**Applicazione di Reinforcement Learning su un**

**Platform Game in Python**

**Introduzione**

L’obiettivo del progetto è stato quello di sviluppare ed allenare un agente intelligente, con il compito di apprendere un metodo per completare un “livello” di un *platform game,* utilizzando il *reinforcement learning (RL),* che è una branca del *machine learning* in cui un agente viene addestrato tramite tecniche di ottimizzazione per raggiungere la soluzione del problema. Analizzando l’ambiente e adottando un comportamento di *“try, miss, repeat”*, l’agente cerca di imparare tramite l’uso di *reward* con l’obiettivo di massimizzarle, seguendo gli stati che producono guadagno maggiore ed evitando quelli che ne provocano una perdita. Le reward possono quindi essere positive in caso di comportamenti corretti o negativi altrimenti e vengono assegnate all’agente dopo ogni azione, in modo che esso possa utilizzarle come parametro di apprendimento. Il percorso intrapreso dall’agente è composto da azioni scelte (*exploitation)* e azioni randomiche (*exploration)* ed è necessario, per il giusto sviluppo dell’agente, che sia presente un corretto rapporto tra questi due tipi di azioni per evitare *overfitting* o *underfitting*. Alla conclusione dell’esperimento, l’agente dovrebbe riuscire a completare l’intero livello basandosi soltanto su azioni e stati appresi durante il training; Affinché questo avvenga, l’agente deve prima esplorare numerosi stati tramite azioni casuali per poi selezionare quello migliore. Queste tecniche di *reinforcement* learning vengono applicate in campo di videogiochi, robotica e ottimizzazione di sistemi complessi, grazie alla loro versatilità ed efficienza. Esistono diversi algoritmi di supporto al *reinforcement learning,* tra cui *Q-Learning e Policy Gradient, che non richiedono l’utilizzo* una rete neurale e *Deep Q-Network, Proximal Policy Optimization* ed altri che invece sono basati su un approccio *Model-Based*.

**Gioco di partenza**

Il gioco da noi scelto per applicare il *reinforcement learning* è un platform game simile a Super Mario però creato da noi tramite l’uso delle librerie *pygame* e *g2d* [1],in particolar modo l’ultima, implementata dal Prof. Michele Tomaiuolo, facilita lo sviluppo di giochi in python attraverso numerosi metodi built-in.

L’idea di utilizzare un gioco implementato da noi ci ha dato la possibilità di familiarizzare con l’environment di *OpenAI Gym* e di affrontare il problema in un'ottica più generale, al contrario di quanto avremmo fatto utilizzando un gioco fatto apposta per il *reinforcement learning* come Atari 2006 games.

Le meccaniche di gioco prevedono movimenti su un piano 2D e interazioni con ostacoli quali: parti di terreno mancanti o piattaforme volanti. Lo spazio di mosse possibili è composto da movimenti orizzontali a destra e sinistra e da un movimento sul piano verticale attraverso un semplice salto. Nella versione del gioco sviluppata originariamente erano presenti anche monete e mostri, con cui il giocatore poteva interagire, ma in seguito sono stati rimossi per semplificare l’apprendimento dell’agente di RL. Obiettivo del giocatore è quello di completare il percorso evitando di cadere nel vuoto e di raggiungere la bandiera situata alla fine del livello.

**Metodologia di Reinforcement Learning**

Per quanto riguarda l’addestramento dell’agente di RL, si è scelto di utilizzare l’approccio di *Q-Learning* basato sull’algoritmo *epsilon-greedy* [2], con il vantaggio di non dover utilizzare una rete neurale e quindi la possibilità di avere un maggior controllo sui parametri di apprendimento. Questo approccio prevede il salvataggio degli stati imparati dall’agente in una q-table e l’utilizzo di quest’ultima per “exploitare” le azioni conosciute che portano a massimizzare la reward. L’algoritmo prevede la definizione di iperparametri per modificare e personalizzare l’apprendimento dell’agente: quali *Learning Rate*, *Discount Factor* e *Action Selection Factor*:

* *Gamma (0, 1)* corrisponde al Discount Factor, un parametro che specifica quanto vogliamo che l’agente dia priorità a reward immediate, se messo a 1 l’agente si focalizzerà sugli stati vicini ad eseguirà azioni che massimizzano la reward più prossima, mentre con un valore prossimo allo 0, l’agente sarà più lungimirante e valuterà maggiormente le ricompense future.
* Il *Learning Rate (0, 1)* è il parametro che controlla la convergenza ad una soluzione ottimale; è spesso associato alla discesa del gradiente poiché favorisce una stabilità all’algoritmo per valori tendenti a 0 oppure porta ad un minimo locale più rapidamente, ma con la possibilità di saltare soluzioni ottimali, per valori tendenti a 1.
* L’*epsilon (0, 1),* detto anche *Action Selection Factor*, bilancia il rateo di azioni di “exploration” o “exploitation” intraprese dall’agente, al contrario dei primi due questo parametro non influisce direttamente sul calcolo del percorso migliore, ma permette di bilanciare l’apprendimento in base alla tipologia di problema da affrontare.

Il calcolo dell’epsilon verrà affrontato più nel dettaglio nei prossimi capitoli, poiché si è dimostrato un fattore fondamentale nello sviluppo del progetto.

**OpenAI Gym Custom Environment**

Per lo sviluppo del progetto è stato necessario implementare un *environment* ad-hoc che interfacciasse il nostro gioco con la libreria *OpenAI Gym*, attraverso una standardizzazione delle funzioni di esecuzione e controllo.

L’environnement implementa le seguenti funzioni standard:

* ***step()*** : esegue un tick di gioco, applicando l’azione decisa dall’agente, acquisendo le informazioni dall’ambiente e ricavando la reward ottenuta.
* ***reset()***: inizializza una nuova partita quando l’agente inizia a giocare o dopo ogni sconfitta/vittoria.
* ***render()***: renderizza il gioco nel caso si voglia avere un riscontro visuale al progresso dell’agente.

Inoltre, all’interno del file di gioco, sono state implementate ulteriori funzioni di supporto al *custom environment*, in modo da poter interagire con il gioco senza l’utilizzo di strumenti esterni:

* ***observe()*** *:* restituire le informazioni percepite dall’agente durante la partita (nel nostro caso soltanto la propria posizione assoluta).
* ***evaluate()*** : restituisce la reward totale ottenuta dopo ogni tick di gioco, calcolata sulla base di una funzione di reward.
* ***action()*** : permette all’agente di interagire con il gioco attraverso tre movimenti (*move left, move right, jump*)

Modificando leggermente il gioco si è riusciti inoltre ad implementare una modalità di esecuzione senza render, ovviando al problema di rallentamento dell’applicazione dovuto alle operazioni di I/O, il che ha reso l’apprendimento dell’agente maggiormente rapido ed efficiente.

Una volta integrato l’ambiente, attraverso il file *\_\_init\_\_ (),* che ne definisce il costruttore, è stato possibile importarlo ed utilizzarlo attraverso il metodo *gym.make("SuperMarion-v0")* messo a disposizione dalla libreria*.*

Dopo aver concluso questo passaggio è stato possibile implementare il “cervello” dell’agente e procedere con il *training* vero e proprio.

**Implementazione della logica**

**Ricompense:**

La proporzione in cui vengono assegnate le ricompense e la causa di ogni assegnazione, è un parametro fondamentale per l’implementazione di un buon modello di Reinforcement Learning, per questo si sono sperimentate più possibilità durante lo sviluppo.

Le ricompense utilizzate si sono attribuite a specifiche *“milestones”* raggiunte dall’agente, tra queste: il raggiungimento della fine del percorso, segnalato dalla bandiera; l’esaurimento delle vite del personaggio, derivante dalla caduta nel vuoto o dall’incapacità di avanzare nel livello per un tempo prolungato (la reward assegnata in questi casi è negativa); l’avanzamento del personaggio durante il percorso ad ogni step. Si è provato inoltre a cambiare il modo in cui la reward viene assegnata durante l’avanzamento nel percorso: il primo metodo sperimentato è stato quello di ricompensare l’agente dopo un avanzamento di un pixel, il secondo invece, è stato quello assegnare una reward solo dopo un tot di spostamenti nella direzione giusta, ovvero ogni volta che l’agente supera un *checkpoint* virtuale.

**Parametro ‘ε’ dinamico:**

Il parametro ε, come accennato in precedenza, permette di controllare la proporzione di azioni casuali e mirate compiute dall’agente. A causa della natura del problema, si sono sperimentati vari approcci riportati di seguito.

Mantenendo il ε costante, in ogni punto dell’apprendimento l’agente riesce ad esplorare molteplici stati attorno allo stato corrente attraverso azioni casuali; Attraverso questo approccio però, l’agente sarà portato a eseguire sempre azioni randomiche senza provare a sfruttare gli stati salvati nella *q-table*.

Un approccio migliore è quello di decrementare ε man mano che l’agente prende confidenza con il percorso; Se si considera un'applicazione in cui, l'apprendimento di una strategia che massimizzi la *reward* negli stati iniziali permette di avanzare negli stati successivi, allora decrementare ε linearmente può essere una soluzione efficace per arrivare ad una soluzione ottimale.

Il fattore principale che limita l’utilizzo di questo approccio nella nostra applicazione è intrinseco alla natura del problema affrontato, ossia che nel caso preso in considerazione l’agente si basa solo sulla propria posizione attuale per scegliere un'azione e quindi ad ogni punto del percorso le informazioni accumulate negli stati precedenti non sono rilevanti per il raggiungimento degli stati successivi; Per questo si è deciso di adottare una strategia in cui ε viene incrementato e decrementato dinamicamente a seconda delle ricompense ottenute nella generazione precedente e quelle ottenute allo stato corrente. Ad ogni *step* l’agente incrementa la *reward* totale ottenuta nell'episodio corrente e alla fine di ogni episodio, il valore minimo della lista delle *reward* massime verrà sostituito con la *reward* totale dell'episodio, nel caso in cui il nuovo valore sia maggiore di quello già presente. In questo modo, alla fine di ogni generazione,la lista delle ricompense conterrà tutte le *reward* ottenute dall’agente durante gli episodi precedenti e, calcolando la media (al netto di *outliers*), si potrà così generare un *upper-bound* della ε da utilizzare nella generazione successiva. Ad ogni *step* di ogni episodio della generazione successiva, se la *reward* corrente non supera *l'upper*-*bound*, l’agente preferirà seguire la *q-table* (*exploitation*) mente se la *reward* supera *l’upper-bound,* l’agente preferirà eseguire azioni casuali (*exploration*).

epsilon\_decay = 0.999

epsilon\_min = 0.05

epsilon\_max = 1

gen\_epsilon = epsilon\_min

epi\_epsilon = epsilon\_max

epsilon = epsilon\_max

up\_reward\_mean = 0

for generation in range(MAX\_GENERATIONS):

# INITIALIZE GENERATION REWARD VECTOR

max\_reward = [0] \* MAX\_EPISODES

for episode in range(MAX\_EPISODES):

# INITIALIZE EPISODE REWARD

total\_reward = 0

\*\*\*RESET THE GAME AND GET THE STATE\*\*\*

while True:

# CHOOSE WITCH EPSILON TO USE ON THE CURRENT STEP

if total\_reward < up\_reward\_mean:

epsilon = gen\_epsilon

else:

epsilon = epi\_epsilon

# GET THE ACTION ACCORDING TO THE TABLE

q\_action = np.argmax(q\_table[state])

# DECIDE IF TAKE A RANDOM ACTION OR FOLLOW THE Q-TABLE

if q\_action != 0:

if random.uniform(0, 1) < epsilon:

action = env.action\_space.sample()

else:

action = q\_action

else:

action = env.action\_space.sample()

\*\*\*ACQUISITION OF REWARD AND STATE\*\*\*

# CHECK IF THE EPISODE IS FINISHED

episode\_done = terminated or truncated

# MEAN WITHOUT OUTLIERS

total\_reward += reward

\*\*\*POLICY OPTIMIZATION ROUTINE\*\*\*

if episode\_done:

# RANDOM ACTION RATE INCREMENT

if epi\_epsilon >= epsilon\_min and epi\_epsilon == epsilon:

epi\_epsilon \*= epsilon\_decay

# ADD THE TOTAL REWARD IF IT'S GREATER THAN ANOTHER

REGISTERED REWARD

if total\_reward - DEATH\_REWARD > min(max\_reward):

max\_reward.remove(min(max\_reward))

max\_reward.append(total\_reward - DEATH\_REWARD)

break

# MEAN WITHOUT OUTLIERS

reward\_mean = stats.trim\_mean(max\_reward, 0.1)

# AFTER EVERY GENERATION UPDATE THE EPSILON

gen\_epsilon = min(epi\_epsilon, gen\_epsilon)

epi\_epsilon = epsilon\_max

up\_reward\_mean = reward\_mean

**Azione successiva:**

Per ottenere l’azione successiva che l’agente deve compiere si è applicata la seguente formula basata sull’algoritmo *epsilon-greedy* descritto in precedenza:

*Q-table[state][action] = (1 − α) ⋅ q\_value + α ⋅ (reward + γ ⋅ best\_q)*

* *Q-table[state][action]* : È il valore associato alla coppia di stato e azione correnti nella q-table; La q-table memorizza un valore stimato per ciascuna possibile combinazione di stato e azione che rappresenta quanto sia conveniente intraprendere una determinata azione quando ci si trova in uno stato specifico.
* *alpha* : Rappresenta il *learning rate* discusso in precedenza.
* *q\_value* : Raffigura il precedente valore memorizzato nella Q-table per lo stato e l'azione corrente.
* *reward* : È la ricompensa ricevuta dall’agente dopo aver eseguito la specifica azione nell'ambiente e indica il riscontro immediato della decisione presa dall’agente.
* *gamma* : Rappresenta il *discount factor* discusso in precedenza.
* *best\_q* : Coincide con il valore massimo (migliore) associato allo stato successivo; Questa componente incoraggia l'agente a massimizzare il guadagno futuro, scegliendo azioni che portino ad un valore massimo futuro della q-table.

**Addestramento:**

L’iterazione di questi passaggi, per un numero di episodi e generazioni scelte arbitrariamente, compone il training dell’agente e permette di riempire le celle inizialmente vuote della q-table, in modo da generare una politica che l’agente possa seguire per risolvere il problema. La scelta del numero di iterazioni da compiere non è fondamentale per la riuscita dell’algoritmo, siccome tutti gli stati presenti in tabella e i valori di epsilon vengono salvati su file al completamento di ogni episodio e possono essere re-importati per la ripresa dell’addestramento dall’ultimo salvataggio, senza perdita di progressi. Se si volesse resettare l’allenamento dell’agente basterebbe così rimuovere i file di salvataggio, in modo che il sistema possa generare una nuova tabella vuota da riempire.

**Risultati**

Sono riportati in questo paragrafo i risultati ottenuti durante lo sviluppo del modello, i quali dipendono fortemente dalle metodologie di apprendimento adottate.

**Versione 1.0:**

Nella prima versione, per valutare il funzionamento dell’applicazione, si è utilizzato un agente senza capacità percettive, il quale poteva soltanto compiere azioni e ricevere reward; inoltre, le azioni random compiute dall’agente decrementano con l’aumentare del numero di epoche intraprese, a prescindere dal livello di apprendimento del percorso.

Si è osservato che questo tipo di agente era in grado di comprendere la direzione giusta di movimento, senza però riuscire a superare gli ostacoli.



**Versione 2.0:**

Un notevole miglioramento si è osservato dopo l’implementazione della percezione dell’ambiente nell'agente, ossia della sua posizione corrente. In questa versione l’agente è riuscito a superare i primi ostacoli e avanzare nel percorso. Al contrario della parte iniziale del training, dove l’agente riusciva ad imparare i movimenti giusti, in quella successiva si è notato un comportamento di overfitting ai valori della q-table, il quale preclude l’apprendimento delle fasi intermedie del percorso andando ad aumentare i tempi di training in modo esponenziale.



**Versione 3.0:**

Dopo numerose iterazioni del modello, si è riusciti ad implementare la tecnica di epsilon dinamico descritta in precedenza; questo, con l’aggiunta di un metodo per salvare i risultati dell’apprendimento per poi riprenderlo in seguito, ha portato finalmente l’agente a completare il percorso per la prima volta. Attraverso un numero notevole di epoche, si è riusciti a generare un percorso ottimale da seguire per raggiungere la ricompensa finale.



**Versione 4.0:**

Nell’ultima versione del progetto [3], si è utilizzata ancora la tecnica di epsilon dinamico, con l’aggiunta di una modifica alla reward che consiste nel generare una ricompensa di avanzamento solo dopo “n” spostamenti nella direzione corretta. Implementando questa modifica si è riusciti a incentivare ancora di più un comportamento dell’agente che lo porti a pianificare il percorso in base agli stati più lontani; Questo ha portato ad un addestramento più rapido in termini di tempo e complessivamente più preciso.



**https://www.youtube.com/watch?v=xdwjN54RKm4**

**Osservazione:**

Si è osservato durante l’addestramento che in rare occasioni l’agente riusciva ad ottenere una reward elevata pur trovandosi in uno stato iniziale del percorso; Si suppone, poiché non si è riuscito a riscontrare questo comportamento durante il render dell’applicazione, che l’agente sfruttasse un bug per arrivare direttamente alla fine del livello senza ottenere le reward intermedie.

Questo comportamento non voluto potrebbe essere stato generato da un bug riguardante le collisioni degli attori nel gioco o da un’interazione non prevista causata dalla velocità di esecuzione dell’applicazione, senza il render di gioco attivo.

**Conclusioni e sviluppi futuri**

Il progetto proposto è riuscito a dimostrare che la tecnica di RL sia efficace per la risoluzione di problemi a stato finito, tuttavia, se presa singolarmente, richiede un adeguato studio delle ricompense e dei parametri utilizzati, poiché anche solo per verificare che un parametro modificato in una certa maniera stia avendo più o meno successo bisogna comunque lasciarlo lavorare per molte iterazioni del modello.

Un’altra problematica di questo specifico algoritmo è la poca scalabilità del modello ad applicazioni diverse, siccome l’agente dovrebbe imparare non solo a completare il percorso ma anche a sviluppare una politica di movimento adatta a tutti i contesti;al contrario, l’algoritmo diventa molto scalabile se applicato allo stesso gioco e con gli stessi obiettivi, con l’unico lato negativo di dover allenare nuovamente l’agente da zero che però mantiene tutti gli stessi parametri e alle ricompense valutate durante la progettazione dell’algoritmo.

Alcuni possibili sviluppi futuri potrebbero riguardare:

* L’implementazione di un algoritmo che permette di scalare il modello da poterlo così applicare a più livelli dello stesso gioco, diminuendo l’allenamento necessario dell’agente;
* L’aggiunta di una rete neurale per facilitare il calcolo delle ricompense e dei parametri corretti in base al tipo di problema da risolvere, da utilizzare durante l’allenamento e permettere all’agente di generare una politica di apprendimento *data-based*;
* L’utilizzo di una diversa tecnica di *RL,* così da comparare i risultati e mostrare la tecnica più efficace;
* Il miglioramento dell’algoritmo e del modello esistente per affrontare livelli più complicati come l’aggiunta di ostacoli o obiettivi secondari;

In conclusione, il progetto si è rivelato un ottimo elemento di studio per capire i limiti e le potenzialità della tecnica di *reinforcement learning* basata sul *Q-Learning* ed un ottimo punto di partenza per poter approfondire l’utilizzo di questo metodo anche per altri ambiti applicativi.

**Riferimenti**

[1] <https://github.com/tomamic/fondinfo>

[2] <https://www.baeldung.com/cs/epsilon-greedy-q-learning>

[3] <https://github.com/AndreaBersellini/Reinforcement_Learning_Project/blob/main/rl_results.txt>