Corso di Computational Models For Complex Systems

README Progetto

Anno Accademico 2018/2019

Michele Fontana Matricola: 476091

14 febbraio 2020

1 Introduzione

Lo scopo del progetto consiste nella realizzazione e nella comparazione di due simulatori basati su agenti. In particolare uno di essi include al suo interno un modello di rete neurale per effettuare predizioni sull'azione che deve compiere ogni singolo agente.
In questo breve README si presenta brevemente la struttura del progetto e si riportano le istruzioni per eseguire il codice vero e proprio.
Si precisa che l'intero progetto, risultati compresi, può essere scaricato clonando la seguente repository di Github:

http://github.com/mikelefonty/CMCS

2 Struttura Progetto

Il progetto è organizzato secondo la seguente struttura:

- **Agent**: La cartella contiene l'implementazione degli agenti presenti nella simulazione.
  - **agent.py**: Contiene l'implementazione di un generico agente "non intelligente".
  - **smart_agent.py**: Contiene l'implementazione di un agente smart. Utilizza cioè una rete neurale per predire la distribuzione di probabilità dell'azione da compiere.

- **Clustering**: La cartella racchiude i moduli il cui scopo è eseguire un algoritmo di clustering, al fine di calcolare ad ogni iterazione il numero di clusters di agenti. Questa misura è la metrica utilizzata per comparare tra loro i simulatori.
  - **dbscan.py**: Al suo interno è implementata una versione semplificata di DBSCAN, un algoritmo di clustering basato su densità.
- **CONSTANTS**: Questa cartella contiene il file json avente al suo interno le costanti utilizzate nel corso dell'intero progetto.
  - **costanti.json**: File json contenente le costanti.
  - **constant_reader.py**: Contiene l'implementazione della classe *ConstantReader* che legge il file json e restituisce attraverso dei getter il valore di ogni singola costante.
- **Decide_Direction.py**: Contiene le funzioni utilizzate per calcolare, dato lo stato dell'ambiente e l'ampiezza del vicinato, la distribuzione delle azioni possibili da parte di un agente.
  - **decide_direction.py**: Al suo interno è presente la funzione next_direction, tramite cui viene calcolata la distribuzione in questione. Questa corrisponde alla funzione che il modello di rete neurale deve imparare.
- **Environment**: Contiene il modulo che racchiude l'implementazione del dato astratto, il cui scopo è modellare un generico ambiente.
  - **environment.py**: Al suo interno è presente la classe *Enviroment* che modella un generico ambiente. Esso è rappresentato tramite due strutture dati diverse:
    1. Dizionario env_dict: Ogni entry è della forma $agent\_id : (x, y)$. Essa corrisponde al fatto che l'agente con identificatore $agent\_id$ si trova nella posizione (x, y)
    2. Matrice $m \times n$ env_matrix. La generica cella (i, j) ha valore 0 se è vuota, altrimenti contiene il valore dell'identificatore dell'agente in essa presente. Questa matrice viene utilizzata dagli agenti per decidere l'azione.
- **EnvSet**: Contiene i file di testo che rappresentano gli ambienti a disposizione.
- **Logger**: Contiene il modulo che effettua il log della simulazione su file.
  - **logger.py**: Il modulo salva ad ogni iterazione i risultati della simulazione. Durante il corso della simulazione, i risultati sono salvati in un DataFrame della libreria pandas. Al termine della simulazione il contenuto del DataFrame è salvato in un file json.
- **Main**: Contiene i vari main utilizzati per avviare la simulazione e/o effettuare vari esperimenti.

  - **Compare_Block_Std.py**: Utilizzato per comparare tra loro il simulatore a Blocchi ed il simulatore Standard. Tali simulatori verranno presentati successivamente.
  - **Compare_Sim.py**: Utilizzato per confrontare la versione intelligente con quella "normale" del simulatore Standard.
  - **Create_Env.py** Il main crea un nuovo ambiente e lo salva su file.
  - **Main_Block_Sim.py**: Avvia il simulatore a blocchi.
  - **Main_Standard_Sim.py**: Avvia il simulatore standard.
  - **plot.py**: Usato solamente per effettuare il plot delle misure dei tempi.
  - **Replay_Sim.py**: Mostra l'animazione tramite scatter plot della simulazione designata, a partire dal contenuto dei file di log.

- **Modelli**: Contiene al suo interno le reti neurali necessarie agli agenti smart per effettuare le predizioni sulle azioni da eseguire.

- **Results**: Contiene i risultati delle varie simulazioni.

- **Simulator**: Contiene l'implementazione dei vari simulatori.

  - **block_simulator.py**: contiene l'implementazione del simulatore a blocchi.
  - **standard_simulator.py**: contiene l'implementazione del simulatore standard.

- **Simulator_Output**: Contiene il modulo in grado di mostrare un'animazione della simulazione corrente.

  - **simulator_debug.py**: Tramite uno scatter plot mostra lo sviluppo completo della simulazione corrente. In particolare mostra,ad ogni iterazione, lo stato dell'ambiente ed il valore della metrica di validazione.

- **Util**: Contiene funzioni di utilità generale.

  - **data_structures.py**: Contiene le strutture dati utilizzate nel progetto.
  - **environment_IO.py**: Contiene le funzioni necessarie a creare, salvare e caricare un ambiente da e su file.
  - **matrix_functions.py**: Contiene le funzioni che operano su matrici e sottomatrici. In particolare si trova la funzione *extract_neighborhood* che estrae il vicinato di un generico agente.
  - **print_utils.py**: Contiene la funzione che stampa a schermo in modo maggiormente comprensibile il vettore della distribuzione di probabilità delle azioni.
  - **simulation_utils.py**: Contiene varie funzioni di utilità generale, tra cui la funzione che permette di replicare una generica simulazione.

3 Simulatori Implementati

Nel corso del progetto sono stati implementati principalmente due diverse tipologie di simulatori, che chiameremo *standard* e *simulatore a blocchi*. Essi differiscono esclusivamente per il processo di aggiornamento dello stato dell'ambiente nel corso di una generica iterazione.
Va precisato che all'inizio dell'iterazione i entrambi i simulatori stabiliscono un ordine, cioè una lista di identificatori, secondo il quale aggiornare gli agenti.
Partiamo ad esaminare il simulatore standard.
Consideriamo due agenti A_1 e A_2 e supponiamo che A_1 preceda A_2 nell'ordinamento. Il simulatore standard pertanto dapprima fa eseguire la propria azione ad A_1 ed aggiorna lo stato dell'ambiente in modo coerente. Solo dopo avere aggiornato l'ambiente permette all'agente A_2 di poter scegliere la direzione da seguire. Tale azione sarà decisa basandosi sull'ambiente appena aggiornato.
Risulta facile intuire che l'iterazione i non risulta essere parallelizzabile, poiché intercorrono dipendenze a livello di dati tra le attività di aggiornamento degli agenti.
Questa osservazione risulta essere cruciale poiché risulta essere la causa principale dell'inefficienza della versione intelligente del simulatore. Come si vede dalla Figura 1, all'aumentare del numero di agenti il tempo impiegato dal simulatore smart aumenta sempre di più.

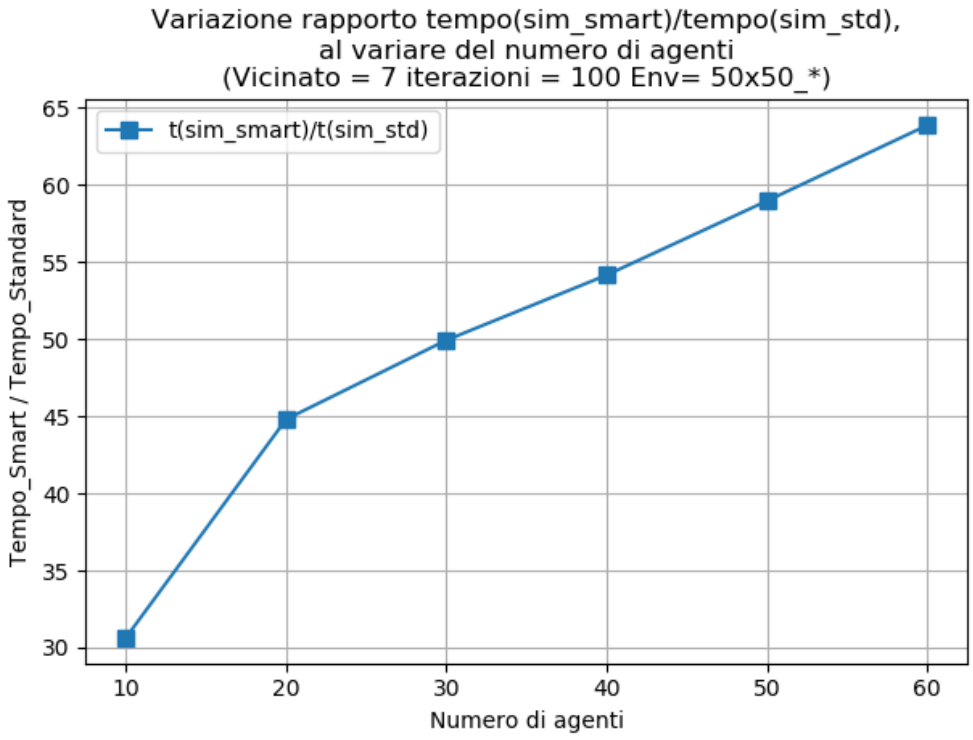


Figura 1: La figura mostra l'incremento del rapporto tra tempo di completamento della versione smart e a quella *non-smart*, al variare del numero di agenti.

Le librerie di Machine Learning sono infatti progettate ed ottimizzate per lavorare su insiemi di matrici contemporaneamente; come spiegato in precedenza in questo caso non è possibile eseguire tale attività a causa delle dipendenze presenti nel processo di simulazione.
Per verificare quest'ultima affermazione è stato sviluppato una seconda tipologia di simulatore: il simulatore *a blocchi*.
Presa una generica iterazione i e il rispettivo ordinamento degli agenti, il simulatore suddivide tale ordinamento in un numero di blocchi pari a n, con n parametro scelto dall'utente. Ogni blocco b viene aggiornato contemporaneamente, andando chiaramente a rilassare le ipotesi del problema. Solo dopo che gli agenti del blocco b hanno deciso le rispettive azioni, l'ambiente può essere aggiornato.
L'utilizzo di questo simulatore può portare in linea di principio ad avere due o più agenti presenti all'interno della stessa cella dell'ambiente.
In Figura 2 si mostra come varia il tempo impiegato dalla versione smart del simulatore a blocchi al variare del numero di blocchi. Come si vede, all'aumentare del numero di blocchi, aumenta anche la il tempo di completamento della simulazione.

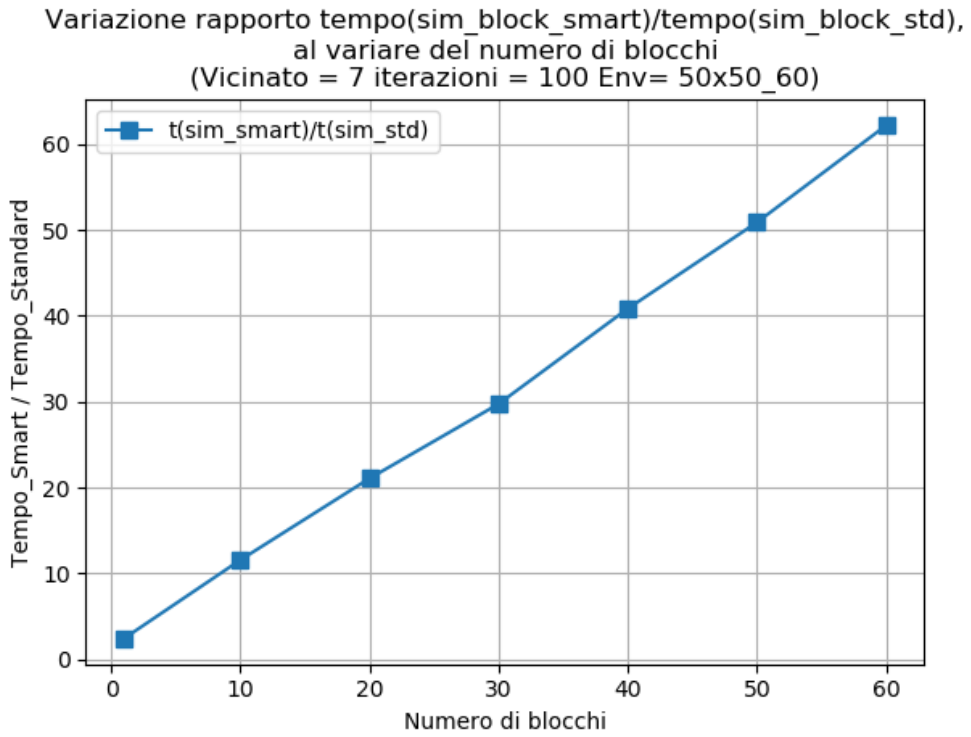
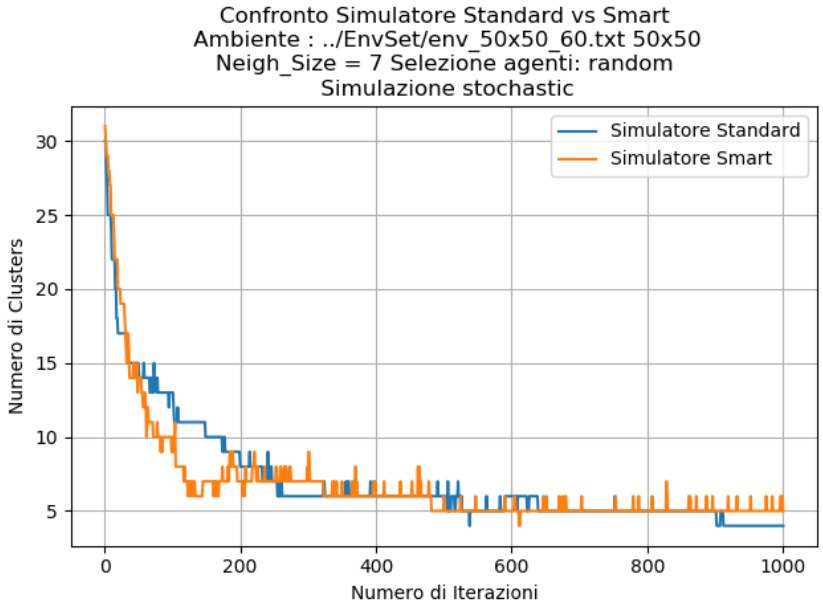


Figura 2: La figura mostra l'incremento del rapporto tra tempo di completamento della versione smart e a quella *non-smart*, al variare del numero di blocchi.

Concludiamo la sezione presentando i risultati del confronto tra la versione smart e quella "normale" del simulatore standard. Come si vede in Figura 3, la rete neurale riesce ad approssimare in modo soddisfacente la funzione che determina la distribuzione di probabilità delle azioni da compiere per i vari agenti. Di conseguenza durante la simulazione

è possibile osservare che il comportamento degli agenti smart e non intelligenti risulta essere simile.



(a)

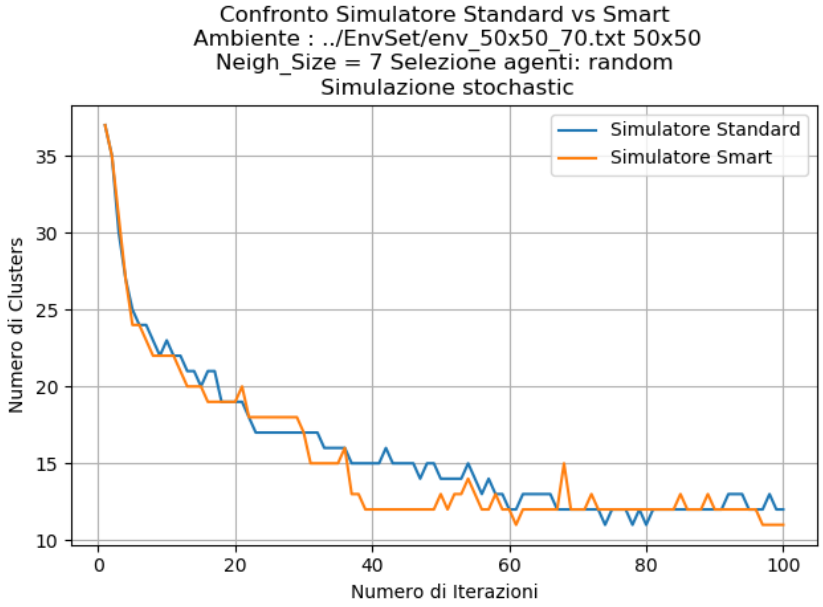


Figura 3: Confronto tra versione smart e normale del simulatore standard.

4 Istruzioni per l'esecuzione

Iniziamo questa sezione illustrando i parametri attesi dai vari Main.
I parametri posizionali sono:

- **env** : Path del file dell'ambiente, relativo alla cartella EnvSet.
- **n_iters**: Numero di iterazioni da svolgere.

I seguenti parametri sono opzionali.

- **-k** : Raggio del vicinato da considerate per un generico agente. Il valore deve essere compreso in $[2, 11]$ se si vuole utilizzare la versione smart del simulatore.
- –**rand_strat**: Se inserita, l'ordine di aggiornamento degli agenti ad ogni iterazione è casuale. Altrimenti, gli agenti sono aggiornati secondo l'ordine numerico dei loro identificatori.
- –**stoc**: Se settata, l'azione da effettuare per ogni agente è scelta in modo casuale, in accordo alla distribuzione calcolata. Altrimenti, viene scelta l'azione avente maggiore probabilità.
- –**smart**: Viene utilizzata la versione smart del simulatore, creando quindi agenti aventi al loro interno una rete neurale.
- –**seed SEED**: Nella generazione di numeri pseudocasuali, viene utilizzato il valore SEED come seme.
- –**show_anim**: Se settata, mostra l'animazione della simulazione.
- –**replay**: Se settata, viene ricreata la simulazione a partire dai file di log e mostrata la rispettiva animazione.
- –**path PATH**: Path della cartella che conterrà i risultati della simulazione. Il path è relativo alla cartella Results.
- –**res_name RES_NAME**: Prefisso del nome dei file in cui saranno salvati i risultati della simulazione.
- –**verbose VERBOSE**: Setta il livello con cui vengono mostrate informazioni sul terminale. Il valore di VERBOSE deve appartenere a $\{0, 1, 2\}$.
- –**blocks BLOCKS**: Da usare esclusivamente per il Block Simulator. Gli agenti vengono divisi in BLOCKS blocchi.

Esempi di lanci dei vari main è possibile trovarli nel file *esempio_lancio_main.txt*