la quale prevede la definizione di una funzione approssimante lineare del tipo: $\widetilde{Q}(x(t), u(t)) = \phi^T(x,u)\theta$ con θ vettore dei parametri e $\phi\colon X \times U \longrightarrow R^n$ vettore delle funzioni di base (BFs). Ad ogni iterazione perciò questa verrà aggiornata sulla base dei valori $x_k, u_k, r_{k+1}, x_{k+1}$ relativi all'esempio selezionato all'iterazione k_{-esima} in modo che
$\theta_{k+1} = \theta_k + \alpha_k[r_{k+1} + \gamma \max_{u'} \phi(x_{k+1}, u')^T\theta_k - \phi(x_k, u_k)^T\theta_k]\phi(x_k, u_k)$
con $\alpha \in (0,1)$ parametro detto *learning rate*.

Come evidenziato nel paper [1], la convergenza nei metodi di Q-learning approssimato non è garantita, a meno che la policy utilizzata in fase di scelta delle azioni da eseguire non sia fissata.

2.2. Experience Replay

Le tecnice di Q-learning approssimato fin qua presentate, risultano essere efficienti dal punto di vista computazionale, ma come ampiamente evidenziato nell'articolo [1], queste non consentono di esserlo in egual modo dal punto di vista dell'utilizzo dei dati. Questi vengono difatti definiti come approcci *data inefficient* in quanto i dati vengono utilizzati da tali algoritmi una sola volta per poi essere scartati, il che può causare un peggioramento nelle performance soprattutto in casi in cui l'ordine di arrivo di questi ultimi non sia completamente casuale ma bensì dati successivi siano fortemente correlati tra di loro. In tali situazioni, non valgono i risultati di convergenza del gradiente stocastico ed occorre perciò ricorrere a strategie alternative, quali ad esempio l'utilizzo dell'algoritmo **Experience Replay**, l'analisi del quale risulta come preannunciato essere alla base dell'elaborato svolto.

Tale metodologia, aumenta l'efficienza nell'utilizzo dei dati per mezzo della loro memorizzazione e del loro continuo riutilizzo in fase di apprendimento online. L'algoritmo, riportato in Figura 1, mostra come la procedura iterativa di apprendimento sia piuttosto banale. Questa si suddivide in gruppi di iterazioni, ognuno dei quali prevede la generazione di una *traiettoria*, intesa come simulazione di una dinamica che seleziona ad ogni passo l'azione da effettuare utilizzando una strategia nota come $\epsilon - Greedy$. Ad ogni iterazione k viene difatti innanzitutto selezionata un'azione u_k che con una certa probabilità è scelta casualmente nello spazio delle azioni oppure è selezionata come quella che massimizza la funzione approssimante della action value function per lo stato corrente x_k. Tale azione viene quindi applicata nella simulazione, con

Algorithm 1 Experience replay RL

Input: underlying RL algorithm LEARN,
number of replays N, length of each trajectory T,
BFs $\phi_1, \ldots, \phi_n$, discount factor γ, learning rate α,
exploration probability ε

initialize θ arbitrarily (e.g., identically 0)
$D \leftarrow \emptyset$; $l \leftarrow 1$; $k \leftarrow 0$
observe initial state x_0
for every time step k **do**
  $u_k \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} \phi^{\mathrm{T}}(x_k, \bar{u})\theta & \text{w.p. } 1-\varepsilon \\ \text{a uniform random action in } U & \text{w.p. } \varepsilon \end{cases}$
  apply u_k, observe resulting x_{k+1} and reward r_{k+1}
  compute transition index within current trajectory:
    $\tau \leftarrow k-(l-1)T$
  add transition sample to the database:
    $D \leftarrow D \cup \{(k, l, \tau, x_k, u_k, x_{k+1}, r_{k+1})\}$
  $k \leftarrow k+1$
  if $k = lT$ (a T-step trajectory was collected) **then**
    update Q-function parameters:
      $\theta \leftarrow \text{LEARN}(\theta, D, N, T, l, \alpha, \gamma)$
    $l \leftarrow l+1$
  end if
end for

Figure 1. Algoritmo Experience Replay

Algorithm 2 Q-LEARN-SAMPLES$(\theta, D, N, T, l, \alpha, \gamma)$

loop NlT times
  retrieve a random sample $(k, l', \tau, x, u, x', r)$ from D,
    using a uniform distribution
  $\theta \leftarrow \theta + \alpha[r + \gamma \max_{u'} \phi(x', u')^{\mathrm{T}}\theta - \phi(x,u)^{\mathrm{T}}\theta] \cdot \phi(x,u)$
end loop
Output: θ

Figure 2. Metodo Q-LEARN-SAMPLES utilizzato nel progetto per la fase di training della action value function

la conseguenza che se ne ricava il valore dello stato futuro x_{k+1} ed il reward $r_{k+1} = r(x_k, u_k)$. L'esempio corrente composto da tali valori e dai vari indici che ne indicano la posizione all'interno della simulazione, vengono quindi memorizzati nel dataset. Nel momento in cui si conclude l'esecuzione di una traiettoria, la cui lunghezza viene specificata tra i dati in ingresso all'algoritmo, viene invocato un metodo, *LEARN*, atto ad utilizzare gli esempi acquisiti fino a quel punto per l'aggiornamento del vettore di parametri θ, così da ridurre la variazione tra la corretta action value function ottima e la funzione approssimante. A seguito di tale processo di apprendimento, la procedura viene ripetuta con la traiettoria successiva.

Ciò che può contraddistinguere l'algoritmo presentato è la tecnica di implementazione del metodo *LEARN*. Per il presente elaborato, si è deciso di utilizzare l'algoritmo *Q-LEARN-SAMPLES* mostrato in Figura 2, il quale rappresenta una delle due varianti proposte nell'articolo [1]. Come si nota da una sua analisi, questo prevede l'addestramento del modello per un numero di volte pari al prodotto tra il numero dei dati contenuti in memoria al tempo corrente

ed un certo fattore N (numero di replays) fornito in input all'algoritmo. Ad ogni iterazione viene selezionato un esempio casualmente dal dataset e questo viene utilizzato per la modifica del vettore dei parametri θ attraverso l'applicazione di un passo del metodo del gradiente.

Per quanto riguarda la convergenza, come detto nei metodi di Q-learning approssimato, questa è garantita, a meno di ipotesi più deboli, solamente se la policy utilizzata per la scelta delle azioni è fissata. Con l'algoritmo Experience Replay, si nota quindi come, il fatto di utilizzare durante una traiettoria la medesima policy per poi eseguire l'aggiornamento della stessa solamente tra una fase di iterazioni e l'altra, consenta di aumentare la stabilità rispetto a quella rilevata negli algoritmi di Q-learning approssimato.

3. Implementazione

Una volta eseguita l'analisi dell'algoritmo Experience Replay, riportata nel precedente capitolo, affinchè fosse possibile sperimentarne il comportamento e valutarne le performance, si è optato per la realizzazione di una sua implementazione in linguaggio di programmazione Python.
Il progetto si articola nello specifico su più moduli, ognuno dei quali implementa una delle varie componenti richieste dall'algoritmo e le cui caratteristiche sono discusse di seguito nel dettaglio.

3.1. Experience Replay

Ruolo centrale del lavoro, è sicuramente rappresentato dall'implementazione dell'algoritmo ER, la quale è incapsulata all'interno del progetto nella classe *ExperienceReplay* del file *experience_replay_learning.py*. Tale classe mantiene come attributi, oltre ai dati di input richiesti dall'algoritmo ER (in Figura 1), un oggetto che rappresenta l'approssimatore lineare della action value function, uno che mantiene in memoria il dataset di esempi e ne consente la lettura/scrittura, ed una variabile all'interno della quale è memorizzata la dinamica.
Utilizzando tali attributi, le cui classi che ne definiscono i tipi sono illustrate di seguito, questa implementa poi i seguenti metodi:

- **trainModel()**: consente di addestrare il modello eseguendo l'algoritmo ER riportato in Figura 1. Tra i dettagli di implementazione non presenti nello schema in figura, occorre denotare in primo luogo che ciascuna traiettoria ha origine in uno stato casuale, e che inoltre al termine di ogni processo di addestramento vengono calcolate le performance della policy corrente.

- **Q_Learn_Samples()**: implementa il metodo in Figura 2, il quale viene utilizzato per l'addestramento dell'approssimatore lineare durante la fase di training. Si noti come al termine dell'aggiornamento dei parametri, questi vengano salvati in un file ".csv" così che il modello generato durante la corrente computazione possa essere riutilizzato in futuro.

- **simulate()** a partire da uno stato iniziale ricevuto come parametro simula un' esecuzione della dinamica scegliendo ad ogni iterazione l'azione che massimizza la funzione approssimante della action value function. Genera in output un file con estensione ".csv" che ne descrive l'andamento per mezzo della rappresentazione degli stati visitati, delle azioni applicate e dei reward rilevati ad ogni iterazione,.

- **learningPerformance():** consente di calcolare le performance del modello intese come il reward totale medio rilevato durante delle simulazioni fatte partire da un insieme di stati di test. Il metodo viene lanciato durante la fase di training ogni qualvolta termina un processo di apprendimento e quest'ultimo si occupa al termine della sua computazione anche di scrivere tale esito generato all'interno di un file ".csv". Tale file conterrà le performance rilevate in funzione del numero di traiettorie eseguite.

- **selectAction()**: a partire dallo stato corrente seleziona l'azione da applicare. Se si sta eseguendo la fase di training, applica la politica $\epsilon - Greedy$ mentre se si è nella fase di simulazione della dinamica seleziona sempre l'azione che massimizza la funzione approssimante della action value function.

3.2. Funzione lineare approssimante

Al fine di rappresentare all'interno della classe Experience Replay la funzione approssimante lineare della Q-function si è optato per la definizione di una classe denominata ApproximatorQFunction all'interno del file *approximator_QFunction.py*. Quest'ultima riceve in input al costruttore e mantiene come attributi i centri, ed opzionalmente anche i raggi, delle **RBFs gaussiane** ed il learning rate da utilizzare per l'apprendimento. Da notare, che se i raggi delle RBFs non vengono indicati come parametro al costruttore, questi vengono calcolati per ogni direzione come la distanza tra uno step della griglia che ne indica i centri lungo la direzione in questione. Per la rappresentazione dei parametri associati alla funzione approssimante poi, la classe memorizza un dizionario che associa ad ogni azione i relativi valori dei parametri.

La classe implementa i seguenti metodi:

- **fit():** esegue N passi di discesa del gradiente andando ad aggiornare i parametri del dizionario così che la funzione approssimante rappresentata diminuisca l'errore nei confronti della corretta funzione da approssimare. Per farlo riceve in input, oltre all'azione ed al numero

di replays N, la coppia composta da un punto e dal relativo valore che la funzione da approssimare assume in corrispondenza di quest'ultimo.

- **predict():** restituisce il valore della predizione associata ad una data coppia composta da un punto x e da un'azione u.

- **rbf():** ricevuti un centro $c = [c_1, c_2, ..., c_n]$, un raggio $s = [s_1, s_2, ..., s_n]$ ed un punto $x = [x_1, x_2, ..., x_n]$, calcola e restituisce il valore che la j-esima RBF gaussiana centrata in c e con raggio s assume in x. Questo viene calcolato come:
$\widetilde{\phi}_j(x) = exp \sum_{n=1}^{N} -\frac{(x_i - c_i)^2}{s_i^2}$

- **calculateRBFsValue():** utilizzando il metodo rbf(), calcola il valore di ciascuna delle RBFs gaussiane in corrispondenza di un dato punto e restituisce un vettore che li contiene. Nel merito dei valori contenuti in tale vettore occorre specificare come questi siano normalizzati. Sia quindi $\widetilde{\phi}_i(x)$ il valore non normalizzato della i-esima RBF in corrispondenza del punto x, il valore aggiunto al vettore che verrà restituito al chiamante viene calcolato come: $\phi_i(x) = \frac{\widetilde{\phi}_i(x)}{\sum_{n=1}^{N} \phi_n(x)}$

3.3. Memoria

Ogni qualvolta l'algoritmo ER genera un nuovo esempio, questo come visto deve essere memorizzato in un dataset affinchè possa in futuro essere riutilizzato per l'apprendimento. Tale dataset, all'interno del progetto, è memorizzato per mezzo di una classe Memory definita all'interno del file *memory.py*. Quest'ultima mantiene in memoria un buffer, ovvero una lista all'interno della quale sono contenuti i dati relativi agli esempi. Implementa quindi un metodo denominato *appendElement()* che aggiunge i valori correlati ad un esempio al buffer. Il metodo in questione non si limita bensì a ciò, ma va anche a scrivere il nuovo dato in un file con estensione ".csv" che memorizza il dataset degli esempi relativi alla computazione corrente, così che questi possano essere recuperati per un eventuale analisi futura.
La classe *Memory* qua descritta, viene utilizzata anche durante la fase di simulazione della dinamica. Al suo interno difatti vi è anche la definizione di un ulteriore metodo *writeElementTrajectory()*, il quale scrive in un file, il cui percorso viene specificato come parametro al costruttore della classe, lo stato visitato durante la simulazione. Tale procedura consente di poter recuperare in futuro i dati ottenuti durante lo svolgimento della simulazione stessa, così che questi possano essere effettivamente analizzati al termine dell'esecuzione degli esperimenti.

Infine occorre notare come la classe mantenga riferimento anche alla dinamica, la quale implementazione è discussa di seguito. Ciò, è reso necessario dal fatto che la memorizzazione di tale oggetto all'interno della classe che rappresenta la memoria, consente di ottenere informazioni caratterizzanti per il problema, quali ad esempio il numero di componenti di ciascun punto dello spazio degli stati e di quello delle azioni, necessarie per la fase di aggiunta dei dati al buffer.

3.4. Dinamica

Un' ulteriore componente di notevole rilevanza è sicuramente rappresentata da quella adibita alla rappresentazione della dinamica del problema al quale si applica l'algoritmo ER.
Questa è implementata all'interno del progetto per mezzo della definizione di un classe astratta *Dynamic* interna al file *dynamic.py*. Tale interfaccia definisce nello specifico tutti quei metodi e quegli attributi che una dinamica concreta dovrà implementare. Per quanto concerne gli attributi, all'interno vi è definito che ogni classe dinamica concreta dovrà necessariamente definire il proprio spazio delle azioni (un array contenente un insieme discreto di valori) la cui informazione contenutavi risulta necessaria ai fini dell'esecuzione dell'algoritmo, e le label relative allo spazio degli stati ed a quello delle azioni. Queste ultime rappresentano cioè i nomi associati alle componenti di tali due insiemi e si rivelano necessarie durante la fase di scrittura degli esempi nei file relativi al dataset (training) ed alle traiettorie (simulazione).
Per quanto concerne invece i metodi che una dinamica deve necessariamente implementare, questi sono i seguenti:

- **dynamic():** restituisce il valore associato alla dinamica del problema.

- **step_simulate():** a partire da uno stato corrente e da un'azione applicata ricevuti come parametri, restituisce lo stato futuro.

- **reward():** calcola e restituisce il reward, sulla base della coppia composta da stato corrente ed azione applicata definiti tra i parametri.

- **casualState():** restituisce uno stato casuale appartenente allo spazio degli stati.

- **generateRBFGrid():** genera e restituisce al chiamante una matrice che rappresenta i centri delle RBFs da utilizzare per la funzione lineare approssimante della Q-function.

- **plotTrajectory():** rappresenta graficamente una traiettoria. Nello specifico genera un grafico che mostra l'andamento di ciascuna delle componenenti dello stato, delle azioni applicate e dei reward rilevati.

- **getTestStates():** restituisce una lista di stati da utilizzare per il calcolo delle performance.

- **getInitialState():** restituisce lo stato iniziale dal quale far partire una simulazione.

Per la realizzazione del presente elaborato, si sono implementate nel dettaglio due classi concrete relative a due problemi real-time. Il primo di questi, la cui dinamica è definita nella classe *PendulumDynamic* nel file *pendulumDynamic.py*, consiste nella stabilizzazione di un pendolo in una posizione di equilibrio, mentre l'altro, implementato nella classe LinkRoboticDynamic interna al file *linkRoboticDynamic.py* prevede la stabilizzazione di un manipolatore robotico a due collegamenti. Per una descrizione più accurata delle due dinamiche, si rimanda al successivo capitolo della relazione.
All'interno di tali due file citati, vi è la definizione di altre due classi denominate *AnimatedPendulum* e *AnimatedLinkRobotic* che consentono di mostrare l'animazione di una simulazione relativa alla rispettiva dinamica.

3.5. Test

L'elaborato presenta anche un modulo, interno ad un file denominato *test.py*, atto all'esecuzione di una serie di esperimenti necessari per la verifica delle performance rilevate dall'applicazione dell'algoritmo Experience Replay alle due dinamiche considerate. Nel dettaglio, lo script in questione consente di eseguire due fasi di testing.
Innanzitutto, una prima parte atta all'addestramento di un certo numero di modelli ed il successivo confronto delle loro performance rilevate.
Successivamente a ciò, una volta generati i modelli in questione, è possibile provare per ciascuno di essi l'esecuzione di una simulazione, a partire da un certo stato iniziale definito come parametro allo script. Il modulo di codice quindi, al termine della computazione mostra graficamente l'andamento della simulazione migliore sia in termini di rappresentazione grafica dei valori relativi a stati visitati, azioni generate e reward rilevati, sia per mezzo di un'animazione che ne simula a tutti gli effetti l'andamento.
Si specifica inoltre come tutti i parametri necessari per l'esecuzione dei test, nonchè per la computazione dell'algoritmo, siano settati all'interno di un apposito file *app.conf*.

4. Esperimenti e risultati

Al fine di comprendere i benefici apportati dall'algoritmo a dei problemi real-time, si sono presi in considerazione due degli esempi tra quelli analizzati nel paper [1]. Per ciascuno dei modelli in questione, entrambi discussi nel dettaglio di seguito, si è quindi eseguito una serie di esperimenti articolata in due parti:

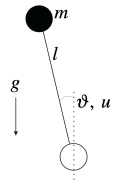

Figure 3. Rappresentazione del pendolo

- Una prima fase atta all'addestramento di un certo numero di modelli di test.

- Una seconda fase durante la quale tali modelli sono stati utilizzati per l'esecuzione di simulazioni a partire da alcuni stati di partenza considerati caratterizzanti, con l'obiettivo di comprendere in quanto tempo l'agente riuscisse a raggiungere l'obiettivo auspicato al variare del modello considerato. Al termine della computazione quindi la migliore delle traiettorie rilevate, intesa come reward totale minore, è stata mostrata in output per mezzo di un grafico che ne mostra gli stati visitati, le azioni compiute ed i reward ottenuti in funzione del tempo.

4.1. Pendolo

Il primo problema considerato consiste nella stabilizzazione di un pendolo in una posizione di equilibrio verticale. Lo stato di un agente del problema in questione, del quale è mostrato uno schema in Figura 3, è un vettore del tipo $x = [\theta, \dot{\theta}]^T$ con $\theta \in [-\pi, \pi)$ rad che ne indica l'angolo ($\theta = 0$ indica pendolo rivolto verso l'alto e $\theta = -\pi$ rivolto verso il basso) e $\dot{\theta} \in [-15\pi, 15\pi]$ la velocità angolare. La dinamica del problema è definita come:

$$\ddot{\theta} = (mgl \sin\theta - b\dot{\theta} - K^2\dot{\theta}/R + Ku/R)/J$$

con $J = 1.91 \cdot 10^{-4} kgm^2$, $m = 0.055kg$, $g = 9.81m/s^2$, $l = 0.042$, $b = 3 \cdot 10^{-6} Nms/rad$, $K = 0.0536Nm/A$, $R = 9.5\Omega$ ed u forza applicata. Come detto, l'obiettivo del pendolo è quello di stabilizzarsi nella posizione di equilibrio verticale, ovvero in uno stato che più si avvicini ad $x_f = [0, 0]^T$. Per misurare la ricompensa ottenuta da un agente quindi, nel momento in cui questo applica una forza u a partire da uno stato corrente x, viene rilevato un reward pari a $r_{k+1} = r(x, u) = -x^T Q_{rew} x - R_{rew} u^2$ con $Q_{rew} = diag[5, 0.1]$ e $R_{rew} = 1$. Il valore restituito dalla funzione reward sarà quindi difatti maggiore all'avvicinarsi dello stato corrente a quello finale (x_f), tenendo anche conto della forza applicata di modo che se quest'ultima risulta essere nulla il reward sarà maggiore. Lo spazio delle azioni è definito come l'insieme discreto $U = \{-3, 0, 3\}$, mentre il tempo di campionamento è pari al valore di $T_s = 0.005s$.
Per quanto concerne l'approssimatore lineare della action

value function, si è optato per l'utilizzo della stessa strategia descritta nell'articolo [1], la quale prevede la definizione di 11^2 RBFs gaussiane, ciascuna delle quali centrata in uno dei punti appartenenti alla griglia di dimensione 11×11 contenente punti equidistanti tra loro all'interno dello spazio delle azioni. I raggi delle RBFs lungo ciascuna dimensione sono scelti come la lunghezza di uno step all'interno della griglia stessa lungo la direzione in questione. Utilizzando tali dati, è stato implementato il modulo di codice *PendulumDynamic* che codifica la dinamica del problema appena descritto (vedi capitolo 3.4).

Come preannunciato, una volta implementata la dinamica, è stata svolta una serie di test atta a verificare se dopo l'applicazione dell'algoritmo Experience Replay, l'agente, utilizzando la funzione lineare approssimante della Q-function prodotta, riuscisse effettivamente a stabilizzare il pendolo nella posizione di equilibrio verticale.
Per far ciò, si sono eseguite 20 computazioni dell'algoritmo ER in modalità di training, ciascuna delle quali ha generato in output un modello utilizzabile per poterne eseguire delle simulazioni.
Ogni modello è stato addestrato per un totale di 70 traiettorie, ciascuna delle quali contenente 300 esempi. Esattamente come suggerito nell'articolo [1], per l'addestramento durante ciascuna computazione è stato utilizzato un learning rate fisso $\alpha = 0.1$, un fattore di sconto $\gamma = 0.98$, un numero di replays $N = 10$ ed una probabilità di esplorazione della strategia $\epsilon - Greedy$ che parte da un valore di 1 per poi diminuire di traiettoria in traiettoria con rate pari a 0.9886.
Durante l'addestramento, come già ampiamente illustrato, tra la generazione di una traiettoria e la successiva, il modello viene addestrato utilizzando l'algoritmo *Q-LEARN-SAMPLES*. Al termine di ogni esecuzione di tale procedura, avviene il calcolo delle performance, il quale consente di ottenere una misura della bontà della funzione approssimante corrente in funzione del numero di traiettorie fino a quel momento visitate durante la fase di training. La misura di tale valore avviene per mezzo del calcolo del reward totale medio rilevato da delle singole simulazioni che utilizzano per la scelta dell'azione da eseguire proprio tale policy corrente. Nello specifico, ciascuna di tali simulazioni utilizzate per il calcolo delle prestazioni ha inizio negli stati appartenenti all'insieme $X_0 = \{-\pi, -\pi/2, 0, \pi/2\} \times \{-10\pi, -5\pi, -2\pi, 0, 2\pi, 5\pi, 10\pi\}$. Per mezzo di tale procedura, ciascuna delle 20 esecuzioni fornirà perciò in output non solo un file ".csv" che codifica i parametri del modello approssimante, ma anche un ulteriore file che associa il valore delle performance al numero di traiettorie fino a quel momento aggiunte al dataset. Al termine dell' addestramento dei 20 modelli quindi, è stato generato il grafico mostrato in Figura 4, il quale mostra l'andamento delle performance in funzione proprio delle traiettorie fino a quel momento visitate dall'algoritmo. Dall'analisi del grafico

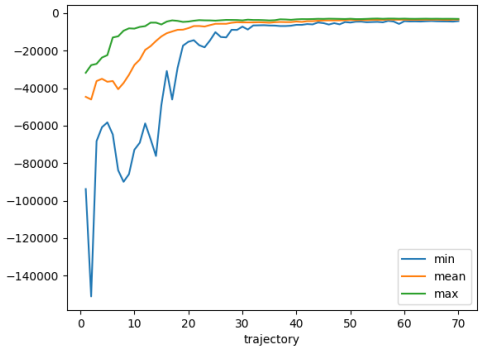

Figure 4. Il grafico mostra il valore minimo, massimo e medio delle performance rilevate durante 20 esecuzioni indipendenti in funzione del numero di traiettorie visitate dall'algoritmo

si può facilmente notare come la crescita delle prestazioni del modello sia estremamente rapida e come questa raggiunga una saturazione intorno alle 40 traiettorie visitate. Da tale numero in poi difatti il reward medio ottenuto durante le simulazioni di prova risulta essere sempre più o meno lo stesso. Da ciò si può quindi comprendere come l'addestramento del modello generato dall'algoritmo Experience Replay richieda l'analisi di circa 40 traiettorie prima di raggiungere la sua migliore approssimazione, e di come questo non possa essere ulteriormente migliorato per mezzo di un maggior numero di cicli di computazione.

I modelli generati durante tale prima fase di testing sono poi stati utilizzati nella successiva parte del lavoro, atta a verificare se questi riuscissero effettivamente a permettere al pendolo di stabilizzarsi nella posizione richiesta per mezzo della action value function approssimante da questi generata durante il training.

Durante la suddetta serie di test quindi, a partire da degli stati iniziali ritenuti caratterizzanti, per ognuno dei 20 modelli si è eseguita una simulazione lunga 800 esempi ($4s$) ed i dati relativi a quella avente il massimo reward totale sono stati mostrati in output. In Figura 5 sono riportati i grafici che mostrano l'andamento dell'angolo θ in funzione del tempo per le migliori simulazioni rilevate aventi origine rispettivamente negli stati $x = [\pi/2, 0]^T$ ed $x = [-\pi, 0]^T$. Analizzando il grafico a sinistra, si nota come il pendolo, partendo da uno stato iniziale avente angolo $\theta = \pi/2$ e velocità angolare nulla, sia in grado di stabilizzarsi nella posizione di equilibrio dopo un tempo solamente di poco superiore a $0.5s$. Risultati senz'altro ottimi, ai quali si aggiungono quelli rilevati per delle simulazioni fatte partire dallo stato iniziale $x = [-\pi/2, 0]^T$, i quali si sono rivelati essere pressochè identici a questi. Passando adesso all'analisi del grafico mostrato sulla destra in Figura 5, si nota come partendo dallo stato $x = [-\pi, 0]^T$, benchè il pendolo impieghi

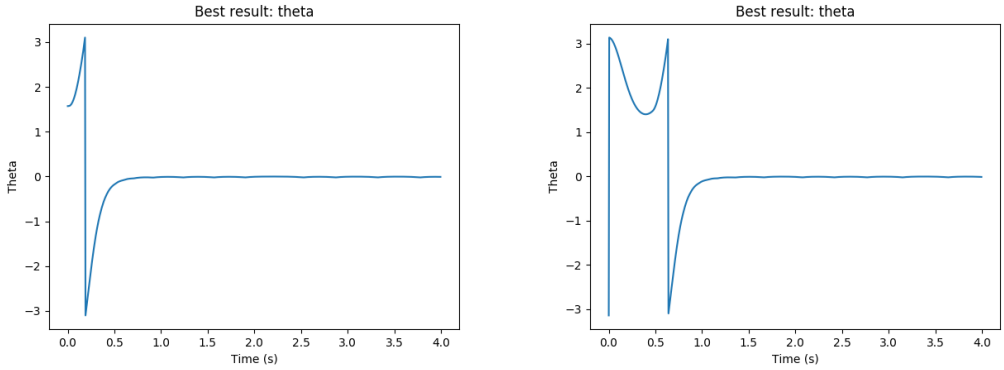

Figure 5. Grafici che mostrano l'andamento dell'angolo θ in funzione del tempo per le migliori simulazioni aventi rispettivamente come stato iniziale $x = [\pi/2, 0]^T$ (a sinistra) ed $x = [-\pi, 0]^T$ (a destra).

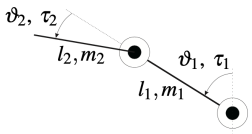

Figure 6. Rappresentazione del manipolatore robotico a due collegamenti

Symbol	Value	Units	Meaning
$l_1; l_2$	0.4; 0.4	m	link lengths
$m_1; m_2$	1.25; 0.8	kg	link masses
$I_1; I_2$	0.066; 0.043	kg m^2	link inertias
$c_1; c_2$	0.2; 0.2	m	center of mass coordinates
$b_1; b_2$	0.08; 0.02	kg/s	damping in the joints

Figure 7. Valori delle costanti considerate per il manipolatore robotico a due collegamenti

più tempo rispetto agli altri due casi nel raggiungere la posizione di equilibrio auspicata, tale tempo impiegato risulta comunque irrisorio in quanto si rivela essere di poco inferiore ad $1s$.

Dall'analisi dei risultati elargiti quindi, risulta evidente quanto l'algoritmo Experience Replay, con una fase di training di 70 traiettorie, sia da considerarsi efficiente se applicato al caso della dinamica relativa alla stabilizzazione di un pendolo in una posizione di equilibrio verticale, e di come in realtà sia sufficiente una fase di addestramento di appena 40 traiettorie per raggiungere le migliori performance.

4.2. Manipolatore robotico a due collegamenti

Il secondo problema considerato, rappresenta fondamentalmente un'estensione su più dimensioni del caso precedentemente analizzato. Nel dettaglio, questo difatti prevede la stabilizzazione nella solita posizione di equilibrio verticale non di un singolo pendolo ma bensì di un manipolatore robotico a due collegamenti, del quale è riportato uno schema in Figura 6. Un agente operante nel merito del problema in analisi, ha uno stato del tipo $x = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]^T$ con $\theta_1, \theta_2 \in [-\pi, \pi)$ angoli e $\dot{\theta}_1, \dot{\theta}_2 \in [-2\pi, 2\pi]$ velocità angolari rispettivamente del primo e del secondo collegamento. Ciascuna azione è composta da un vettore $u = [u_1, u_2]$ in cui u_1 indica la forza applicata ad uno dei due collegamenti ed u_2 quella applicata all'altro. Lo spazio delle azioni è quindi rappresentato in questo caso dall'insieme discreto risultante dal prodotto cartesiano $U = \{-1.5, 0, 1.5\} \times \{-1, 0, 1\}$.
La dinamica è invece espressa come:

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} = \tau$$

con $\tau = [\tau_1, \tau_2]$ vettore che indica l'azione applicata. Le matrici M e C sono così definite:

$$M(\alpha) = \begin{bmatrix} P_1 + P_2 + 2P_3 \cos\alpha_2 & P_2 + P_3 \cos\alpha_2 \\ P_2 + P_3 \cos\alpha_2 & P_2 \end{bmatrix}$$

$$C(\alpha, \dot{\alpha}) = \begin{bmatrix} b_1 - P_3\dot{\alpha}_2 \sin\alpha_2 & -P_3(\dot{\alpha}_1 + \dot{\alpha}_2) \\ P_3\dot{\alpha}_1 \sin\alpha_2 & b_2 \end{bmatrix}$$

con $P_1 = m_1c_1^2 + m_2l_1^2 + I_1$, $P_2 = m_2c_2^2 + I_2$ e $P_3 = m_2l_1c_2$. I valori delle rimanenti costanti sono riportati in Figura 7.
Come detto anche in questo caso l'obiettivo è quello di portare il sistema in una posizione di equilibrio verticale ($\theta = \dot{\theta} = 0$), obiettivo questo che è rappresentato dalla seguente funzione di reward:

$$r_{k+1} = r(x, u) = -x^T Q_{rew} x$$

con $Q_{rew} = diag[1, 0.05, 1, 0.05]$, x stato corrente ed u forza applicata.
Il tempo di campionamento in questo caso si pone pari al

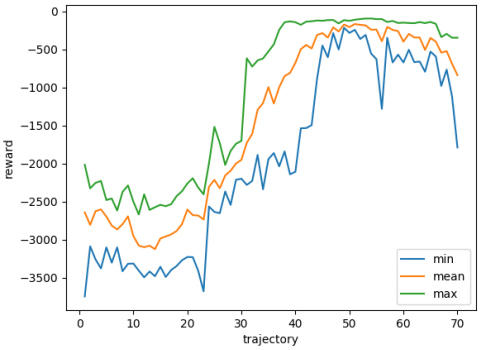

Figure 8. Il grafico mostra il valore minimo, massimo e medio delle performance rilevate durante 5 esecuzioni indipendenti in funzione del numero di traiettorie visitate dall'algoritmo.

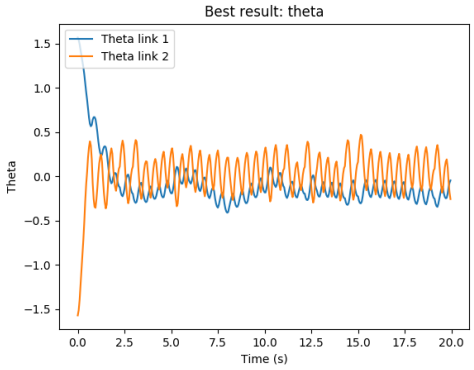

Figure 9. Grafico che mostra l'andamento degli angoli assunti dai due collegamenti robotici nella migliore simulazione tra quelle ricavate utilizzando i 5 modelli addestrati su 70 traiettorie. Le simulazioni sono state eseguite a partire dallo stato iniziale $x = [\pi/2, 0, -\pi/2, 0]$

valore di $T_s = 0.05s$.
Per quanto riguarda i centri delle RBFs gaussiane da utilizzare per l' approssimante della Q-function, la metodologia utilizzata risulta essere la stessa già illustrata per il caso del pendolo, con l'unica variante che lo spazio degli stati viene suddiviso lungo ciascuna direzione in 5 punti equidistanti anzichè 11, con la conseguenza che si avranno 5^4 RBFs gaussiane.
Una volta implementata la dinamica utilizzando i dati fin qua illustrati, si è eseguito una serie di esperimenti articolati su due fasi. Esattamente come per il problema del pendolo, la prima parte di test ha riguardato l'esecuzione di molteplici istanze dell'algoritmo Experience Replay alla dinamica in analisi. In questo caso, a causa dell'elevato costo computazionale richiesto dall'addestramento dei modelli per un simile problema definito su più dimensioni, le esecuzioni si sono limitate a 5, in numero quindi inferiore alle 20 previste dal caso precedentemente analizzato. Anche in questa circostanza, i modelli sono stati tutti addestrati per un totale di 70 traiettorie, ciascuna delle quali contenente però 100 esempi anzichè 300. Identici al caso precedente risultano essere anche i valori relativi al numero di replays $N = 10$, al fattore di sconto $\gamma = 0.98$, ed alla probabilità di esplorazione, la quale è inizialmente posta ad 1 per poi diminuire con un rate pari a 0.9886. Il learning rate è stato invece fissato al valore $\alpha = 0.3$. Per il calcolo delle performance si sono eseguiti test su simulazioni aventi stati iniziali appartenenti all' insieme $X_0 = \{-\pi, -2\pi/3, -\pi/3, 0, \pi/3, 2\pi/3\} \times \{0\} \times \{-\pi, -2\pi/3, -\pi/3, 0, \pi/3, 2\pi/3\} \times \{0\}$.

Una volta generati i 5 modelli, esattamente come visto per il caso del pendolo, le performance rilevate durante le varie fasi di training sono state analizzate per mezzo della generazione di un grafico, riportato in Figura 8. Dall' analisi dello stesso, si può notare che rispetto al più banale problema della stabilizzazione di un pendolo, il quale è definito su un minor numero di dimensioni dello spazio degli stati e delle azioni, in questo caso la crescita delle performance rispetto al numero di traiettorie visitate durante l'addestramento presenta molte più valli e molti più picchi. Ciò, può essere in parte dovuto al fatto che i dati si basano su meno esperimenti rispetto al caso precedente, ma il motivo principale risiede ovviamente nel fatto che in questo caso il modello deve poter visitare un maggior numero di stati prima di riuscire a predire la Q-function con una precisione che gli consenta di non dipendere dagli stati visitati durante le traiettorie, i quali sono scelti casualmente secondo la politica $\epsilon - Greedy$. Nel caso del pendolo inoltre, si era notato che dopo poco più di 30 traiettorie visitate, le prestazioni miglioravano significativamente per poi raggiungere una saturazione intorno alle 40 traiettorie. Nell'analisi corrente invece, si evince una crescita delle performance intorno alle 45 traiettorie visitate, a seguito della quale però la saturazione appare qua meno evidente, in quanto si continua a denotare anche per un elevato numero di esempi acquisiti la presenza di numerose valli.

Come visto per i test applicati alla dinamica del pendolo, anche in questo caso i 5 modelli generati sono stati utilizzati per la simulazione di altrettante traiettorie, in questa circostanza lunghe 1000 esempi ($20s$) ciascuna.
Gli esiti dei risultati riportati in Figura 9, mostrano la migliore delle simulazioni rilevate in termini di reward totale ottenuto tra quelle aventi origine nello stato iniziale $x = [\pi/2, 0, -\pi/2, 0]$. Dall'analisi dello stesso, si nota come l'addestramento del modello con 70 traiettorie non sia sufficiente ad ottenere una stabilizzazione accettabile dei due collegamenti nella posizione di equilibrio richiesta. In tal caso infatti entrambi i collegamenti tendono a po-

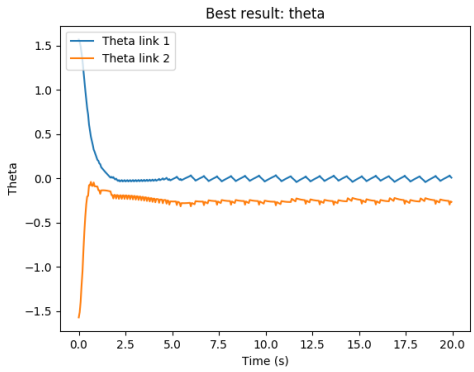

Figure 10. Grafico che mostra l'andamento degli angoli assunti dai due collegamenti robotici nella simulazione ricavata utilizzando un modello addestrato su 100 traiettorie. Le simulazioni sono state eseguite a partire dallo stato iniziale $x = [\pi/2, 0, -\pi/2, 0]$

sizionarsi dopo appena $2.5s$ nella posizione richiesta ma vi oscillano considerevolemente per tutta la durata della simulazione.
Per verificare se fosse possibile migliorare i risultati eseguendo la fase di training dell'algoritmo su più di 70 gruppi di esempi, si è ricavato un modello addestrato su 100 traiettorie e lo si è utilizzato per l'esecuzione di un'ulteriore simulazione. I risultati, raffigurati in Figura 10, mostrano come in questo caso l'oscillazione dei due collegamenti attorno alla posizione di equilibrio si sia ridotta considerevolmente rispetto a quella prodotta con i modelli addestrati per 70 traiettorie.

Dall'analisi dei suddetti risultati, si evince quindi come per casi su più dimensioni, l'algoritmo Experience Replay, tenda a produrre comunque degli ottimi modelli a discapito di una più costosa fase di training richiesta.

5. Conclusioni e sviluppi futuri

L'elaborato discusso nel merito della presente relazione, consiste in un'implementazione in linguaggio di programmazione Python dell'algoritmo Experience Replay applicato ai due problemi real-time relativi alla stabilizzazione in una posizione di equilibrio di un pendolo e di un manipolatore robotico a due collegamenti. L'architettura presentata, come spiegato nel corso del capitolo 3.4, può essere facilmente estesa con l'implementazione di ulteriori dinamiche che vadano ad applicare l'algoritmo in questione ad altri contesti di riferimento.

L'analisi dei risultati prodotti durante la fase di testing dell'elaborato in oggetto, ha denotato delle ottime performance in relazione all'applicazione dell'algoritmo Experience Replay ai problemi considerati. La sua applicazione per una breve fase di training, ha difatti permesso, per il primo problema, al pendolo di stabilizzarsi nella posizione di equilibrio richiesta in un tempo irrisorio. Lo stesso vale per il manipolatore robotico, il quale ha però richiesto un più costoso processo di addestramento atto a ridurre le oscillazioni dei due collegamenti intorno alla posizione di equilibrio.
Da ciò si evince quindi, in conclusione, come l'algoritmo Experience Replay consenta di risolvere problemi di controllo real-time di piccole dimensioni con un tempo computazionale non troppo elevato, e come questo sia inoltre applicabile con ottimi risultati anche per casi più complessi a discapito di una maggiore fase di training richiesta.

References

[1] S. Adam, L. Busoniu, and R. Babuska. Experience replay for real-time reinforcement learning control, 2011.