Istituto Tecnico Tecnologico "E. Barsanti"
Via dei Carpani 19/B – 31033 – Castelfranco V. (TV)
Tel. 0423 – 492847 – 493614
email:info@barsanti.edu.it

ESAME DI STATO 2020.2021

ELABORATO DELLE DISCIPLINE D'INDIRIZZO

(individuate come oggetto della seconda prova scritta – Articolo 18, O. M. n. 52)

COGNOME E NOME DEL CANDIDATO: Tinfena Lorenzo Classe 5^AI

ARGOMENTO DELL'ELABORATO:

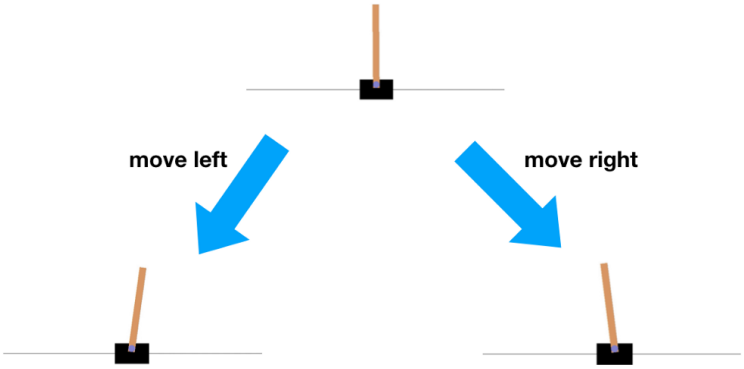
Sviluppo di un'applicazione di intelligenza artificiale

Indice

1 Introduzione

1.1 Scopo

In questo elaborato vorrei spiegare come si può arrivare a realizzare una intelligenza artificiale che riesca a capire quando sbaglia, o quando compie azioni corrette e in base a questo si migliora continuamente; questa è la forma di intelligenza artificiale più generalizzata attualmente. La applicazione di esempio è un pendolo inverso, semplicemente un'asse appesa ad un perno, che deve muoversi a destra o a sinistra cercando di non far cadere l'asse. Tutte le implementazioni degli algoritmi che ho usato nel progetto allegato sono scritti da zero in Python.



Ma prima di arrivare a capire come funziona, dobbiamo partire dalle basi, quindi da cosa è un'intelligenza artificiale, il machine learning, il concetto di reinforcement learning, per arrivare all'algoritmo che ho usato, chiamato deep Q-learning.

1.2 Cos'è un'intelligenza artificiale?

Per intelligenza artificiale intendiamo la capacitá che ha una macchina di riprodurre un'intelligenza biologica, quella che tutti conosciamo, e che hanno gli essere umani, animali e piante.

Generalizzando, possiamo dire che per intelligenza (sia artificiale, sia biologica) di un'entitá all'interno di un ambiente è la sua capacitá di scegliere azioni corrette da compiere in base al suo stato e alla sua esperienza. La azione scelta è calcolata come $A = E(S)$, dove A è l'azione scelta, E è l'esperienza, e S è lo stato.

L'unica differenza tra stato ed esperienza è puramente logica, cioè, mentre il primo fa parte dell'entità, il secondo deriva dall'ambiente con cui l'entità interagisce.

Come, ad esempio, un'essere umano deve camminare deve scegliere quali muscoli muovere in uno spazio (ambiente), in base alla loro posizione (stato), e alla sua esperienza (che dice in modo generale come muoversi).

1.2.1 Relazione tra intelligenza generale e correttezza delle azioni

L'intelligenza di un'entità è proporzionale alla correttezza delle azioni compiute dall'entità alla variazione del suo stato e della sua esperienza; quindi, come la correttezza è soggettiva, anche l'intelligenza lo è.

1.2.2 Criteri di correttezza delle azioni

Per questo motivo, per creare delle intelligenze artificiali che aiutino l'uomo, i criteri di correttezza dovranno essere soggettivi rispetto agli utilizzatori; ad esempio una persona A considera intelligente un robot che pulisce casa una volta la settimana, invece una persona B pensa il contrario, perchè vorrebbe che pulisse casa 5 volte al giorno. Nonostante il fatto che possiamo includere nello stato dell'entità robot la persona a cui essa deve pulire casa, e quindi rendere l'intelligenza un po' più oggettiva, quest'ultima non potrà mai esserlo appieno.

1.3 Come si forma l'esperienza di un'entità?

L'esperienza, che assieme allo stato dell'entità, determina le azioni da compiere, possiamo considerarla la strategia dell'entità e si divide in 2 grandi tipi:

- Esperienza statica (solo nelle entità artificiali)

  Da un punto di vista matematico, possiamo riconoscerla come una funzione non parametrica.

  Ad esempio un programma (entità artificiale) che dato il tragitto di una macchina, quindi lo spazio percorso in chilometri, e la durata in ore (che assieme formano lo stato), e una funzione non parametrica (esperienza statica) $f(spazio, durata) = \frac{spazio}{durata}$, calcola la velocità media (l'azione è il valore stesso della velocità media)

- Esperienza dinamica (sia nelle entità artificiali, sia in quelle biologiche)

  Da un punto di vista matematico, possiamo riconoscerla come una funzione parametrica.

  Ad esempio un programma (entità atificiale) che data la superficie di una casa (stato) deve predire il suo prezzo (l'azione scelta è il valore stesso), non può usare una funzione non parametrizzata, perchè ci sono molte variabili che incidono sul prezzo, come la posizione o l'anno di costruzione. Potremmo includere queste variabili nello stato assieme alla superficie, ma comunque si fa fatica a scrivere una funzione che ne calcoli il prezzo; per questo motivo si scrive una funzione molto generalizzata parametrizzata secondo delle variabili determinate in modo dinamico, cercando di massimizzare l'accuratezza (o intelligenza) dell'entità.

2 Reinforcement learning

2.1 Machine learning generale

Mentre in un'entità artificiale con esperienza statica è compito del programmatore determinare l'esperienza statica adatta a massimizzare la sua accuratezza, quando si ha a che fare con un'esperienza dinamica, è l'entità artificiale stessa (almeno nella maggior parte dei casi) a modificare la propria esperienza, e solo in questo caso per permettere all'entità di fare ciò le devono essere forniti dei criteri di correttezza.

L'insieme di processi che regolano l'esperienza è chiamato machine learning.

2.1.1 Tipi di machine learning

Perchè un'entità modifichi la sua esperienza in modo da massimizzare la sua accuratezza, qualcuno deve fornirgli dei criteri di correttezza. I modi per i quali un'entità modifica la sua esperienza dati i criteri di correttezza, determinano vari tipi di machine learning.

- Supervised learning (esperienza dinamica)

  Nel supervised learning i criteri di correttezza forniti sono una serie di record (chiamato dataset, o più nello specifico training-set), dove ogni record contiene uno stato, e una azione considerata corretta, ci sono molti modelli/algoritmi nel supervised learning, ma l'idea generale è che l'esperienza viene costantemente modificata in modo che per ogni stato, viene scelta una azione quanto più possibile simile a quella corrispondente nel dataset.

- Unsupervised learning (esperienza statica)

  Nel unsupervised learning l'esperienza, cioè come deve funzionare in pratica è determinata dall'inizio, quindi statica, perchè lo scopo dei modelli/algoritmi di questo tipo di machine learning sono generalmente di raccogliere intuizioni/statistiche riguardanti dei dati, ad esempio un singolo stato contiene una lista di immagini, che deve classificare cercando le differenze (la lista di predizioni è la azione scelta).

- Reinforcement learning (RL) (esperienza dinamica)

  Il metodo più generale per fornire criteri di correttezza è semplicemente dire quanto è corretta una azione scelta dall'entità, dato uno stato. Questo valore è chiamato reward e l'esperienza sarà modificata in modo da cercare di massimizzarlo.

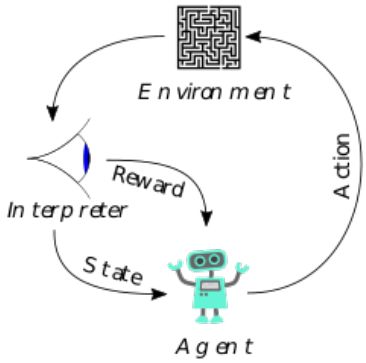
  Dare un reward (nella maggior parte continuo, quindi non discreto) all'entità per ogni azione che compie, è un workflow estremamente generalizzato. Un esempio siamo noi stessi, che con

i sensi capiamo l'ambiente che ci circonda e compiamo delle azioni, cercando di massimizzare la nostra felicità (cioè il reward).

Per ogni stato, azione, e fattore random (solo in certi ambienti), corrisponde un reward.

2.2 Workflow di base

Per riassumere: fornire un feedback (reward) scalare, per ogni azione e stato, è il più semplice e generale criterio di correttezza, mentre a modificare l'esperienza con i reward, ci penseremo tra poco. Ora capiamo meglio il workflow di base di un'entità artificiale di reinforcement learning.



L'agente riceve inizialmente il primo stato s, poi in base ad esso ed alla sua esperienza, compie una azione a, un interprete (che in certi casi corrisponde all'ambiente stesso), riceve s' (stato successivo), valuta la scelta di a (in base a s, s' ed a volte anche in base ad un fattore casuale che poi andremo a formalizzare), il reward r' scelto dall'interprete viene dato all'agente assieme ad s'.

Vediamo un piccolo esempio in python di un agente che compie azioni casuali:

```
import gym
env = gym.make('CartPole-v1')
# inizializzo le metriche
total_reward = 0
steps = 0
current_state = env.reset() # ottengo il primo state
done = False
while not done: # quando done è True l'episodio finisce
    action = env.action_space.sample() #ottengo una azione casuale da A dell'environment
    next_state, reward, done, _ = env.step(action) # compio la azione
    print(f'Transition from state {current_state} to state {next_state}, '
    + f'I earned reward: {reward} and now the episode is done is {done}')
    # aggiorno le metriche
    total_reward += reward
    steps += 1
    current_state = next_state # aggiorno il current_state
print(f'Episode done in {steps} steps, total reward {total_reward}')
```

2.3 Formalizzazione

Per cominciare, è necessario capire bene questi concetti:

- Gli stati a volte sono chiamati osservazioni.
- Tutti i possibili stati fanno parte di un observation space.
- Come gli stati, anche le azioni fanno parte di un action space.
- Ogni space può essere discreto, continuo, o un insieme di altri space. Ad esempio uno space discreto è definito come $[0,1,..n]$, dove n è il numero di stati discreti, oppure un observation space continuo è un tensore di scalari con un valore minimo e uno massimo.

Per formalizzare un workflow di reinforcement learning, viene visto sotto forma di un particolare tipo di markov chain, cioè un markov decision process (MDP): un modo per rappresentare un observation space discreto (S) di un ambiente e il suo action space discreto (A). Un MDP non fa parte dell'esperienza dell'entità, è semplicement un modo per rappresentare le transizioni nel reinforcement learning.

Attualmente stiamo quindi considerando degli ambienti con observation space ed action space discreti, ma le equazioni derivate da un MDP si useranno anche per observation ed action space continui, come vedremo più avanti.

2.3.1 Tipi di MDP

- Fully observable MDP

  Oltre ad S e A, comprende anche una matrice P (denominata anche T), che può essere di tipo stocastico, tale che $P_{s,a,s'}$ (oppure $p(s'|s,a)$) è la probabilità di transitare dallo stato s allo stato s' avendo compiuto la azione a; oppure deterministico: $s' = P_{s,a}$

  Inoltre ha una matrice R, tale che $R_{s,a,s'}$ è il reward ottenuto dalla transizione dallo stato s allo stato s' avendo compiuto la azione a. In alcuni MDP con gli stessi s, a e s', si possono avere reward diversi, quindi la matrice R può rappresentare la media di tali valori ad esempio.

  Solitamente P è espresso in modo stocastico per generalizzare; al massimo si avrà una distribuzione di probabilità deterministica.

  Questo tipo di MDP rispetta la markov property, la quale dice che le transizioni e i reward successivi ad uno tempo t, sono indipendenti dalle transizioni con tempi precedenti a t. Solitamente quando si parla di MDP in generale, ci si riferisce a questo tipo. $MDP = (S,A,P,R)$

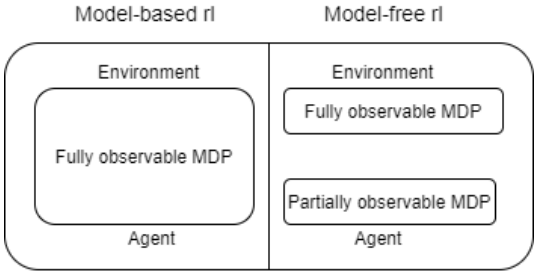
- Partially observable MDP

  Non ha una matrice di probabilità P, e una di reward R, perchè sono sconosciute, si può solo cercare di approssimarle. Questo tipo di MDP può non rispettare la markov property e nella prossima sezione capiremo il perchè.

Le dinamiche dell'ambiente sono spesso descritte usando un Fully observable MDP, mentre le dinamiche dell'agente possono essere descritte usando lo stesso Fully observable MDP del suo ambiente, oppure un altro Partially observable MDP. Nella prossima sezione si farà chiarezza.

2.4 Model-based e model-free rl

Ci sono 2 tipi di reinforcement learning, con algoritmi differenti: model-based rl, e model-free rl, si differenziano da quanto l'agente conosce dell'ambiente, formalmente la differenza la possiamo definire usando i 2 tipi di MDP descritti sopra.



- Model-based rl

  In questo tipo di rl, un Fully observable MDP descrive le dinamiche dell'agente e dell'ambiente, che infatti corrispondono, quindi l'agente può prevedere tutte le transizioni. Questo tipo di rl possiamo ritrovarlo ad esempio in alcuni videogiochi come snake, dove ogni stato dell'ambiente (matrice) corrisponde allo stato dell'agente (artificiale); però si fa fatica a trovare applicazioni reali, perchè ad esempio un umano, per dirlo in modo semplificato, non può conoscere quello che accade in ogni parte del mondo.

- Model-free rl

  Nel model-free rl, le dinamiche dell'agente sono descritte da un Partially observable MDP, mentre le dinamiche del suo ambiente da un Fully observable MDP. L'agente quindi non conosce a priori i reward (R) e P, quello che può solo fare è cercare di predirli in base alle sue possibilità, che sono determinate dalla relazione tra i suoi diversi MDP. In questi due diversi MDP, gli stati possono non corrispondere, e non sempre il Partially observable MDP segue la markov property; ad esempio, se un giorno una persona compie una azione che produce un effetto a farfalla dall'altra parte del mondo e poi prende un colpo, non ricorda più niente, il giorno dopo può

subire le conseguenze di quello che ha fatto. Invece o se non avesse fatto quella azione quel giorno, avrebbe influenzato diversamente il futuro, quindi le future transizioni.

2.5 Policy

Ricordando che un agente quando riceve uno stato, in base alla sua esperienza, sceglie una azione da compiere, ora capiamo come questo processo avviene. La strategia/esperienza di un agente nel reinforcement learning è determinata da una policy, che può essere stocastica ($\pi : A \times S \to [0,1]$) o deterministica ($\pi : S \to A$), dove S ed A sono l'observation space e l'action space dell'MDP dell'agente.

La policy stocastica è definita generalmente come $\pi(a|s) = p(a|s)$ oppure (notazione alternativa) $\pi(a,s) = p(a|s)$. La probabilità data dalla policy stocastica, dato uno stato e una azione, è la probabilità che quella azione sia la migliore secondo l'agente.

La policy deterministica invece sceglie in modo deterministico la azione migliore dato uno stato, ed è definita come $a_t = \pi(s_t)$

La azione scelta dalla policy sarà successivamente compiuta. Per generalizzare, solitamente nelle formule si usa la policy stocastica.

2.6 Valutazione della policy

Premessa: In una serie di transizioni infinita, in un tempo t si può definire una variabile chiamata return (o meglio discounted return), con simbolo G (si usa anche R, ma può creare confusione con la matrice dei reward), definita come $G = \sum_{t=0}^{\infty} \gamma^t r_t$, se invece lo stato attuale è l'ultimo dell'episodio G vale 0, perchè è inutile considerare reward futuri che non saranno mai ottenuti.

Il return rappresenta una somma dei futuri reward, dove viene dato più peso a quelli immediati. Il discount factor (gamma), tale che $\gamma \in [0,1]$, minore è, maggiore i reward immediati influiscono sul discounted return.

Data questa premessa, possiamo definire la policy migliore (π^*), come quella policy che al tempo $t = 0$, massimizza il valore atteso del return G, oppure come quella policy che se usata, massimizza la media di tutti i reward totali su un gran numero di episodi. Formalmente:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, E_{s_0 \backsim p_0(\cdot), a_t \backsim \pi(\cdot|s_t), s_{t+1} \backsim p(\cdot|s_t,a_t)}(G) \qquad con \quad p(\cdot|s,a) = P_{s,a,\cdot}$$

Il punto si usa solo per convenzione, ad esempio $p_0(\cdot)$ nell'equazione è la distribuzione discreta di probabilità che ogni stato sia selezionato come primo, un'altra notaziona usata è $p(s_0)$.

Un'altro esempio: $p(\cdot|s_t, a_t)$ è la distribuzione discreta di probabilità che ogni stato sia selezionato come successivo, dato lo stato attuale e la azione scelta.

La formula può apparire complicata, e lo è, ma analizziamola: $E(G)$ non si riferisce ad una serie di transizioni, ma si riferisce a delle possibili transizioni nell'MDP dell'agente, infatti il valore atteso del return è calcolato con la variabili aleatorie discrete s_0, a_t e s_{t+1}. La prima è il primo stato della sommatoria del return, esso ha una distribuzione di probabilità data dalla probabilità che s_0 sia lo stato della prima transizione, bisogna stare attenti al fatto che per $t = 0$ si intende il tempo iniziale del calcolo del return, non il tempo dell'episodio. Poi c'è la variabile aleatoria a_t, che ha una distribuzione ottenuta dalla policy dato lo stato sempre allo stesso tempo t. Inoltre l'ultima variabile aleatoria è s_{t+1}, che ha una distribuzione di probabilità data dalla probabilità che tale stato al tempo $t+1$ sia stato selezionato dall'ambiente, dato lo stato al tempo t, e la azione presa a_t.

Nel caso di un fully observable MDP, gli algoritmi/ottimizzatori cercano di risolvere quest'equazione, invece nei partially observable MDP, dato che le probabilità delle transizioni e i reward sono sconosciuti inizialmente, e dato che la policy migliore dipende dai reward, la policy migliore può cambiare, poichè con il tempo la predizione dei reward varia, in altre parole, la policy migliore al tempo t, non sarà la migliore a tempo $t+1$, infatti possiamo distinguere la policy migliore secondo l'agente e la policy migliore secondo l'ambiente (quella "realmente migliore"). L'obiettivo è con il tempo avvicinare la policy migliore dell'agente a quella dell'ambiente.

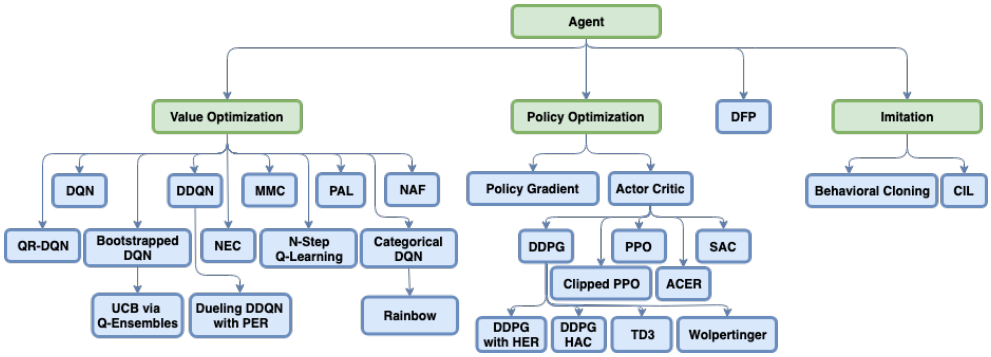
2.6.1 Differenza tra algoritmi on-policy e off-policy

Negli algoritmi di tipo on-policy, nel training viene usata la stessa policy dell'evaluation, invece negli algoritmi di tipo off-policy, solo nell'evaluation viene usata la policy migliore, nella sezione del Q-learning si capirà perchè.

La policy migliore è sempre deterministica, invece una policy non migliore può essere sia stocastica che deterministica, questo concetto sarà chiarito nella sezione 3.1

Se qualcosa non è chiaro, consiglio di continuare la lettura del Q-learning, e magari ritornare a leggere questa sezione successivamente.

Ci sono molti tipi di algoritmi di reinforcement learning, con approcci diversi, ad esempio si distinguono algoritmi di tipo value optimization e policy optimization, non andremo nel dettaglio, ma per avere un'idea basti osservare la figura sottostante.



2.7 State-value e action-value

Lo state-value $V_\pi(s)$, è una funzione che dice quanto uno stato è buono, in pratica è il return atteso seguendo la policy π, partendo da quello stato, formalmente:

$$V_\pi(s) = E_{a_t \backsim \pi(\cdot|s_t), s_{t+1} \backsim p(\cdot|s_t,a_t)}(G|s_0 = s)$$

L'action value $Q_\pi(s,a)$, chiamato anche Q-value, è un'altra funzione, come lo state-value, solo che deve essere specificata la prima azione scelta, quindi è il return atteso seguendo la policy π, partendo da uno determinato stato e scegliendo inizialmente una determinata azione.

$$Q_\pi(s,a) = E_{a_t \backsim \pi(\cdot|s_t), s_{t+1} \backsim p(\cdot|s_t,a_t)}(G|s_0 = s, a_0 = a)$$

2.8 Valutazione del Q-value e Bellman equation

La Bellman equation riesce ad esprimere il Q-value in modo ricorsivo:

$$Q_\pi(s,a) = E_{a' \backsim \pi(\cdot|s), s' \backsim p(\cdot|s,a)}(E(R_{s,a,s'}) + \gamma Q_\pi(s',a'))$$

Considerando la policy migliore, (ricordo che se è migliore è deterministica), dato uno stato, otteniamo l'action che massimizza il Q-value, possiamo dire:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q_{\pi^*}(s,a)$$

Ora possiamo definire il Q-value migliore (quello che segue la policy migliore), con la Bellman equation (in questo caso chiamata Bellman equation per l'ottimalità):

$$Q_{\pi^*}(s,a) = E_{s' \backsim p(\cdot|s,a)}(E(R_{s,a,s'}) + \gamma Q_{\pi^*}(s', \pi^*(s')))$$

$$Q_{\pi^*}(s,a) = E_{s' \backsim p(\cdot|s,a)}(E(R_{s,a,s'}) + \gamma \max_{a'} Q_{\pi^*}(s',a'))$$

Sarà proprio questa formula che ci permetterà nella prossima sezione di ottimizzare l'agente.

3 Q-Learning

Il Q-learning è un algoritmo di reinforcement learning, di tipo model-free, e di tipo value optimization. Esso si basa sul Q-value e sulla Bellman-equation per l'ottimizzazione. Le limitazioni sono che observation space e action space sono discreti e la policy è deterministica.

L'agente ha una tabella bidimensionale chiamata Q-table, dove ogni riga corrisponde ad uno stato nell'observation space, e ogni colonna ad una azione nell'action space, questa tabella ha per ogni stato e azione, il suo Q-value, da questo momento in poi daremo per scontato si intenda il Q-value che segue la policy migliore, quindi varrà la seguente convenzione: $Q_{s,a} = Q_{\pi^*}(s,a)$

Come già accennato, la policy migliore (che ricordo essere sempre deterministica), dato uno stato, sceglie la azione con il Q-value più alto: $\pi^*(s) = \text{argmax}_a Q_{s,a}$

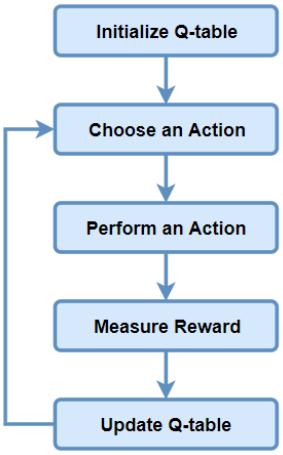
Se riusciamo a raccogliere i giusti Q-value, abbiamo ottenuto la policy migliore! Ma come calcoliamo i Q-values? Se riosserviamo la Bellman equation per l'ottimalità (che usa la stessa policy), notiamo che in un tempo t, se conosciamo il Q-value al tempo $t-1$, possiamo approssimare il Q-value al tempo t. Dico approssimare per 3 motivi: i reward possono variare, anche con stesso s, a, e s', quindi se in una stessa transizione ora ottengo un reward, dopo se rifaccio la stessa transizione potrei ottenerne un'altro, inoltre anche s' può non essere fisso se P è di tipo stocastico, infine anche il Q-value al tempo $t-1$ è anch'esso approssimato.

Però se proseguendo negli episodi aggiorniamo sempre i Q-values, per un tempo che tende all'infinito riusciamo a ottenere dei Q-values medi (o attesi), approssimando la policy migliore secondo l'agente con quella secondo l'ambiente (o quella realmente migliore), che è proprio quello che cerchiamo, l'aggiornamento avviene nel seguente modo:

$$Q_{s,a} \leftarrow Q_{s,a} + \alpha(r + \max_{a'} Q_{s',a'} - Q_{s,a})$$

Capiamo meglio questo processo: $r + \max_{a'} Q_{s',a'}$ è il Q-value atteso (o target) di s e a, mentre $Q_{s,a}$ è quello attuale, calcoliamo la loro differenza ed otteniamo un errore, che andiamo a sommare al Q-value attuale, dopo averlo moltiplicato per un fattore alpha, tale che $a \in]0,1]$, chiamato learning rate, se è vicino a 1, il Q-value nuovo è vicino a quello target, invece se è vicino a 0, il Q-value nuovo è vicino a quello attuale.

Il workflow di base del Q-learning è il seguente:



3.1 Exploration vs exploitation dilemma

Immagina di essere in una stanza con 2 porte, su una sai che dietro ci saranno 100 monete, invece non sai cosa ci può essere dietro l'altra; possono esserci più di 100 monete, come meno, la domanda è: conviene tentare la sorte o accettare le 100 monete sicure? Oppure un altro esempio: per andare a scuola uno studente percorre sempre la stessa strada, gli conviene provare delle stradine che non conosce, magari scoprendo che accorciano, o al peggio allungano il percorso? Sicuramente è svantaggioso ogni giorno provare stradine nuove, rischiando di allungare troppo il percorso, ma neanche usare sempre il percorso migliore che si conosce, una soluzione può essere di usare il percorso migliore, e ogni tanto provare qualche nuova strada, ma ogni quanto?

Questo è un grande dilemma nel reinforcement learning, e si basa sulla scelta tra exploration (esplorazione di nuove strade), ed exploitation (vado sulle strade migliori che già conosco), infatti non sempre la azione scelta dalla policy migliore, sarà quella compiuta, detto questo possiamo identificare la policy reale (precedentemente chiamata "non migliore"), che è quella che sceglie le azioni che realmente verranno compiute, essa si basa sulla policy migliore per l'exploitation (anche se a volte si può evitare l'exploitation nel training), e su un fattore casuale (solitamente) per l'exploration. In altre parole, data la azione scelta dalla policy migliore, dobbiamo capire se sarà quella ad essere compiuta o meno. Per fare questa scelta ci sono molti algoritmi, chiamati Bandit Algorithms, i più importanti sono i seguenti:

- epsilon-greedy (ε -greedy): La epsilon è un valore costante compreso tra 0 e 1 compresi, che è la probabilità che una azione casuale sia compiuta, il pseudocodice è:

  ```
  if random ∈ [0,1] < epsilon: action = random ∈ A
  else: action = π(state)
  ```

  La scelta della costante epsilon si basa su quanto l'MDP è deterministico, più lo è, meno deve essere esplorato, e va bene un epsilon piccolo, inoltre incide anche la durata degli episodi: se

tendono a finire in fretta, quindi alla minima azione sbagliata, è meglio preferire un epsilon basso.

- Softmax: Questo è un algoritmo di tipo "Value Adaptive Epsilon algorithm", cioè su basa sui Q-values per determinare la probabilità di scelta delle azioni.

  Questo algoritmo sceglie la azione basata su una distribuzione di probabilità calcolata in base ai return delle azioni, maggiore è il return, maggiore è la probabilità che la sua azione sia scelta. Ogni probabilità della distribuzione è ottenuta da:

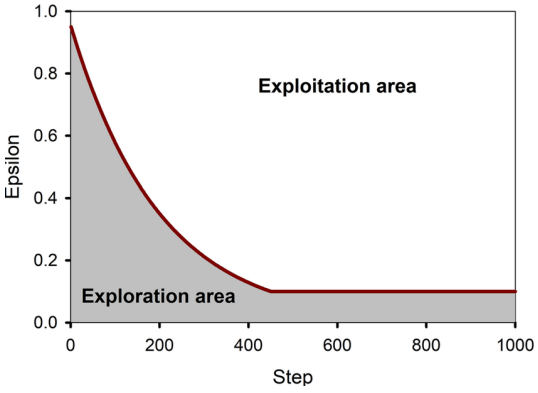
  $$p(a_0|s) = \frac{e^{Q_{s,a_0}/\tau}}{\sum_a^A e^{Q_{s,a}/\tau}}$$

  Il parametro τ, che deve essere sempre maggiore di 0; se è alto, la differenza di probabilità tra azioni con return diversi è minore, invece se è vicino a 0, viene fatta una scelta "greedy", cioè scegliendo sempre la azione migliore.

- Decay epsilon-greedy (basato su ε -greedy): Questo algoritmo, ad ogni step, decrementa l'epsilon in modo esponenziale, nel seguente modo:

```
epsilon = epsilon * epsilon_decay
if epsilon < epsilon_min: epsilon = epsilon_min
```

  dove epsilon decay $\in [0, 1]$



3.2 Esempio in python

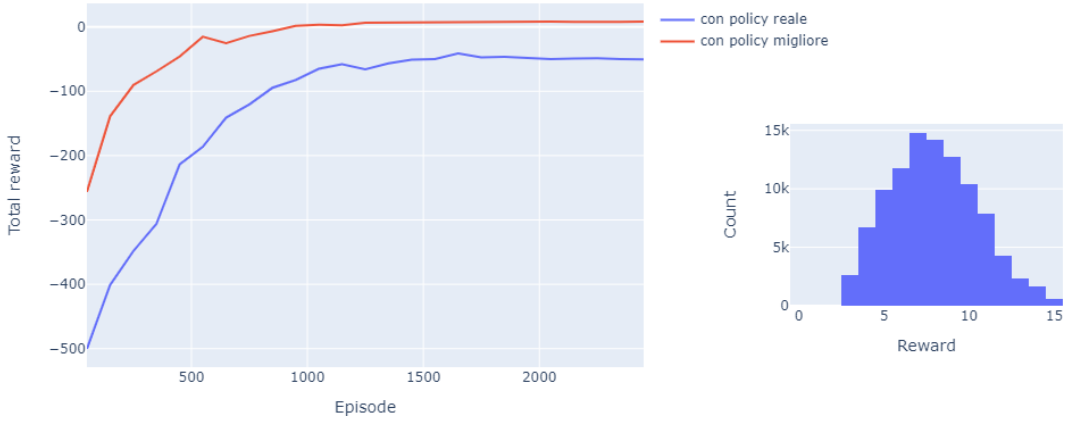
Ora diamo una occhiata ad un piccolo esempio in python, che andremo a spiegare. Nel progetto allegato è presente un notebook Python chiamato "q-learning-example.py", diviso in 2 parti. Analizziamo la prima:

Importiamo le librerie che useremo: numpy per varie operazioni matematiche, gym, per creare environments (ambienti) con interfaccia comune in modo facile e veloce, e plotly, per mostrare grafici. Viene definita la classe QAgent, dove nel costruttore viene passato l'environment, e viene inizializzata la Q-table con zeri. Viene definita la funzione start episode, dove, in un loop che dura fin

quando l'episodio è terminato, sceglie la azione con la policy reale (non migliore), con l'algoritmo epsilon-greedy. Ora, l'agente compie la azione, ed ottiene il reward, il prossimo state, e una variabile booleana, che indica se l'episodio è terminato; in caso affermativo, nell'aggiornamento non si tiene conto del Q-value dello state successivo, perchè non avrebbe senso pensare di ottenere reward futuri, se l'episodio è terminato.

Dopo aver definito l'agente, nella seconda parte andremo ad eseguire un piccolo programma di esempio. Il discount facotor usato sarà pari a 0.99 e il learing rate 0.1. Notare che il valore di epsilon nel testing è pari a 0, in modo che la policy reale corrisponda a quella migliore. Nel primo grafico sarà visualizzato l'andamento del total reward nei vari episodi, seguendo la policy migliore e quella reale. Infine l'ultimo grafico mostra la distribuzione del reward totale seguendo la policy migliore.

L'output dello script è il seguente:



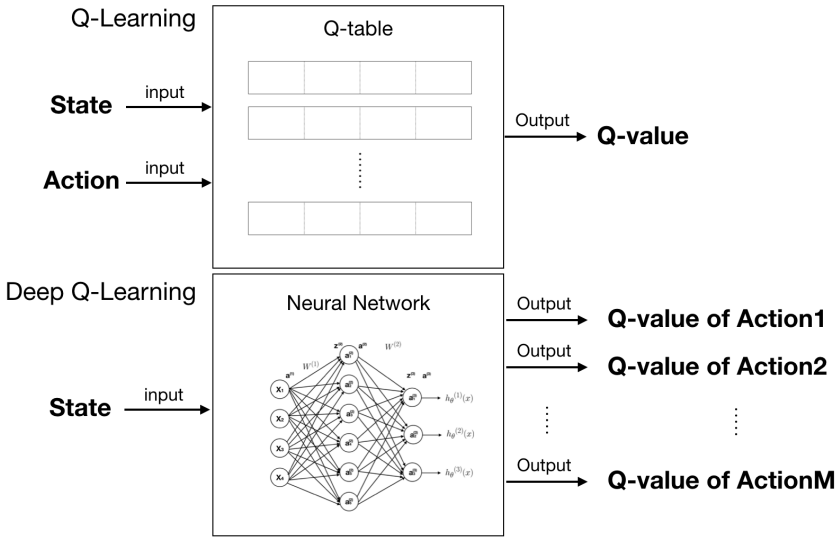
media total reward: 7.92, deviazione standard total reward: 2.58

3.3 Conclusioni

Il Q-learning è un algoritmo molto facile, e i parametri (discount factor, learning rate, ecc), hanno un relativamente buono margine di errore, le limitazioni però si vedono quando aumenta la complessità dell'MDP dell'agente.

4 Deep Q-Learning

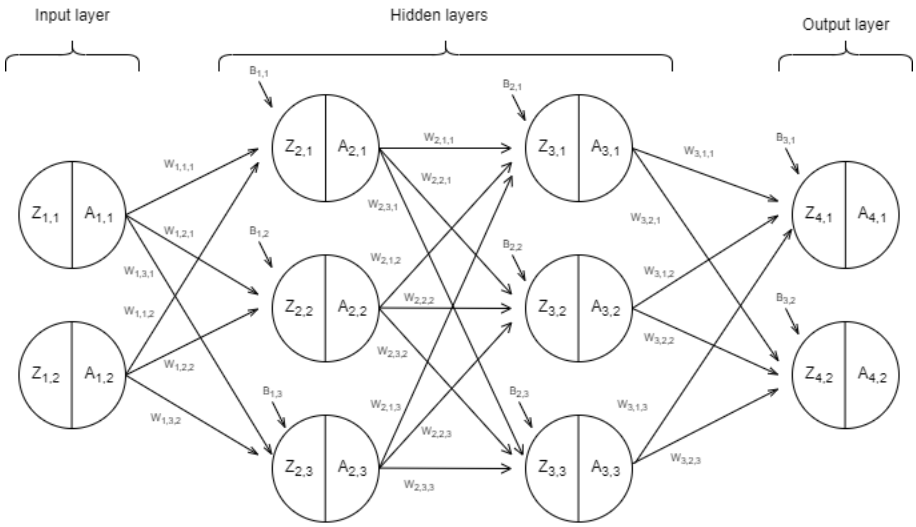
Una grande limitazione del Q-learning è l'observation space discreto, in quanto i Q-values sono ottenuti mediante l'accesso ad una matrice. La soluzione adottata dal deep Q-learning è quella di rimpiazzare la Q-table con una funzione che approssima i Q-values, e questa funzione è una rete neurale, dove dato uno stato, approssima/predice i Q-values di tutte le possibili azioni. Il deep Q-learning è quindi un algoritmo di deep reinforcement learning, parola che deriva da reinforcement learning, e deep learning.



La funzione che calcola i Q-values seguirà la seguente convenzione: $(Q_W(s))_a = Q_{\pi^*}(s,a)$, dove W sono i parametri della rete neurale.

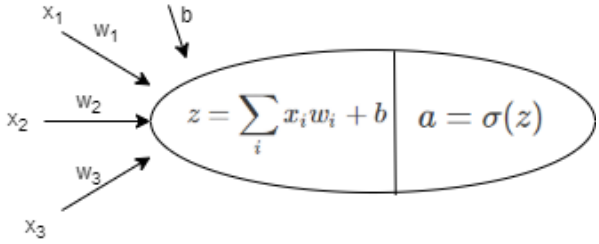
4.1 Reti neurali

Una rete neurale (oppure neural network, o anche deep neural network) è un approssimatore di funzioni, che prende ispirazione dalle reti neurali biologiche, può essere strutturata in molti modi, di seguito vedremo un semplice esempio.



Ogni rete neurale ha un input layer che contiene gli input, degli hidden layers dove gli input vengono elaborati, e un output layer che contiene gli output della rete neurale. I parametri W (o anche θ) sono chiamati pesi, e assieme ai bias B, sono quelli che parametrizzano la predizione della rete neurale. W è semplicemente una matrice tridimensionale, dove la prima dimensione identifica il layer, la seconda il neurone successivo, e la terza il neurone precedente, pensando ad una funzione lineare, W rappresenta i coefficienti angolari e B il termine noto. Gli input della rete neurale corrispondono ai valori Z_1, mentre gli output sono i valori A_L, dove L è il numero di layer totali. I valori z e a di ogni

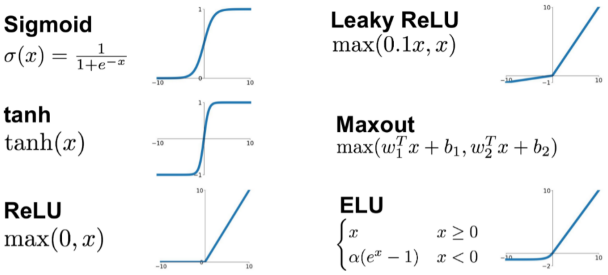
neurone (chiamati anche valore di entrata e di uscita), tranne quelli nell'input layer, sono calcolati nel seguente modo che riassumo in uno schema, con input un vettore x:



in generale:

$$Z_{l,k} = \sum_{A_{l-1,k}}^{n} a_k b_k \qquad A_{l,k} = \sigma(Z_{l,k})$$

dove σ è una funzione chiamata funzione di attivazione, che ha principalmente il compito di preparare il valore a ad una prossima sommatoria, ad esempio cercando di evitare valori troppo grandi, ma ha anche altri scopi che non approfondiremo. Ci sono molti tipi di funzioni di attivazione, che possono variare rispetto ai layer o anche rispetto ai singoli neuroni di un solo layer, le funzioni di attivazione più usate sono la sigmoide, la ReLU, leacky ReLU, funzione lineare, ecc. Nel progetto del cartpole ho usato la leacky ReLU per gli hidden layers e la funzione lineare per l'input e output layer.



Quanti neuroni nell' hidden layer deve avere una rete neurale, o quanti hidden layers deve avere? Entrambe le cose variano da problema a problema, possiamo dire però che per approssimare funzioni complesse abbiamo bisogno di più layers, anche se si rischia di avere un overfitting della rete neurale. Invece, per quanto riguarda il numero di neuroni, ci sono alcune regole empiriche, come quella che dice che il giusto numero sta alla metà tra il numero di neuroni dell'input layer e il numero di neuroni dell'output layer, ma in realtà bisogna andare per tentativi molte volte.

4.2 Ottimizzazione della rete neurale

Premessa: Da questo momento il bias B lo intenderemo come parte dei peso W, quindi è possibile pensare ad un neurone extra per ogni layer (tranne quello di output), con valore di uscita pari ad 1 e collegato solamente con i neuroni successivi.

Per ottimizzare la rete neurale in modo che predica correttamente tutti i Q-values target, in base ad uno stato, calcoliamo i Q-values delle sue actions, poi tramite la policy, troviamo l'action da compiere.

Dopo averla eseguita, tramite la Bellman equation, troviamo il Q-value target, lo rimpiazziamo ai Q-values già calcolati al posto dell'action scelta, e ottimizziamo la rete neurale. Il pseudocodice è:

```
q_values_predicted = Q_W(s)
action = π(q_values_predicted)
next_state, reward = execute(action)
q_values_target = copy(q_values_predicted)
q_values_target[action] = reward + max(h(next_state))
train(q_values_predicted, q_values_target)
```

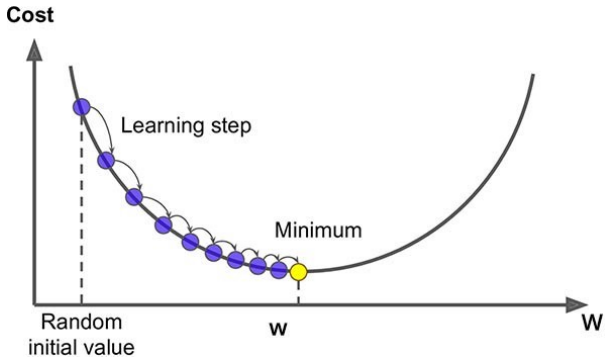
Per l'ottimizzazione vera e propria rete neurale, dobbiamo identificare una funzione da minimizzare, questa è la cost function (o anche loss function), solitamente denonimata con la lettera J o C, che come parametri ha un vettore di input s, e un vettore di output target q^*, il suo scopo è di dire quanto sono distanti tra loro la predizione dell'input e il target, ce sono di molti tipi, la più comune che ho usato anch'io è la MSE (errore quadratico medio), definita come $J(s,q^*,W) = \frac{1}{2}\sum(Q_W(s) - q^*)^2$

Il nostro obiettivo è minimizzare questa funzione variando W, quindi:

$$W^* = \underset{W}{\operatorname{argmin}} J(s,q^*,W)$$

Da notare che per ottenere l'errore quadratico medio non bisognerebbe dividere il tutto per 2, ma per la lunghezza di A, ma dato che il nostro obiettivo è minimizzare questa funzione, il risultato sarà lo stesso, e dividere tutto per 2, renderà più facile il calcolo della derivata parziale che andremo a vedere di seguito. Per risolvere questo problema di ottimizzazione, usiamo un algoritmo chiamato gradient descent. Inizialmente i pesi vengono inizializzati con dei valori casuali (ci sono diverse distribuzioni che si possono seguire), poi in base alle derivate parziali della cost function , rispetto ad ogni peso (la derivata corrisponde all'errore del peso), quindi $\frac{\partial J(s,q^*,W)}{\partial W_{l,k,j}}$, varia i pesi in modo da minimizzare la cost function. Ad esempio se la derivata è negativa, vuol dire che un valore del peso appena maggiore diminuirà la cost function. L'ottimizzazione avviene quindi nel seguente modo: $W_{l,k,j} \leftarrow W_{l,k,j} - \alpha \frac{\partial J(s,q^*,W)}{\partial W_{l,k,j}}$

Dove α è il learning rate, se troppo basso l'ottimizzazione è lenta, se troppo alto, i pesi divergono, quindi per trovare un learning rate giusto è meglio partire da uno basso, a aumentare gradualmente.



Come calcoliamo la derivate parziali per ogni peso? Un modo efficiente è usare l’algoritmo backpropagation, propagando l’errore nella rete neurale dall’output layer all’input layer, sfruttando la chain rule, che permette di calcolare la derivata di funzioni composte.

Nelle prossime sezioni vedremo 3 ottimizzazioni che ho implementato e usato nel progetto allegato, e successivamente ci sarà un esempio in pseudocodice.

4.3 Gradient descent con momentum

Un rischio di gradient descent è di bloccare l’ottimizzazione ad un minimo locale, nonostante esistano molti altri algoritmi alternativi a questo più efficienti; per semplicità ho usato "gradient descent con momentum", che riesce ad evitare alcuni minimi locali e a velocizzare il processo di ottimizzazione. Si basa sull’aggiornare ogni peso non in base alla sua derivata, ma in base alla media pesata delle sue derivate calcolate nel corso del tempo, dando più peso a quelle più recenti (il "quanto" più peso è regolato da un parametro chiamato momentum).

4.4 Replay memory

Ad ogni step aggiornare i pesi della rete neurale può variare troppo velocemente le predizioni della rete, e quindi peggiorarle, in quanto ogni aggiornamento, dato uno stato, influenzerà anche la predizione dei Q-values di tutti gli altri stati. Una soluzione è memorizzare le istruzioni per ottimizzare la rete (quindi state, action, reward, done e next state) in una memoria chiamata replay memory (o anche experience replay), e per ogni step aggiornare la rete, seguendo un numero di istruzioni estratte casualmente dalla replay memory (questo numero è chiamato batch-size). In generale, l’aggiornare la rete neurale con "gradient descent con momentum" con dei mini-batch (un mini-batch è una serie di istruzioni prese dalla replay memory), prende il nome di "mini-batch gradient descent con momentum".

Una limitazione dell’experience replay è il dare importanza uguale a predizioni che magari sono secondarie, con predizioni che magari è fondamentale che l’agente impari; una ottimizzazione possibile è chiamata "prioritized experience replay", che non approfondiremo.

4.5 Target network

Per rendere l’ottimizzazione più stabile, bisogna usare una target network, una rete neurale strutturalmente identica a quella principale, con la differenza che nell’aggiornamento di un mini-batch, la predizione dei Q-values sucessivi (dai quali si derivano i target Q-values) viene fatta dalla target network, e i pesi vengono ottimizzati solo nella rete neurale principale, ed in ogni tot step i pesi della target network si sincronizzano con quelli della rete neurale pincipale.

4.6 Esempio di un agente in pseudocodice

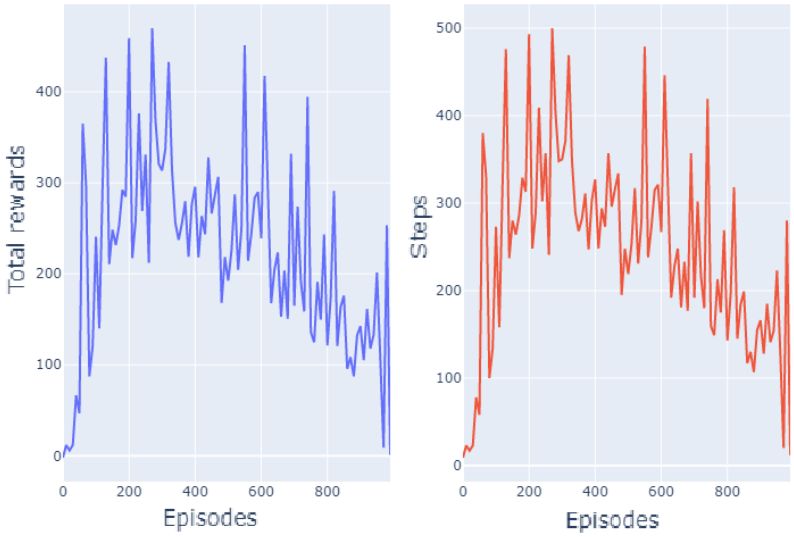
Ora vediamo un esempio di funzione in pseudocodice di un agente DQN, cioè deep Q network, che esegue un episodio. Il bandit algorithm usato è epsilon decay, per l’ottimizzazione è usato l’algoritmo mini-batch gradient descent con momentum (la backpropagation vera e propria viene fatta nella funzione "train" della rete neurale, tutto il codice completo anche quindi di rete neurale è nel progetto allegato.

```
state = get_first_state()
while not done:
    # policy
    if random ∈ [0,1] < epsilon: action = random_action
    else: action = argmax(nn.predict(state))
   next_state, reward, done = step(action) # compio la azione
   replay_memory.put(state, action, reward, done, next_state) # memorizzo nella replay memory
   if len(replay_memory) >= batch_size: # se c'è abbastanza memoria
      # ottengo un mini-batch dalla replay memory
      for state_exp, action_exp, reward_exp,
          done_exp, next_state_exp in replay_memory.get_random(batch_size):
         q_values = nn.predict(state_exp) # calcolo i Q-values
         q_values_target = copy(q_values)
         if done: q_values_target[action_exp] = reward_exp
         else: q_values_target[action_exp] =
             # calcolo dei target Q-values con la target network
             reward_exp + discount_factor * max(target_nn.predict(next_state_exp))
         nn.train(state_exp, q_values_target, learning_rate, momentum) # ottimizzazione
      if steps  mod  steps_to_sync_target_nn == 0:
         self._sync_target_nn_weights() # sincronizzo la target nn con la principale nn
      epsilon *= epsilon_decay # epsilon-decay algorithm
      if epsilon < min_epsilon: epsilon = min_epsilon
   state = next_state # imposto il nuovo current state
self._sync_target_nn_weights() # sincronizzo la target nn con la principale nn anche alla fine
```

4.7 applicazione con il cartpole

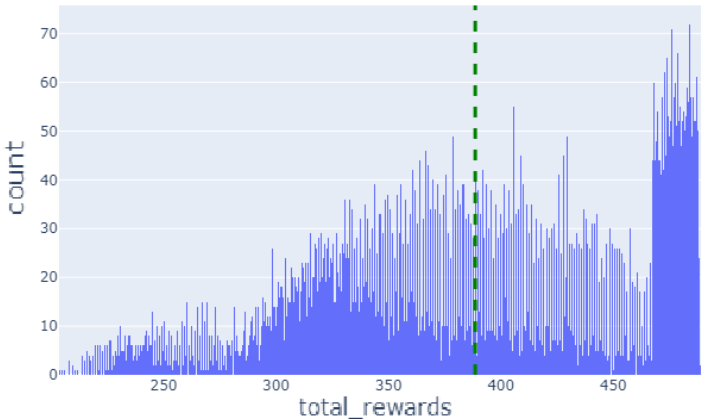
Nella applicazione allegata, ho fatto un piccolo progetto di deep Q-learning con il cartpole, ho usato i seguenti parametri: replay memory max size = 10000, batch size = 30, discount factor = 0.99, learning rate = 0.0001, number of steps to sync target network = 10, epsilon decay = 0.99, min epsilon = 0.01, momentum = 0.4, per la rete neurale ho usato 4 layer totali, con rispettivamente 4, 50, 100 e 2 neuroni, la activation function nel primo neurone e nell’ultimo è lineare, mentre per gli hidden layer ho usato la leacky ReLU, infine per l’inizializzazione dei pesi ho usato una distribuzione uniforme con minimo -0.5 e massimo 0.5.

Su 1000 episodi di training totali ho notato che l'agente ha ottenuto il massimo dei reward intorno all'episodio 320, quindi mi sono salvato su un file i pesi a quell'episodio e li ho usati per valutare i risultati.



Si può notare che dopo circa l'episodio 320, il total reward comincia a scendere; questo fenomeno è chiamato "Catastrophic interference", fenomeno tipico delle reti neurali usate nel reinforcement learning, causato dal fatto che nuove ottimizzazioni possono peggiorare quelle fatte in precedenza, quindi dopo aver inserito ottimizzazioni avanzate temporalmente nei vari episodi, è possibile che la rete neurale non riesca più a predire correttamente i durante i primi step degli episodi. Questo problema può essere aggirato nell'offline learning (dove il training è separato dall'evaluation, molto simile ad un off-policy learning), che è quello che ho fatto, ma è grave nell'online learning (nessuna differenza tra training ed evaluation), che è la base di un'intelligenza artificiale generale.

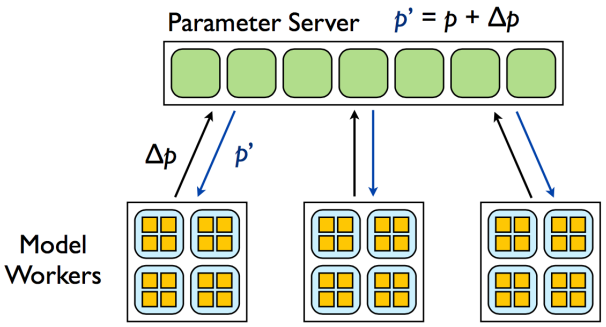
Per l'evaluation (quindi la valutazione dei risultati) ho ovviamente usato la policy migliore. Di seguito ho calcolato la distribuzione dei reward totali su un grande numero di episodi, dove si può notare una specie di gaussiana approssimata, ma anche una tendenza a e finire l'episodio tra i 475 e i 500 total reward, questo dipende fortemente dall'environment. Da notare però che il total reward non supera mai i 500, perchè è il massimo raggiungibile in questo environment, dopodiché l'episodio finisce, la linea verde tratteggiata è la media (388.6).



5 Deep Q-learning distribuito

Il corso "CME 323: Distributed Algorithms and Optimization" di Stanford nel 2015 ha proposto un algoritmo di Deep Q-learning distribuito (`https://stanford.edu/~rezab/classes/cme323/S15/projects/deep_Qlearning_report.pdf`), basato sull’algoritmo Downpour SGD, una variante asincrona del gradient descent.

Questo algoritmo definisce un server, e dei workers, rappresentati nella figura sottostante:



dove p sono i pesi.

5.1 Workers

Ogni worker lavora in modo asincrono e indipendente dagli altri, e ha il compito di richiedere al server i pesi più aggiornati, mantenendo localmente una target network, di generare dati da inserire nella replay memory, anch’essa mantenuta localmente, di calcolare i gradienti con dei mini-batch, e di mandarli alla fine del mini-batch al server.

5.2 Server

Il server mantiene dei pesi della rete neurale localmente, ha il compito di fornirli ai workers quando lo richiedono, e anche di aggiornarli con i gradienti che ricevono.

5.3 Scalabilità

La scalabilità inizialmente è orizzontale, aggiungendo workers, e questo oltre a velocizzare l’apprendimento, aumenta anche la stabilità. Uno dei rischi è che il server possa essere troppo lento a rispondere ai workers: una soluzione può essere aumentare il batch-size, e quindi rendere la comunicazione dei gradienti meno frequente, oppure scalare verticalmente il server, quindi potenziargli l’hardware o aumentargli la banda. Un altro rischio è che, data la grande frequenza di aggiornamento dei pesi nel server, i workers possano usare un modello obsoleto, rallentando l’apprendimento, e una soluzione può essere usare degli ottimizzatori chiamati "adaptive learning rate methods", come RMSprop o AdaGrad.

Istituto Tecnico Tecnologico “E. Barsanti”
Via dei Carpani 19/B – 31033 – Castelfranco V. (TV)
Tel. 0423 – 492847 – 493614
email:info@barsanti.edu.it

Fonti

[1] http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf.

[2] https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[3] https://en.wikipedia.org/wiki/Reinforcement_learning.

[4] https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c.

[5] https://towardsdatascience.com/reinforcement-learning-concept-on-cart-pole-with-dqn-799105ca670.

[6] https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c.

[7] https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c.

[8] https://www.allaboutcircuits.com/technical-articles/how-many-hidden-layers-and-hidden-nodes-does-a-neural-network-need.

[9] https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092.

[10] https://stanford.edu/~rezab/classes/cme323/S15/projects/deep_Qlearning_report.pdf.

[11] https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example.

[12] https://intellabs.github.io/coach/components/agents/index.html.

[13] Mohit Sewak. *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*. 2019.

IL DIRIGENTE SCOLASTICO

Francesco Daniele Laterza