



Cart pole: implementazione di algoritmi per la gestione del pendolo inverso

Documentazione progettuale

Progetto del corso di Intelligenza Artificiale
Università degli Studi di Bergamo
A.A. 2018/2019

PIFFARI MICHELE - 1040658

October 8, 2019

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. (Sutton and Barto)

Contents

1	Introduzione	1
1.1	Cart Pole	1
1.2	RL e controllo	2
1.3	Cart Pole environments and reward	4
2	Implementazione	7
2.1	Algoritmo Q learning	7
2.1.1	Perchè Q-learning?	7
2.1.2	Q-learning off policy control	8
2.1.3	Scelta dell'azione e Q table	8
2.1.4	Result	10
2.2	Algoritmo DQN	11
2.2.1	Q network	11
2.2.2	Replay memory	12
2.2.3	Result	13
2.3	Algoritmo <i>Finite Differences</i>	14
2.3.1	Approccio <i>Policy gradient estimation</i>	15
2.3.2	Algoritmo <i>FD</i>	16
2.3.3	Result	18

List of Figures

1.1	Cart pole e variabili di stato	1
1.2	Concetto di input-output	2
1.3	Rete di retroazione	2
1.4	Modellizzazione <i>black - box</i>	2
1.5	Sequenza <i>State-Action-Reward</i>	3
1.6	Policy	3
1.7	Significato di <i>imparare</i> per un agente	3
1.8	Policy update con algoritmi di <i>RL</i>	4
1.9	RL inteso come <i>control theory</i>	4
2.1	4 value functions utilizzate per la policy update	8
2.2	Equazione di <i>Bellman</i> utilizzata per andare ad aggiornare i valori della funzione Q . .	8
2.3	Aggiornamento Q table	8
2.4	Metodo per la stima della azione da eseguire	9
2.5	Possibile struttura della Q-table	10
2.6	Andamento del Q learning	10
2.7	Andamento del Q learning confrontabile con altri algoritmi implementati	11
2.8	DQN network	11
2.9	Modello della rete neurale: 4 input <i>state</i> e 2 output <i>Q-action</i>	12
2.10	Creazione variabile per sequential memory	12
2.11	Reward con DQN network	13
2.12	Valutazione della funzione sulla base della Q-values	14
2.13	Valutazione della funzione <i>non</i> basandosi sulla Q-values	14
2.14	Matrix form	16
2.15	Reward <i>Finite Differences in Python</i>	16
2.16	Rollout in Python	17
2.17	Centri di ogni neurone della <i>RBF</i>	17
2.18	Reward <i>Finite Differences</i>	18

1

Introduzione

1.1 Cart Pole

Un problema molto conosciuto e diffuso nella letteratura relativa al reinforcement learning (RL) è quello del bilanciamento del *Cart Pole*.

Esso non è altro che una semplice asta collegata ad un carrello tramite un joint non attuato : questo significa che l'asta risulta essere libera di muoversi, per il semplice fatto che non vi è applicata alcuna azione esterna, proprio come un pendolo inverso.

Come si può ben capire dall'immagine in figura 1.1, il Cart Pole rappresenta un sistema instabile: infatti, senza un aiuto esterno, il pendolo cadrebbe verso la posizione verticale, spostandosi nell'altro equilibrio stabile (ovvero quello ovvio).

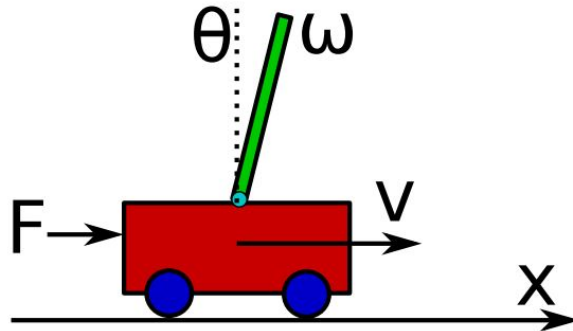


Figure 1.1: Cart pole e variabili di stato

La risoluzione di questo problema di equilibrio verte sull'ambito della *dinamica* e della *teoria del controllo* ed è spesso utilizzato come *problema dummy* per verificare e testare alcune strategie.

Si capisce quindi che, per mantenere il pendolo inverso nella posizione di equilibrio instabile, risulta essere necessario andare ad applicare una coppia sul punto di giunzione tra l'asta e il carrello, andando a muovere orizzontalmente a destra e a sinistra il carrello stesso.

1.2 RL e controllo

Questo problema trova quindi una forte applicazione, come già sottolineato, nell'ambito della teoria del controllo: ma quindi, come possiamo applicare tecniche di reinforcement learning per andare a completare questo compito, ovvero quello di cercare di mantenere il pendolo in posizione verticale?

Come ben sappiamo, e come è evidente nel contesto del Cart Pole (figura 1.2), si tratta di andare a determinare il corretto input (*azione*) al sistema il quale genererà il comportamento desiderato.

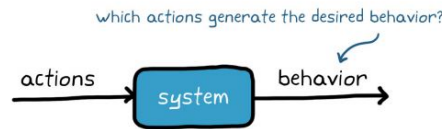


Figure 1.2: Concetto di input-output

La scelta dell'azione da portare all'ingresso del sistema, ovvero di quale movimento il carrello deve effettuare, solitamente viene realizzata andando ad utilizzare un sistema di retroazione ad anello chiuso, per cercare così di migliorare le prestazioni, riducendo l'errore (figura 1.3)

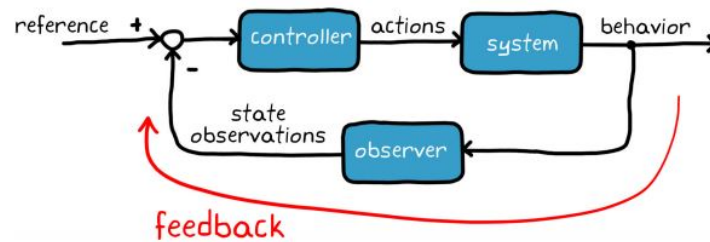


Figure 1.3: Rete di retroazione

Questo concetto è semplice da esprimere a parole e a livello grafico, però, nelle applicazioni reali, può diventare estremamente difficile da ottenere, nel momento in cui il sistema è molto complesso da modellizzare matematicamente, oppure presenta forti componenti non lineari, oppure ancora presenta un elevato spazio degli stati.

Ci viene quindi in aiuto, nel contesto del controllo, il reinforcement learning, il quale permette di andare a condensare in un'unica *black-box* tutto il sistema di controllo, andando a ricevere come ingresso l'insieme di tutte le osservazioni dell'ambiente esterno e fornendo come output le azioni dirette (figura 1.4)

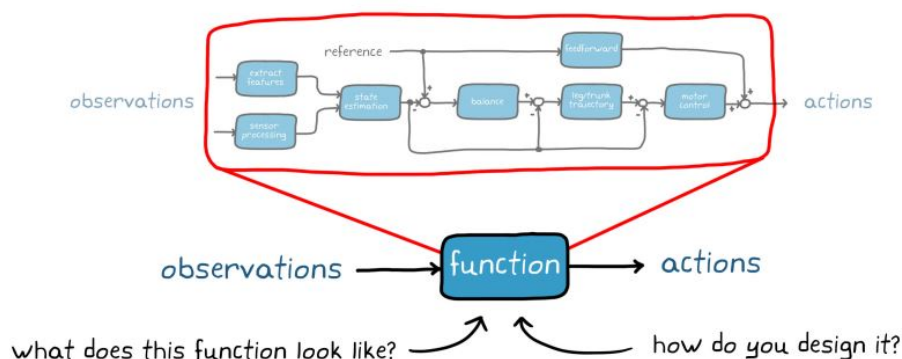


Figure 1.4: Modellizzazione *black - box*

La peculiarità dei sistemi di reinforcement learning, a differenza delle altre due branchie del *machine learning* (*supervised learning* e *unsupervised learning*), sta nel fatto che essi vanno a lavorare con dati provenienti da ambienti dinamici, imponendosi l'obiettivo di andare a trovare la miglior sequenza di azioni che andrà a generare il miglior *output*, ovvero, vale a dire il più alto reward ottenibile dall'agente stesso. Nel fare questo l'agente risulta essere in grado di osservare lo stato attuale dell'*environment*, decidendo poi quali azioni compiere: ovviamente, andando ad eseguire certe azioni, lo stato dell'ambiente cambia, fornendo un certo *reward* all'agente stesso, in base al quale esso può valutare se l'azione eseguita era "buona" oppure se è meglio evitare di ripeterla (questo ciclo è sinteticamente rappresentato in figura 1.5).

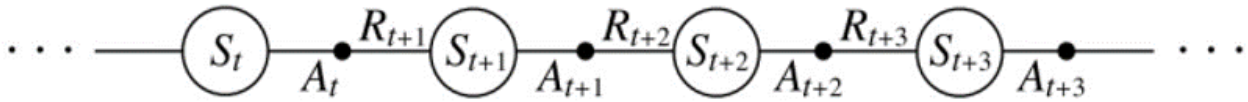


Figure 1.5: Sequenza State-Action-Reward

In particolare, la scelta dell'agente di quale azione eseguire, in seguito a quello che è lo stato osservato, avviene per mezzo di una funzione la quale, nella nomenclatura del RL è chiamata *policy* (figura 1.6).

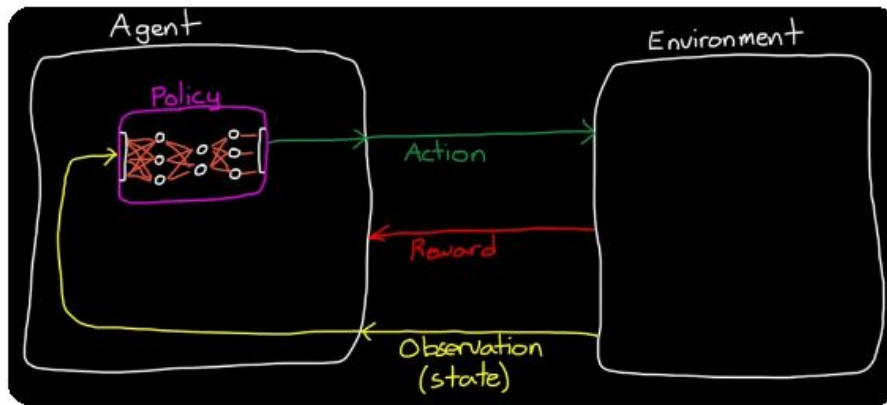
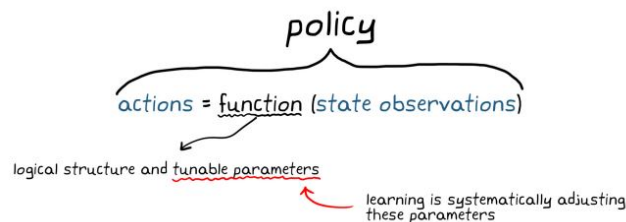


Figure 1.6: Policy

Ovviamente questa mappatura *observation (state)* - *action* non può essere realizzata in maniera statica, anche se trovassimo la miglior policy: questo perchè l'ambiente potrebbe cambiare (e quasi sicuramente lo farà!) nel corso del tempo, e quindi una mappatura statica non sarebbe del tutto ottimale.

Questo quindi porta alla necessità di introdurre i *RL algorithm* i quali permettono di andare ad aggiornare la policy, in base alla stato-azione-reward, cercando quindi di scegliere la policy ottimale per ogni stato dell'ambiente., ovvero cercando di settare nella maniera migliore i parametri che caratterizzano la funzione espressa, in maniera generica, in figura 1.7

Figure 1.7: Significato di *imparare* per un agente

In sostanza questo update della policy può avvenire seguendo differenti algoritmi, come vedremo successivamente: in ogni caso, indipendentemente dalla strategia che si decide di seguire per scegliere l'azione successiva, possiamo modellizzare questo update come un blocco interno all'agente e che lavora direttamente sulla policy, basandosi sulla condizione attuale dell'ambiente, sull'azione che si va ad eseguire e sul reward che si ottiene eseguendola (il tutto è schematizzato in figura ??)

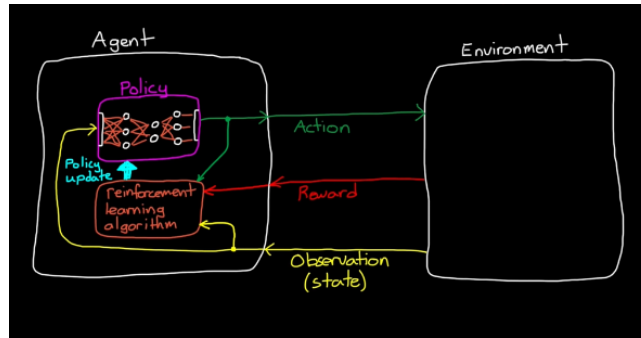


Figure 1.8: Policy update con algoritmi di *RL*

E' a questo punto che possiamo quindi esplicitare il diretto legame tra *RL* e control theory: come abbiamo visto con entrambi i metodi vogliamo andare a determinare la corretta azione da eseguire sul sistema, per ottenere il comportamento desiderato, da cui si ricava il feedback, che corrisponde alle osservazioni dello stato del sistema (figura 1.9).

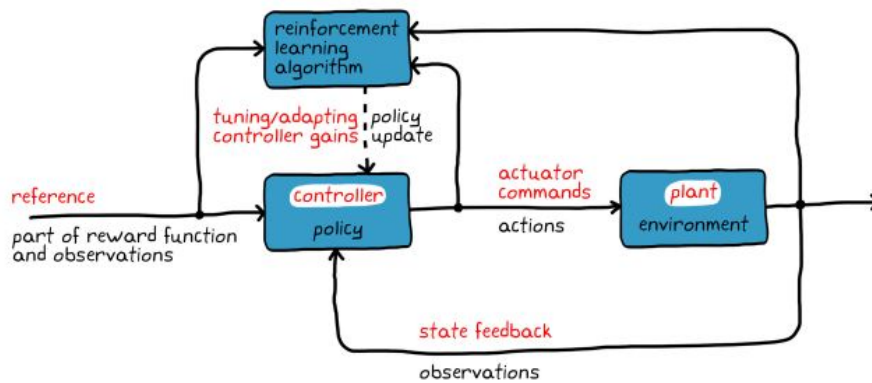


Figure 1.9: *RL* inteso come *control theory*

1.3 Cart Pole environments and reward

Come già esplicitato in precedenza l'obiettivo del problema legato al *cart-pole* è quello di andare a bilanciare l'asta applicando una forza orizzontale al carrello.

L'aspetto principale da considerare è quello relativo allo spazio degli stati: esso rappresenta cosa l'agente va ad osservare del sistema, e quindi quali variabili andrà ad utilizzare per controllare il sistema stesso. Nel nostro caso, e nel caso generico di studio del cart pole, le variabili di stato utilizzate sono:

- Posizione del carrello (x);
- Velocità del carrello (v);
- Angolo di inclinazione della barra (θ);
- Velocità angolare della barra (ω);

L'asta è considerata bilanciata in maniera corretta se entrambi le seguenti condizioni sono rispettate:

- l'angolo dell'asta rimane compreso in un range attorno alla posizione verticale (la quale corrisponde a 0 rad);
- la posizione del carrello rimane all'interno di uno specifico intervallo.

Se una di queste due condizioni risulta essere invalidata, l'episodio specifico termina, facendo ripartire il training. Un'altra possibile via che conduce l'episodio verso la sua terminazione è quello in cui il reward dell'episodio in esame risulta essere superiore ad una certa soglia stabilita in maniera statica, il che corrisponde a verificare che il carrello sia in grado di bilanciare l'asta per più di un certo numero di azioni (ovvero che ha effettivamente imparato a risolvere il problema che gli è stato sottoposto).

Possiamo individuare inoltre due possibili varianti dell'ambiente legato al problema del cart-pole, le quali differiscono per lo spazio degli stati dell'agente:

- **Discreto:** l'agente può applicare al carrello una forza che può valere F_{max} oppure $-F_{max}$, che è rappresentata dalla proprietà *MaxForce* all'interno del codice. In sostanza ciò consiste nell'avere solo due azioni, le quali all'interno del codice sono identificate con due differenti numerazioni. Per esempio, la classica scelta, può essere quella di andare ad assegnare
 - **0** per l'azione di spinta del carrello verso sinistra;
 - **1** invece se l'agente vuole muoversi verso destra.
- **Continuo:** l'agente può applicare una forza che varia in maniera continua all'interno dell'intervallo $[-F_{max}, F_{max}]$.

Un altro aspetto importante da definire è la tipologia di reward che l'ambiente ritorna all'agente: si possono operare diverse scelte. Un'opzione comunemente seguita, la quale sarà sfruttata anche all'interno di questo progetto, è quello di andare a **incrementare il reward di una unità** ad ogni iterazione all'interno di un episodio, fintantochè esso non termina.

Quindi, più l'episodio si protrae nel tempo, più l'agente mantiene in equilibrio l'asta e, di conseguenza, più reward otterrà.

2

Implementazione

2.1 Algoritmo Q learning

2.1.1 Perché Q-learning?

E' importante sottolineare come, nella letteratura relativa all'ambito del reinforcement learning, si è soliti andare a lavorare con quattro funzioni, che permettono di specificare come l'agente cambia la propria policy in funzione dell'esperienza acquisita, come si vede nella figura 1.8. Ovviamente, l'obiettivo dell'agente è quello di ottenere il reward massimo fintantoche continua ad eseguire l'azione per cui è stato concepito: potremmo quindi introdurre il concetto di *discounted return* al tempo t definibile come:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 \bullet R_{t+3} + \dots$$

dove con il coefficiente γ andiamo a definire la *lungimiranza* dell'agente: infatti se $\gamma = 0$ l'agente risulta essere *greedy*, cercando di massimizzare il reward immediato; invece, mantenendo $\gamma < 1$, l'agente risulta essere in grado di dare un peso più o meno maggiore ai reward successivi, garantendo comunque la convergenza della sommatoria 2.1.1.

Questa funzione appena introdotta risulta essere necessaria per definire due funzioni molto importanti:

- **Value of a state, given a policy:** $v_\pi(s)$ che rappresenta il reward totale che ci si aspetta se l'agente segue la policy π dall'istante t fino a ∞ ;
- **Value of a state-action pair, given a policy:** $q_\pi(s, a)$ indica sempre il reward che cumulerò seguendo sempre la stessa policy π , dopo però aver eseguito, nell'istante t una specifica azione a .

I valori ottimali di queste funzioni, per completezza, sono rappresentati con l'apice *, mentre i valori stimati, che sono quelli con cui poi si va effettivamente a lavorare, sono indicati tramite le lettere maiuscole, come si può notare nella figura 2.1.

Si può quindi facilmente intuire da dove deriva la nominazione di questo algoritmo: esso basa il proprio funzionamento sulla funzione Q , la quale rappresenta la funzione cosiddetta *quality*, ovvero che indica quanto è utile una certa azione per aumentare il reward che si andrà ad ottenere nel futuro: si tratta inoltre di un algoritmo *off-policy* poichè la funzione di q-learning apprende da azioni che stanno all'infuori delle policy corrente. Essa può essere inserita in un quadro più generale di 4 funzioni, utilizzabili appunto dell'algoritmo di reinforcement learning per andare a gestire quella che è la policy dell'agente: esse sono riportate in figura 2.1, con i rispettivi calcoli *asintotici* necessari per ricavarne il valore.

Come evidenziato in precedenza, non possiamo però lavorare con i valori ottenibili tramite operatori di valore atteso, ma bensì risulta essere necessario andare ad utilizzare gli elementi stimati, ovvero rispettivamente $V_t(s)$ e $Q_t(s, a)$.

	state values	action values
prediction	v_π	q_π
Optimal (for control)	v_*	q_*

$$v_\pi(s) = \mathbb{E}\{G_t \mid S_t = s, A_{t:\infty} \sim \pi\} \quad v_\pi : \mathcal{S} \rightarrow \mathbb{R}$$

$$q_\pi(s, a) = \mathbb{E}\{G_t \mid S_t = s, A_t = a, A_{t+1:\infty} \sim \pi\} \quad q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

$$v_*(s) = \max_{\pi} v_\pi(s) \quad v_* : \mathcal{S} \rightarrow \mathbb{R}$$

$$q_*(s, a) = \max_{\pi} q_\pi(s, a) \quad q_* : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

Figure 2.1: 4 value functions utilizzate per la policy update

2.1.2 Q-learning off policy control

Il Q learning, come la maggior parte degli algoritmi di RL, va alla ricerca di un trade-off tra *exploration* e *exploitation*: nel caso del Q learning si cerca, tramite un fattore di esplorazione (*exploration rate*), di spingere l'algoritmo verso una scelta esplorativa piuttosto che di apprendimento, cercando di mantenere quindi una politica maggiormente esplorativa all'inizio del training per poi spostare l'attenzione sul fattore di apprendimento. Il fatto che si tratti di un algoritmo off policy richiede in ingresso parametri aggiuntivi oltre a quelli forniti dall'ambiente (dimensione dello stato, numero di azioni...). Tre sono i parametri aggiuntivi: α, ϵ, γ che rappresentano rispettivamente

- learning rate: determina con quale estensione le nuove informazioni acquisite sovrascriveranno le vecchie informazioni; un fattore 0 impedirebbe all'agente di apprendere, al contrario un fattore 1 farebbe sì che l'agente si interessi solo delle informazioni recenti
- exploration rate
- discounting rate: dovrebbe essere l'unico fattore non soggetto a decadimento, dato che esso determina l'importanza delle ricompense future, bilanciando reward immediato e futuro. Un fattore pari a 0 renderà l'agente opportunistico, facendo sì che consideri solo le ricompense attuali, mentre un fattore tendente ad 1 renderà l'agente attento anche alle ricompense che riceverà in futuro a lungo termine.

Per semplificare la trattazione, in questo contesto si è deciso di non considerare il decadimento dei parametri.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Figure 2.2: Equazione di *Bellman* utilizzata per andare ad aggiornare i valori della funzione Q

Questa formula per l'update è stata implementata direttamente nel codice (come si vede in figura 2.3) all'interno dell'algoritmo 1: in particolare è da evidenziare come il pedice t applicato allo *state* e *action* rappresenta la condizione attuale; il pedice $t+1$ applicato allo stato, vuole invece rappresentare lo stato successivo che si raggiunge eseguendo l'azione A all'istante t .

```
def update_q(state_old, action, reward, state_new, t):
    alpha = 0.1 # Alpha
    discount_rate = 0.9 # gamma
    q_new = alpha * (reward + discount_rate * np.max(np.array([Q[state_new] for a in actions]) - Q[state_old][action]))
```

Figure 2.3: Aggiornamento Q table

2.1.3 Scelta dell'azione e Q table

Il metodo *choose action* mostrato in figura 2.4, permette la scelta dell'azione da intraprendere sulla base del trade off illustrato precedentemente tra esplorazione e apprendimento: si va infatti a scegliere

Algorithm 1: Q learning per la stima della policy π

Data: Parametri dell'algoritmo: step size α (con $0 \leq \alpha \leq 1$), $\epsilon > 0$

while non sono terminati gli episodi **do**

 Inizializza lo stato S

while non sono terminati gli step **do**

 Scegli azione A usando la policy derivata da Q (per esempio epsilon-greedy)

 Esegui l'azione A , osservando il reward (R) ottenuto e il nuovo stato (S') raggiunto

 Aggiorna $Q(S,A)$ secondo la formula 2.2

$S \leftarrow S'$

 Continua fino a che non si raggiunge uno stato terminare

end

end

la policy greedy che consiste in una scelta randomica dell'azione, oppure viene utilizzata la target policy a cui corrisponde il valore maggiore nella tabella stato-azione. Si capisce quindi come Q-learning (in genere) sia in grado di andare a tener conto del problema di avere un trade-off tra:

- **Exploiting:** scegliendo l'azione basata sul massimo valore della specifica azione, l'agente continua a rimanere fissato in quel punto, sfruttando al massimo il reward ottenibile, senza cercare eventuali altri stati in grado di fornire maggior reward;
- **Exploring:** selezionando randomicamente un'azione, rendiamo l'agente in grado di esplorare e scoprire nuovi stati, che altrimenti non avrebbe mai esplorato.

```
def choose_action(self, state, t):
    epsilon = 0.3 # e for greedy: explore_rate

    # Select a random action
    if random.uniform(0, 1) < epsilon:
        # Choose action randomly
        if random.uniform(0, 1) > 0.5:
            action = 0
        else:
            action = 1
    # Select the action with the highest q
    else:
        action = np.argmax(self.Q[state])
        if self.debug:
            print("Action " + str(action))
            print("State " + str(state))
            print("Maximize Q table that has this fields " + str(self.Q))
    return action
```

Figure 2.4: Metodo per la stima della azione da eseguire

In particolare risulta quindi essere necessario spostarsi da uno spazio degli stati continuo ad uno discreto, con un numero finito e preferibilmente piccolo di stati discreti: infatti, meno stati abbiamo, più piccola risulterà essere la Q-table, e meno step l'agente dovrà eseguire per apprendere correttamente i valori che compongono la tabella.

Tuttavia, pochi stati, potrebbero non essere sufficienti per rappresentare l'ambiente: nella nostra implementazione si è deciso di andare a discretizzare solamente 2 delle 4 variabili di stato, ovvero l'angolo θ e la velocità angolare θ' della barra; per quanto riguarda invece la posizione e la velocità del carrello, non discretizzandole, le si va a mappare come dei singoli valori scalari. La motivazione è il fatto che la probabilità del carrello di lasciare l'ambiente a destra o a sinistra è molto bassa, dando quindi maggior peso alla riduzione della dimensionalità. Questa discretizzazione degli stati si traduce, a lato pratico, in una matrice con una struttura particolare, la quale dipende dal numero di bucket che si stanno utilizzando: supponiamo di aver settato, all'interno del nostro script Python, un vettore `buckets = (1, 1, 4, 5,)`: la Q-table risultante da questa discretizzazione è quella rappresentata in figura 2.5 in cui si può vedere come, per ogni discretizzazione dell'angolo corrisponde un nuovo blocco di un

numero di righe pari alla discretizzazione richiesta invece per la velocità angolare. All'interno di ogni singola riga troviamo due differenti valori, i quali rappresentano appunto i Q-values relativi alle due possibili azioni eseguibili in ogni diverso stato.

```
[[[[[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]]]

[[[219.9672504 26.39727031]
[273.51855531 193.83737452]
[310.88171961 232.16323458]
[140.28894227 311.43550361]
[ 0. 29.53193065]]]

|

[[[267.92204978 0. ]
[333.79412619 169.65964031]
[363.10841077 355.99389624]
[353.81618131 363.27612953]
[280.93959945 342.48619415]]]

[[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]
[ 0. 0. ]]]]]]
```

Figure 2.5: Possibile struttura della Q-table

2.1.4 Result

La figura 2.6 riporterà l'andamento del reward in fase di training. L'algoritmo si considera risolto quando completa con successo 195 episodi consecutivi (sui 200 massimi disponibili): si nota infatti come nel grafico tutti gli ultimi total reward registrati siano sopra la soglia di vincita dell'algoritmo (linea verde). L'andamento espresso dalla linea rossa invece indica un valore più *smooth* del reward.

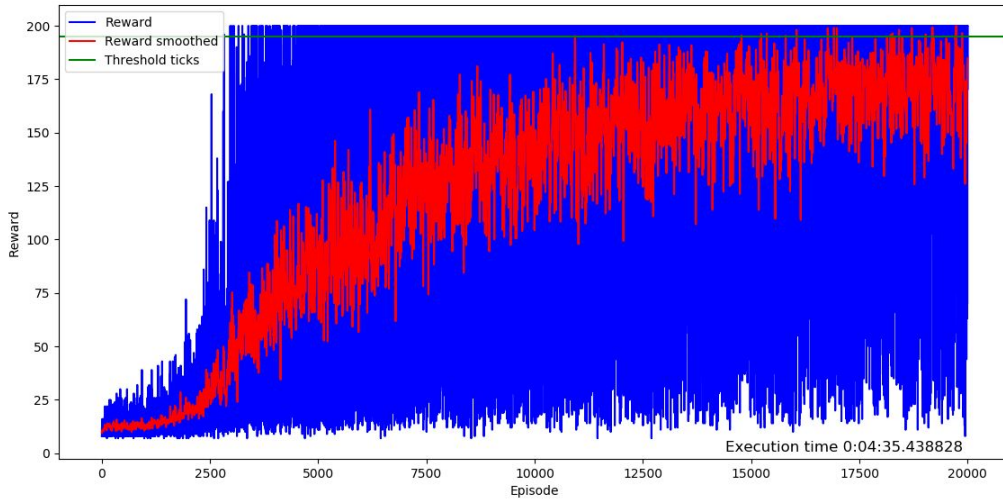


Figure 2.6: Andamento del Q learning

Per andare a realizzare un confronto tra i diversi algoritmi, si è runnato l'agoritmo di Q-learning a bucket con un batch di 2000 iterazioni, ognuna delle quali aveva una *time threshold* di 200 iterazioni: come si può vedere in figura 2.7 il limite di iterazioni minime in cui il pole viene mantenuto in equilibrio non viene raggiunto. Si può notare anzi come l'agente non venga *trainato* a sufficienza, raggiungendo prestazioni molto basse, rispetto ai successivi algoritmi.

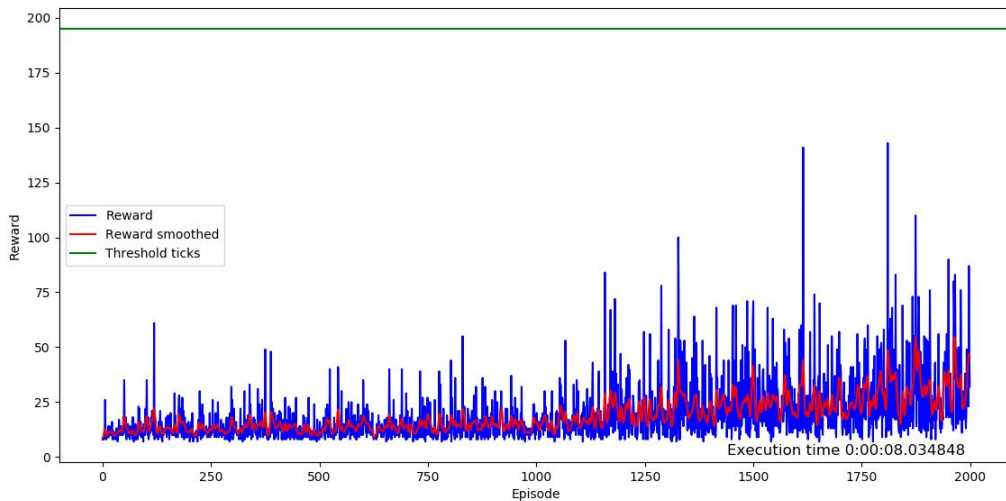


Figure 2.7: Andamento del Q learning confrontabile con altri algoritmi implementati

2.2 Algoritmo DQN

La soluzione al problema di discretizzazione degli stati con conseguente perdita di informazioni che potrebbe risultare importanti e decisive al learning dell'ambiente, viene risolta grazie all'algoritmo di Deep Q-network: la Q-table viene infatti sostituita da una rete neurale, la quale non necessita più di alcuna forma di discretizzazione che approssima la funzione valore.

La rete prende come input lo stato e produce una stima della funzione valore per ogni azione: in particolare si tratta di una rete *fully-connected*, in cui l'input della network sono le quattro variabili di stato

- Posizione (x)
- Velocità lineare (\dot{x})
- Angolo (θ)
- Velocità angolare ($\dot{\theta}$)

mentre le uscite rappresentano i Q-values per le due possibili azioni, ovvero il movimento a destra e a sinistra, come si può vedere in figura 2.8.

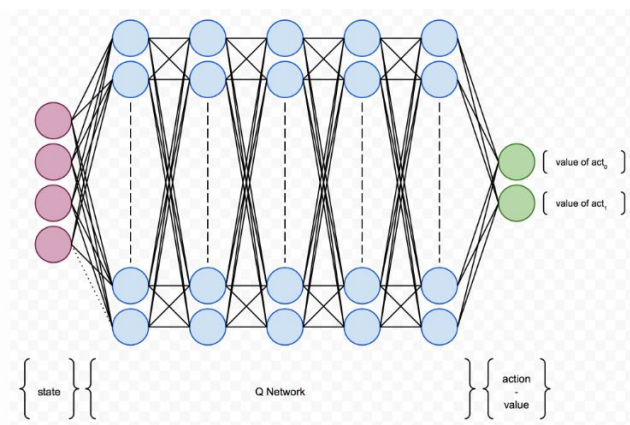


Figure 2.8: DQN network

2.2.1 Q network

Nell'implementazione, per andare a realizzare la rete neurale, è stata utilizzata la libreria python TensorFlow, basandosi sulle API fornite da Keras, per una facile creazione della rete neurale, in

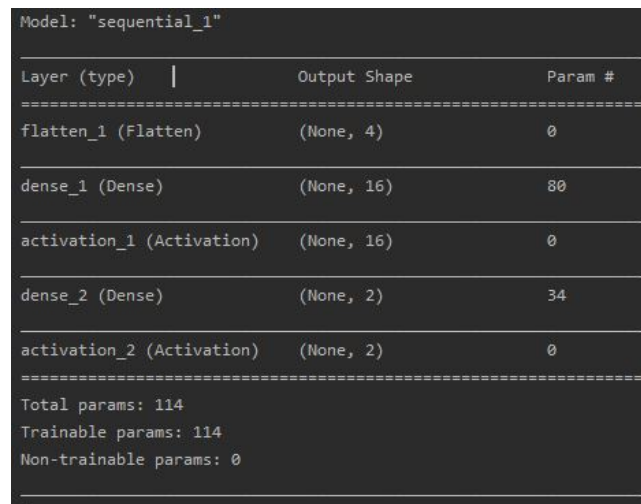
maniera molto rapida e concisa.

```

1      model = Sequential()
2      model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
3      model.add(Dense(16))
4      model.add(Activation('relu'))
5      model.add(Dense(nb_actions))
6      model.add(Activation('linear'))

```

Tramite l'utilizzo di una rete neurale quindi, andiamo a sostituire l'update della Q table, con il train della nostra rete neurale: come infatti ben sappiamo, il modello *DQN neural network* è un modello di regressione, il cui output è tipicamente un valore continuo (*float value*, che rappresenta direttamente valore della nostra Q function. L'agente, come sempre, andrà a scegliere l'azione che presenta il Q-value maggiore, che indica appunto quale è l'azione che si presume che andrà a dare un maggiore reward nel futuro.



Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 4)	0
dense_1 (Dense)	(None, 16)	80
activation_1 (Activation)	(None, 16)	0
dense_2 (Dense)	(None, 2)	34
activation_2 (Activation)	(None, 2)	0
Total params: 114		
Trainable params: 114		
Non-trainable params: 0		

Figure 2.9: Modello della rete neurale: 4 input *state* e 2 output *Q-action*

2.2.2 Replay memory

Introducendo una rete neurale, invece della Q-table utilizzata durante la prima versione del Q learning, la complessità del nostro ambiente può crescere significativamente, senza richiedere necessariamente più memoria: come si può facilmente vedere, un environment a celle con grandezza 50×50 , andrebbe ad esaurire facilmente la memoria della maggior parte dei pc in commercio. Invece, con una rete neurale, anche con ambienti complessi, non dobbiamo affrontare problemi legati ai requisiti di memoria. Per gestire questo utilizzo di memoria, un concetto da illustrare è quello dell'experience replay, codificato tramite il comando presentato in figura 2.10.



```

# ===== SETUP OF MEMORY =====
memory = SequentialMemory(limit=50000, window_length=1)

```

Figure 2.10: Creazione variabile per sequential memory

Gli algoritmi di reinforcement learning presentano infatti spesso il problema di avere un alto livello di correlazione tra esperienze successive con l'alto rischio di portare ad un veloce overfitting sui dati a disposizione, evitando così la necessaria generalizzazione.

L'idea della experience replay è quindi quella di salvare le esperienze in una memoria chiamata replay memory e durante ogni passo di learning recuperare in maniera randomica un campione di tali transazioni, per andare così ad introdurre una forte incorrelazione tra misure sequenziali. Questo è dimostrato come esso vada a stabilizzare e a migliorare la procedura di training della rete neurale DQN.

2.2.3 Result

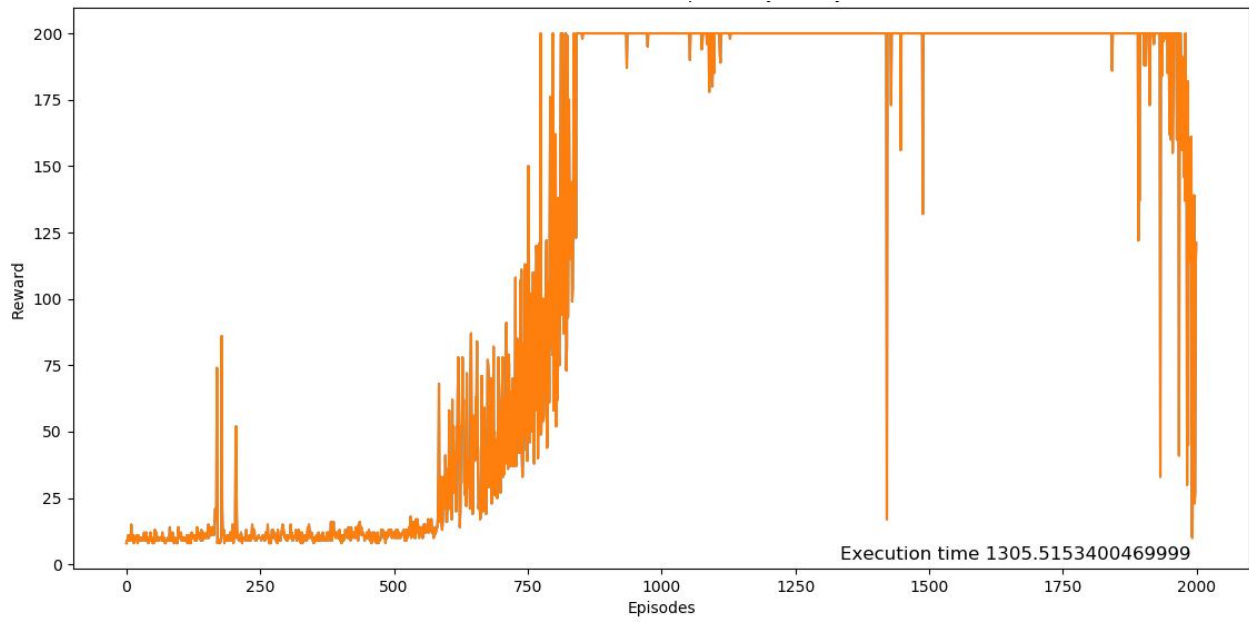


Figure 2.11: Reward con DQN network

2.3 Algoritmo *Finite Differences*

Negli algoritmi visti fin'ora, per trovare la policy adatta ($a = \pi(s)$ dove con π indichiamo appunto la *policy*) siamo sempre andati ad utilizzare una funzione approssimata $Q_{\pi^*}(s, a)$, la quale permetteva all'agente di scegliere quale fosse l'azione migliore da compiere, sia nel caso di azioni discrete, sia nel caso di azioni continue che discrete (come si vede in figura 2.12) andando poi a selezionare l'azione in grado di massimizzare l'output.

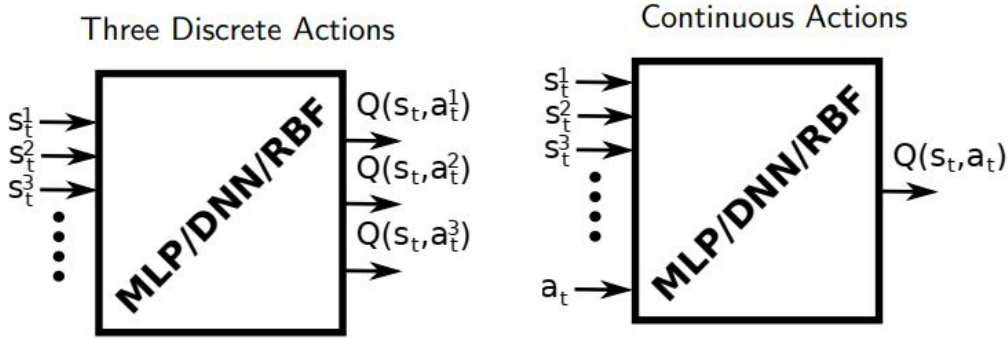


Figure 2.12: Valutazione della funzione sulla base della Q-values

La domanda che ci poniamo è: *possiamo trovare la policy direttamente* partendo dallo stato attuale (s_t) e ottenendo appunto a_t ?

La risposta è sì.



Figure 2.13: Valutazione della funzione *non* basandosi sulla Q-values

Questo concetto, mostrato in maniera molto diretta nella figura 2.13, è quello che nella letteratura del RL viene chiamato come metodo di *policy search*, il quale permette appunto di stimare la policy ottimale ($\pi^*(a|s)$) invece di cercare il valore ottimo per la funzione $Q^*(s, a)$: questo nuovo approccio presenta innumerevoli vantaggi rispetto agli algoritmi mostrati in precedenza, in particolare:

- Policies ottimali, scelte tramite algoritmi di questo tipo, presentano un numero di parametri minore rispetto al valore ottimale della q-functions (*curse of dimensionality*)
- Processo di learn più veloce (e questo è anche dimostrato dai grafici presenti all'interno di questa documentazione)
- Offre la possibilità di utilizzare policies sia deterministiche che stocastiche (*noisily*)

E cosa cerchiamo di ottimizzare in questo nuovo contesto? I metodi di policy search vanno ad ottimizzare l'indice $J(\theta)$, dove θ rappresenta il vettore dei pesi; è possibile realizzare questa ottimizzazione utilizzando differenti metodi:

- Gradient free methods:

- Evolutionary computation
- Simulated annealing
- Hill climbing
- **Gradient based methods** (*policy gradient methods*):
 - Gradient estimation
 - Optimization algorithm

I passi principali che sono stati implementati, sono

- Implementare la dinamica del carrello (di cui non si riporta la trattazione) e una funzione per simulare i roll-outs (sotto sezione 2.3.2) che forniscono il reward di ritorno;
- Utilizzare una *Radial Basis Function* (RBF network - sotto sezione 2.3.2) per approssimare l'apprendimento, ovvero per ottenere $a = \pi(s, W) = W^T \Phi(s, a)$
- Implementare l'algoritmo alla differenze finite (FD) per apprendere i pesi della policy

2.3.1 Approccio *Policy gradient estimation*

Come già sottolineato in precedenza andremo a basarci su un algoritmo *gradient based*, nello specifico l'algoritmo alla differenze finite, il quale, in ambito matematico, rappresenta una strategia utilizzata per risolvere numericamente equazioni differenziali che si basa sull'approssimazione delle derivate con equazioni alle differenze finite.

Questa approssimazione che andiamo ad utilizzare per effettuare un'operazione di minimizzazione (anche locale) sulla funzione $J(\theta) = V_\pi(s)$: l'approccio che si è deciso di seguire non è però quello classico, ma bensì una strada alternativa, in cui, fornita una policy parametrizzata $\pi(a|s, \theta)$, possiamo calcolare $J(\theta)$ simulando l'environment come un *Markov Decision Process*, la funzione $J(\theta)$.

In particolare:

- Invece di calcolare $\partial V / \partial \phi_i$ separatamente (passaggio che verrebbe naturale dopo aver definito $J(\theta)$ in 2.3.1), andiamo ad ottenere un vettore random $\delta \sim N(0, \sigma^2)$. Nel codice questa parte è stata implementata in questo modo:

```

1      # Variance of the random Gaussian perturbation of the parameters
2      variance_of_perturbation = 0.1
3      # random parameter variation (Gaussian)
4      delta = variance_of_perturbation * np.random.randn(numberOfCentrum, 1)
```

- Il vettore randomico generato in precedenza lo utilizzeremo poi per calcolare, tramite la tecnica definita *roll-out* (sezione 2.3.2), due valori della funzione J (J_+ , J_-)
- Utilizzando l'approssimazione di Taylor possiamo scrivere $J_+ \approx J(\Theta) + \nabla J^T \delta$ e $J_- \approx J(\Theta) - \nabla J^T \delta$
- La differenza $J_\Delta = J_+ - J_-$ diventa $J_\Delta = 2\delta^T \nabla J$, dove J_Δ è un numero scalare mentre ∇J e δ sono vettori.

Questo processo andiamo a ripeterlo più volte con differenti valori randomici di δ_i con $i = 1, 2, \dots, N_H$: andremo quindi ad ottenere così differenti J_{Δ_i} , ma ci sarà solamente un gradiente (∇J sconosciuto).

In forma matriciale quindi diventa: La soluzione dovrebbe essere data da $\nabla J = 1/2 \Delta^{-1} J_\Delta$, ma Δ potrebbe non essere una matrice quadrata (per il fatto che il numero di rollout potrebbe non essere pari alla dimensione di Θ): con alcuni passaggi matematici (qui omessi) possiamo andare a una forma più stabile per ricavare la soluzione (riportata anche in precedenza) data da $\Delta J = 1/2 [\Delta^T \Delta + \lambda I]^{-1} \Delta^T J_\Delta$

$$\bar{J}_\Delta = \begin{bmatrix} J_{\Delta_1} \\ J_{\Delta_2} \\ \vdots \\ J_{\Delta_{N_H}} \end{bmatrix} = 2 \begin{bmatrix} \dots & \delta_1^T & \dots \\ \dots & \delta_2^T & \dots \\ \vdots & \vdots & \vdots \\ \dots & \delta_{N_H}^T & \dots \end{bmatrix} \nabla J = 2\Delta \nabla J$$

Now solve for ∇J !

Figure 2.14: Matrix form

2.3.2 Algoritmo *FD*

Questo è l'algoritmo implementato nel codice per apprendere i parametri ottimi. La corrispondenza implementata in *Python* dell'algoritmo 2 la si trova nella figura 2.15.

Algorithm 2: Finite Differences algorithm

Input: Input: approssimazione della funzione parametrica $\pi(\bullet, \bullet, \Theta)$

Output: Output: pesi ottimali Θ^*

Data: Parametri: *Learning rate* α , λ , σ_δ^2 , numero di rollout

while *Policy non converge* **do**

 Inizializza J_Δ e Δ

for $i = 1, 2, \dots, N_H$ **do**

 Inizializza lo stato iniziale s_0

 Genera una variazione randomica δ distribuita come $N(0, \sigma_\delta^2)$

$J_+ \leftarrow \text{rollout}(\pi(\Theta + \delta))$

$J_- \leftarrow \text{rollout}(\pi(\Theta - \delta))$

 Add $J_+ - J_-$ al parametro J_Δ

 Add δ^T a Δ

end

$\Theta^* \leftarrow \Theta^* + 1/2\alpha[\Delta^T \Delta + \lambda I]^{-1} \Delta^T J_\Delta$

end

```
for rollout in range(0, int(round(rolloutMax, 0))):
    # Inizializza lo stato iniziale s0
    random_initial_state = np.array([[0], [2 * thresholdAngle * (np.random.rand() - 0.5)], [0], [0]]) # random initial state

    # Genera una variazione randomica
    delta = variance_of_perturbation * np.random.randn(numberOfCentrum, 1) # random parameter variation (Gaussian)

    # Perturbation
    J_positive = cart_rollout(random_initial_state, W + delta) # Positive change return
    J_negative = cart_rollout(random_initial_state, W - delta) # Negative change return

    difference = np.array([J_positive - J_negative])
    dJ = np.concatenate((dJ, difference), axis=0) # Add return variation to dJ
    if len(Delta) == 0:
        Delta = np.transpose(delta)
    else:
        Delta = np.concatenate((Delta, np.transpose(delta)), axis=0) # Add random perturbation on the parameters to Delta
```

Figure 2.15: Reward *Finite Differences* in *Python*

Rollout

Fornita una policy $a = \pi(s)$ oppure $a = \pi(a|s)$ e uno stato iniziale s_0 ad un certo istante temporale di un *Markov Decision Process*, un rollout (definibile anche come *traiettoria*, *storia* o *trial*), è semplicemente una sequenza $H = s_0, a_0, s_1, a_1, \dots, s_T, a_T$ generata a partire da uno stato s_0 e seguendo una policy π . Nella implementazione relativa a questo progetto, come spiegato nella sotto sezione 2.3.2, siamo andati ad utilizzare come approssimazione della policy da seguire il risultato fornito in uscita da una *Radial Basis Function* neural network.

```
def cart_rollout(initialState, w):
    next_state = initialState
    time = 0
    reward = 0

    while time < thresholdTime and abs(next_state[1]) < thresholdAngle:
        # Use the linear RBF network to approximate learn
        # action = pi(s,W) = W' * phi(s,a)
        phi_value = phi(next_state)
        u = np.dot(w.transpose(), phi_value)
        step_state = cart_dynamics(next_state, u)
        step_state *= sampleTime
        next_state = next_state + step_state
        if abs(next_state[1]) < thresholdAngle:
            reward = reward + sampleTime
        time = time + sampleTime
    return reward
```

Figure 2.16: Rollout in Python

I pesi di questa rete neurale, come si può vedere nell'implementazione in figura 2.16, risultano però essere fissati, il che quindi equivale all'andare a seguire sempre la stessa policy.

RBF network

Come è stato evidenziato in precedenza, siamo andati ad utilizzare una *RBF* lineare per andare ad approssimare l'apprendimento.

In particolar modo le *Radial Basis Function* permettono di andare a partizionare lo spazio degli stati, sovrapponendo dei "*Gaussian neurons*", in cui ogni neurone genera un segnale corrispondente al vettore di input: il segnale prodotto da ogni neurone presenta una potenza che dipende dalla distanza tra il centro della curva gaussiana che rappresenta il vettore e il vettore degli stati in input.

Questo concetto base delle *RBF* viene implementato andando a creare una tabella formata da una sequenza di punti cartesiani, i quali rappresentano i centri di ognuno dei neuroni della rete neurale, ovvero la media di ogni curva gaussiana (figura 2.17).

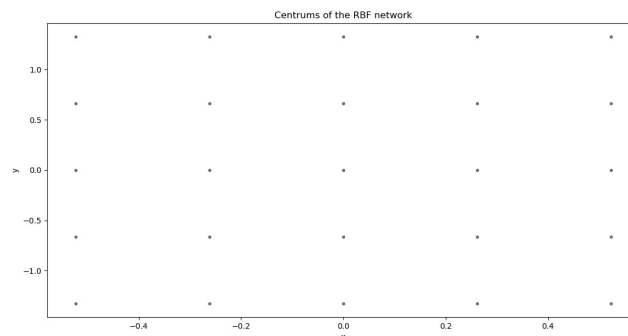


Figure 2.17: Centri di ogni neurone della *RBF*

Un altro aspetto importante da evidenziare è il fatto che, i *weights* relativi alla *RBF* vengono aggiornati tramite algoritmo Finite Difference: quindi, i

2.3.3 Result

Nella figura 2.18, è rappresentato l'andamento del reward con un batch di 10 trial con 70 iterazioni ciascuno: ogni iterazione può essere eseguita per un massimo di 10 s, nel caso in cui l'esecuzione non fallisce prima (ovvero il pole cade fuori dal range angolare imposto). Si nota quindi come siano necessarie meno iterazioni, rispetto agli algoritmi precedenti, per ottenere un reward molto buono.

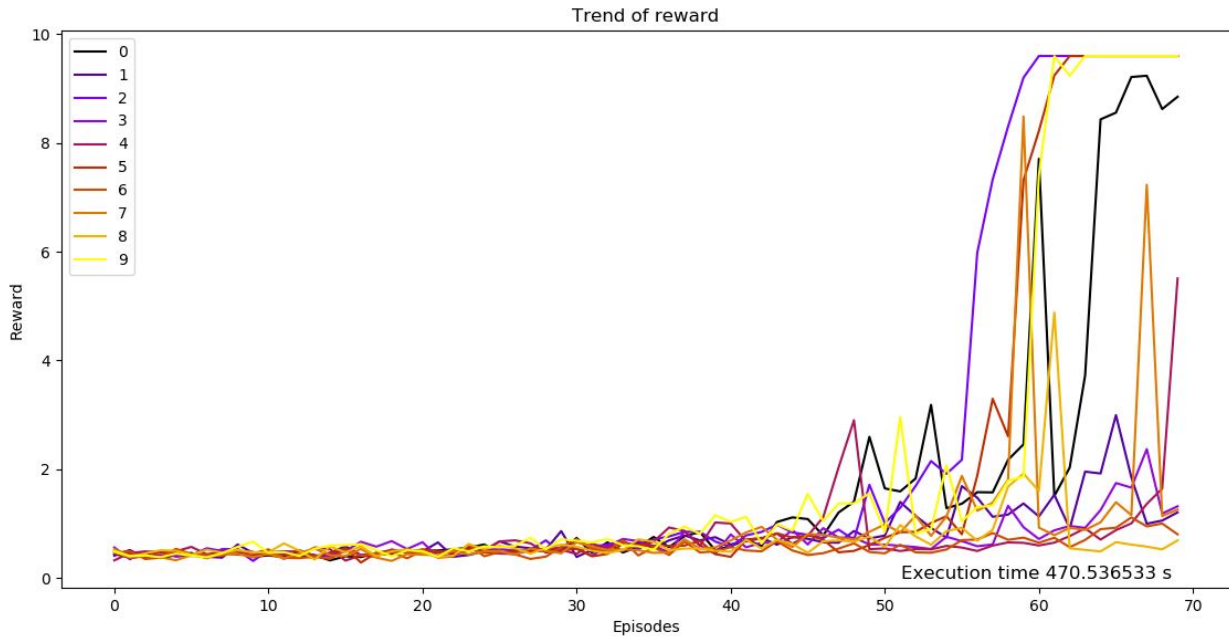


Figure 2.18: Reward *Finite Differences*

Bibliography

[1] Eco, Umberto (1977), *Come si fa una tesi di laurea*, Bompiani, Milano.

[2] Mori, Lapo Filippo (2007), "Scrivere la tesi di laurea"