Progetto Sistemi Intelligenti Avanzati

Alessandro Di Liberti

Contents

1 Introduzione

Il presente report documenta il processo di progettazione di un agente intelligente in grado di giocare al videogioco Atari Breakout sfruttando l'apprendimento con rinforzo.

Breakout è un videogioco sviluppato da Atari negli anni 70, l'obiettivo è quello abbattere tutti i blocchi presenti sullo schermo utilizzando una palla rimbalzante. Il giocatore controlla una barra orizzontale posizionata nella parte inferiore dello schermo, spostandola da sinistra a destra per riflettere la palla e mantenerla in gioco. La palla rimbalza contro i blocchi, distruggendoli man mano che colpisce ognuno di essi. Il livello si completa quando tutti i blocchi sono stati eliminati.

2 Descrizione dell'ambiente

Per simulare l'ambiente si utilizza la libreria 'Gymnasium' (gym) sviluppata da OpenAI.

Gym offre una collezione di ambienti generalmente usati per addestrare agenti RL, uniti ad una semplice implementazione per python.

2.1 Implementazione

La semplicità di gym deriva da come viene permessa l'intereazione con l'ambiente: una volta creato l'environment l'agente interagisce mediante una funzione `step(action)` che ritorna:

- il nuovo stato
- il reward

- se l'episodio è terminato
- se l'episodio è stato troncato
- alcune informazione sull'ambiente.

Il parametro `action` della funzione è un valore discreto che rappresenta un'azione scelta dall'agente e consentita nell'ambiente scelto.

In questo progetto sono stati utilizzati due ambienti di gym: Cartpole e Breakout.

La struttura dello stato dipende dall'ambiente scelto, esistono due tipi di strutture principali:

1. tramite variabili di stato (Cartpole)
2. tramite una matrice di pixel che rapresenta lo schermo (Breakout)

3 Metodologia

Pur essendo l'obiettivo finale quello di addestrare un agente su Breakout, ho iniziato implementando un agente che sapesse risolvere il problema del Cartpole utilizzando l'algoritmo di QLearning.

Poichè Breakout come ambiente è molto più complesso, questo approccio non poteva essere l'ideale, ho quindi usato una rete neurale che approssimasse la value function. Ho implementato questo approccio sempre sul problema del cartpole per osservare quali problematiche emergevano prima di spostarmi su Breakout.

L'ultimo passaggio è stato quello di implementare un algoritmo con alla base una NN utilizzando un framework ottimizzato per infine risolvere il goal di questo progetto.

3.1 Cartpole con QLearning

L'ambiente del cartpole (offerto da gym) rappresenta la stato con quattro variaili che indicano:

- Velocità del carrello
- Posizione del carrello
- Velocità angole del palo
- Angolo rispetto alla base

Queste quattro variabili ritornano valori continui, è quindi necessario discretizzare i valori per poter rappresentare lo stato attraverso una matrice. Il reward viene gestito tornando 1 in seguito ad ogni step in cui l'agente è stato in grado di bilanciare il palo. Per valutare la performance del modello, il reward di un episodio è stato calcolato sommando tutti i reward, quindi a reward più alti corrispondono episodi più lunghi.

3.1.1 Addestramento modello

L'addestramento mediante QLearning comprende la scelta di alcuni parametri, a seguito di diversi esperimenti e combinazioni dei valori dei parametri ho trovato che le seguenti scelte siano adatte per un addestramento efficiente.

Il primo parametro è il **numero di classi** in cui discretizzare le quattro variabili dello stato, ho trovato che con 25 si ottengono generalmente reward più alti

Il modello è stato addestrato su 17000 **episodi**. Si considera epoch/episodio tutti gli step che comprendono il momento in cui l'ambiente viene resettato, fino a quando la variabile 'terminated' viene imposta a TRUE; cioè il gioco è finito e si ha perso. Questo perchè aumentando il numero di episodi il modello tende ad andare in overfit, cioè non è più in grado di approssimare e i reward calano drasticamente.

L'agente è stato addestrato seguento una politica **epsilon-greedy**: un approccio molto usato per avere un agente che esplora molto nelle fasi iniziali e si perfeziona nelle fasi finali. Sfrutta un parametro $\epsilon < 1$, che indica la probabilità di eseguire un'azione randomica, questo parametro viene diminuito man mano che si procede con il training. A seguito di vari esperimenti ho trovato che il modo più efficente per addestrare il modello fosse quello di trattare ϵ nel seguente modo:

- inizializzazione a 0.8
- scelta di azioni random per le prime 8000 epoch, ignorando quindi la politica epsilon greedy ed utilizzando una politica di azioni random
- decrementare epsilon di 0.00005 ad ogni episodio fino al raggiungimento di 0.0001, poi mantenere quel valore costante

Gli ultimi due parametri sono il **learning rate** (α) e il **discount factor** (γ), vengono usati nella formula di Bellman Ford per calcolare i valori delle coppie stato-azioni durante il training. α è stato impostato a 0.1 e viene decrementato di 0.000008 dalla 12000esima epoch, ho scelto di usare un time-dependet LR per prevenire il fenomeno dell'overfitting, fenomeno che si è verificato usando un LR costante.

γ invece rimane costante per tutto il training al valore di 1.

3.1.2 QLearning

L'algoritmo QLearning viene definito off-policy, ovvero il QValue della coppia (stato,azione) viene calcolato tenendo conto dell'azione ottimale che si può scegliere in un determinato stato, e non dell'azione che si sceglierebbe se si seguisse la policy. Più precisamente ad ogni step si aggiorna il QValue con la seguente formula di Bellman-ford.

$$Q^{\pi}_{k+1}(s,a) = Q^{\pi}_{k}(s,a) + \alpha[r' + \gamma \max_{a'} Q^{\pi}_{k}(s',a') - Q^{\pi}_{k}(s,a)] \tag{1}$$

Si utilizza una tabella chiamata QTable in cui viene memorizzato il QValue delle coppie (stato,azione). L'idea è quindi quella di definire una policy basandosi sulla QTable, cioè dato uno stato scegliere l'azione ottimale (quella con il max QValue)

Algorithm 1 QLearning

```
Inizializza lo stato Q(s, a) random
repeat
    Osserva stato s
    for t = 1 to T do
        Scegli un azione a_t secondo la policy da Q (es: ϵ-greedy)
        Esegui azione a_t
        Osserva il reward r_t e il nuovo stato s_{t+1}
        Aggiorna Q secondo 1
    end for
until Terminazione
```

3.1.3 Risultati

In un esempio di training il risultato ottiene i seguenti reward (nelle ultime 7000 epochs)1.

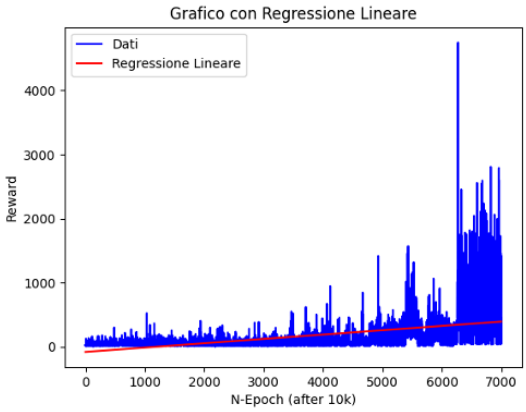

Figure 1: Reward delle ultime 7000 epoche

E' possibile notare che raggiunge picchi molto alti, ma ha un'alta varianza tra episodi. In realtà si può osservare che quando il modello ottiene risultati bassi è perchè fallisce nelle fasi iniziali, quando invece l'agente è in grado di 'bilanciare' il palo già dall'inizio i reward sono molto più alti e l'episodio dura di più. Questo si verifica poichè il cartpole di gym inizializza lo stato in seguito ad un reset con una piccola variazione randomica, quindi se l'agente non ha esplorato quel preciso stato, il palo parte già sbilanciato e l'episodio termina subito. Questo problema potrebbe risolversi con un tuning più preciso dei parametri (soprattutto sul valore di decadimento dei parametri).

3.1.4 Osservazioni

Un'osservazione da fare in seguito a questa prima fase del progetto è che l'implementazione di QLearning tramite matrice, sebbene molto efficiente (il modello viene addestrato in pochi minuti) non è in grado di modellare stati più complessi, quindi questo approccio con Breakout è irrealizzabile visto che lo stato si rappresenta con una matrice 160×210 in cui ogni cella può assumere un valore $\in [0, 254]$.

Un altro problema è che la matrice non è in grado di astrarre situazioni simili: per un essere umano il posizionamento della pallina con un pixel di differenza non rappresenta un vero e proprio stato differente, mentre per un agente con una matrice, anche una minima variazione rappresenta uno stato completamente diverso.

Entrambi questi problemi si risolvono utilizzando una rete neurale che approssima la QTable

3.2 Cartpole con Deep Q-Learning

Con Deep-Q-Learning o Deep-Q-Network, abbreviato in DQN, ci si riferisce ad un algoritmo che fa uso di una rete neurale che sostituisce la QTable. In questo algoritmo quindi l'apprendimento non consiste nell'aggiornare la tabella ma piuttosto nell'aggiustamento dei pesi dei neuroni che compongono la rete attraverso backpropagation. L'apprendimento della funzione valore in DQN è basato sulla modifica dei pesi in funzione di una loss function:

$$\mathrm{L} = (E[r_t + \gamma \max_a Q(s_{t+1}, a_t)] - Q(s_t, a_t))^2 \tag{2}$$

Dove la prima parte rappresenta l'expected value ottimo, mentra la seconda il valore stimato dalla rete.

Gli errori calcolati dalla loss function saranno propagati al'indietro nella rete mediante un passo backword (backpropagation), seguendo la logica di discesa del gradiente. Infatti il gradiente indica la direzione di maggior crescita di una funzione e muovendoci in direzione opposta riduciamo (al massimo) l'errore.

Il fattore che permette alle DQN di impare è una memoria chiamata experience replay (D). Con questa tecnica viene presa l'esperienza dell'agente ad ogni step t $e_t = (s_t, a_t, r_t, s_{t+1})$ e viene salvata nella memoria D (generalmente implementata con una queue data-structure). Si utilizza la replay memory per allenare il modello e permettergli di rivivere esperienze passate; l'algoritmo di DQN funziona nel seguente modo:

Algorithm 2 DQN con Experience Replay

Inizializza Replay Memory D
Inizializza $Q(s, a)$ con pesi random
repeat
  Osserva stato iniziale s_1
  for $t = 1$ to T **do**
    Seleziona un azione a_t usando Q (es: ε-greedy)
    Esegui azione a_t
    Osserva il reward r_t e il nuovo stato s_{t+1}
    Salva la transizione (s_t, a_t, r_t, s_{t+1}) nel Replay Memory D
    Preleva un campione di transizioni (s_j, a_j, r_j, s_{j+1}) da D
    Calcola il target T_j per ogni transizione
    if s_{j+1} è Terminale **then**
      $T = r_j$
    else
      $T = r_j + \gamma \max_a Q(s_{j+1}, a)$
    end if
    Adddestra la rete Q minimizzando $(T - Q(s_j, a))^2$
  end for
until terminazione

Il target T rappresenta il valore ottimale dell'expected value.

3.2.1 Risultati

Addestrando il modello su questo ambiente si possono raggiungere risultati migliori rispetto a QLearning. Il problema che emerge con questa tecnica è il tempo di addestramento; con QLearning implementato attraverso una tabella si è in grado di addestrare il modello in pochi minuti, con DQN sono necessarie diverse ore. Questo problema è accentuato dal fatto che il sistema su cui ho addestrato il modello non supporta il calcolo parellelizzato su GPU (non disponendo di una scheda video Nvidia).

3.2.2 Osservazioni

Le principali osservazioni che emergono da questo step sono: il tempo di addestramento e l'instabilità della rete. Il primo è già stato spiegato sopra, il secondo riguarda il fatto che per task più complessi una DQN con singola rete è molto instabile durante l'apprendimento, questa instabilità rallenta ancora di più una convergenza o la rende quasi irraggiungibile. Per risolvere il problema si utilizzano due reti, una principale e una target, questa rete target viene aggiornata meno frequentemente rispetto alla rete principale e fornisce un obiettivo più stabile per l'addestramento. Periodicamente, infatti, la rete target viene sincronizzata con i pesi della rete principale. Questo approccio di utilizzare due reti contribuisce a una maggiore stabilità nell'addestramento di DQN.

Tenendo a mente quindi che il problema principale del progetto è il tempo di training, e che per ottenere risultati validi sarebbe stato necessario adottare una soluzione che avrebbe aumentato ancora di più il training, ho cercato degli algoritmi più leggeri e ottimizzati per il calcolo su una CPU, da qui la scelta di utilizzare A2C implementato mediante il framework Stablebaselines.

3.3 Breakout con A2C

Il passo finale di questo progetto è stato quello di realizzare l'agente che sapesse ottenere un buon punteggio medio a Breakout, ho considerato superato l'addestramento con un punteggio medio di > 200. Ho scelto di usare come algoritmo Advantage Actor Critic (A2C) e di implementarlo con Stablebaselines3, un framework che offre API per diversi algoritmi di RL. A2C rappresenta un'evoluzione di un altro algoritmo: A3C (Asynchronous Advantage Actor Critic) [1]. A3C ha dimostrato un'elevatissima efficienza su sistemi a CPU multi-core, data dalla capacità di eseguire diversi ambienti e addestrare in modo asincrono molteplici Actors e Critics.

La chiave di questo efficiente addestramento si basa quindi sul fatto che non viene eseguito un singolo ambiente, ma diversi in contemporanea. A2C offre l'efficienza di A3C (in realtà i benchmark

mostrano che sia anche migliore), ma in alternativa all'implementazione asincrona, offre una soluzione sincrona e deterministica che attende che ciascun attore completi il suo segmento di esperienza prima di eseguire un aggiornamento, facendo la media su tutti gli attori [2].

Il mondo dell'addestramento di agenti per giocare ai videogiochi Atari è stato ampiamente coperto da molti paper, quello più rilevante è però [3], in cui viene implementata una metodologia per il preprocessing degli ambienti Atari, ormai diventata uno standard in questo tipo di progetti. Questo preproccesing è così rilevante che in Stablebaselines3 viene fornito un metodo che permette di creare l'ambiente Atari già processato (creando un wrapper intorno all'ambiente originale).

3.3.1 Stable-Baselines3

Stable-Baselines3 è una libreria open-source sviluppata da OpenAI, progettata per semplificare l'implementazione e l'uso di algoritmi di apprendimento per rinforzo. Questa libreria fornisce un'implementazione chiara e modulare di diversi algoritmi avanzati, tra cui Advantage Actor-Critic (A2C), Proximal Policy Optimization (PPO), e altri.

Alcuni delle caratteristiche sono:

1. **Modularità:** La libreria è progettata con un'architettura modulare, facilitando l'estensione con nuovi algoritmi o la personalizzazione di componenti specifiche.
2. **Facilità d'uso:** Stable-Baselines3 fornisce un'interfaccia user-friendly per l'addestramento e la valutazione degli agenti, rendendo più agevole l'implementazione di esperimenti di apprendimento per rinforzo.
3. **Compatibilità con PyTorch:** La libreria è basata su PyTorch, una popolare libreria di Deep Learning.
4. **Community attiva:** La libreria è supportata da una comunità attiva, con aggiornamenti regolari e contributi dalla community, garantendo un ambiente di sviluppo dinamico e in continua evoluzione.
5. **Compatibilità con Tensorboard:** E' possibile integrare integrare la valutazione real-time del modello, grazie alla creazione di log compatibili con Tensorboard.
6. **Creazione di ambienti multipli:** Stable-Baselines3 fornisce la possibilità di creare molteplici ambienti Atari basati su gym, per migliorare l'efficienza dell'addestramento.

3.3.2 Preprocessing

Come detto in precedenza, il preprocessing delle immagini è una componente fondamentale per la riuscita di questo progetto. L'ambiente gym di Atari Breakout ritorna un'immagine di dimensione 210x160 in RGB.

Di seguito tutti gli step di preprocessing eseguiti:

- **Frame stacking:** Siccome dallo stato dell'ambiente rappresentato tramite un'immagine è impossibile ricavare informazioni quali direzione e velocità della pallina, è necessario codificare lo stato in modo diverso. Come stato si considera quindi un vettore di 4 frame che permettono all'agente di estrarre il movimento della pallina.
- **Resize:** La dimensione della matrice di pixel viene ridotta a 84x84 pixel.
- **Conversione a scala di grigi:** Per limitare i possibili valori di ogni pixel, la scala RGB viene convertita in una di grigi.
- **Frame skipping:** Come visto prima lo stato è composto da 4 frame; i frame non sono però consecutivi, viene prelevato un frame ogni 4.
- **Reward clipping:** I reward vengono limitati al valore massimo di 1.
- **Termination when life is lost:** L'episodio viene considerato terminato quando si perde una vita (Breakout normalmente considera perso l'episodio se si perdono 5 vite).

3.3.3 A2C

L'Advantage Actor-Critic (A2C) è un algoritmo di apprendimento automatico progettato per addestrare agenti in ambienti di decisione sequenziale. L'approccio di A2C integra i concetti di Policy Gradient e Advantage, mirando a migliorare la stabilità e l'efficienza dell'apprendimento rispetto agli approcci tradizionali.

In A2C, l'agente è composto da due componenti fondamentali: l'attore e il critico (implementati tramite reti neurali). L'attore apprende una politica $\pi(a|s;\theta)$, dove θ sono i parametri dell'attore. Questa politica rappresenta la distribuzione di probabilità delle azioni a dato lo stato s. Dall'altro lato, il critico stima la funzione di valore dello stato $V(s;\phi)$, dove ϕ sono i parametri del critico, indicando la stima del ritorno atteso partendo dallo stato s.

L'obiettivo principale di A2C è massimizzare la funzione obiettivo $L(\theta,\phi)$, una combinazione di Policy Gradient e Advantage. Questa funzione obiettivo è definita come la somma ponderata del logaritmo della politica dell'attore e del vantaggio dell'azione:

$$L(\theta,\phi) = \sum_{t=0}^{\infty} \left(\log \pi(a_t|s_t;\theta) \cdot A(s_t,a_t;\phi) - \beta \cdot H(\pi(\cdot|s_t;\theta))\right) \tag{3}$$

Dove:

- $A(s_t,a_t;\phi)$ rappresenta il vantaggio, indica quanto un'azione contribuisce in modo positivo o negativo al ritorno totale, tenendo conto della stima del valore dello stato.
- β è un parametro per regolare il valore di entropia nella formula.
- $H(\pi(\cdot|s_t;\theta))$ è l'entropia della politica, misura dell'incertezza dell'agente.
  - Dalla combinazione di β e H si è in grado di aumentare o diminuire l'esplorazione

L'idea chiave è distinguere tra le azioni che portano a risultati migliori o peggiori rispetto a quanto previsto dal valore stimato dello stato. Questo concetto permette di rafforzare le azioni che portano a risultati positivi e di indebolire quelle che portano a risultati negativi durante l'aggiornamento dei parametri dell'agente.

L'implementazione di questo algoritmo può essere suddivisa in quattro fasi:

1. **Raccolta esperienze**:
   - L'agente interagisce con l'ambiente utilizzando la politica corrente $\pi(\cdot|s;\theta)$.
   - Vengono raccolte esperienze nella forma di tuple (s,a,r,s')
2. **Calcolo dei vantaggi**:
   - I vantaggi $A(s,a;\phi)$ sono calcolati come la differenza tra il ritorno ottenuto dopo l'azione a nello stato s e la stima del valore dello stato $V(s;\phi)$.

$$A(s,a;\theta) = r + \gamma V(s';\phi) - V(s;\phi) \tag{4}$$

3. **Discesa del gradiente**:
   - I parametri dell'attore vengono aggiornati utilizzando la discesa del gradiente per massimizzare la funzione obiettivo $L(\theta,\phi)$, quelli del critico per minimizzare il MSE tra il ritorno atteso e il valore dello stato stimato dal critic.
4. **Iterazione**:
   - Si ripetono gli step fino a convergenza

Algorithm 3 A2C

Inizializza l'attore con i pesi θ e il critico con i pesi ϕ
for ogni episodio **do**
  Inizializza lo stato s
  for ogni passo di tempo t **do**
    Scegli l'azione a da una politica data $\pi(a|s;\theta)$
    Esegui l'azione a e osserva il reward r e il nuovo stato s'
    Aggiorna i pesi del critico ϕ per minimizzare $(r + \gamma V(s';\phi) - V(s;\phi))^2$
    Calcola l'entropia della politica $H(\pi)$ come $\sum_a \pi(a|s;\theta) \log \pi(a|s;\theta)$
    Calcola il vantaggio $A(s,a)$ come $r + \gamma V(s';\phi) - V(s;\phi)$
    Aggiorna i pesi dell'attore θ per massimizzare $\log \pi(a|s;\theta) A(s,a) - \beta H(\pi)$
    Aggiorna lo stato $s \leftarrow s'$
  end for
end for

3.3.4 Risultati

Attraverso questo algoritmo l'obiettivo del progetto è stato superato, si è ottenuto un agente in grado di ottenere un reward medio di ~ 250. L'agente è stato addestrato su 7 milioni di episodi con un training time totale di ~ 5 ore su un MacBook Pro con chip M1 Pro (8 core).

Di seguito il grafico del training 2:

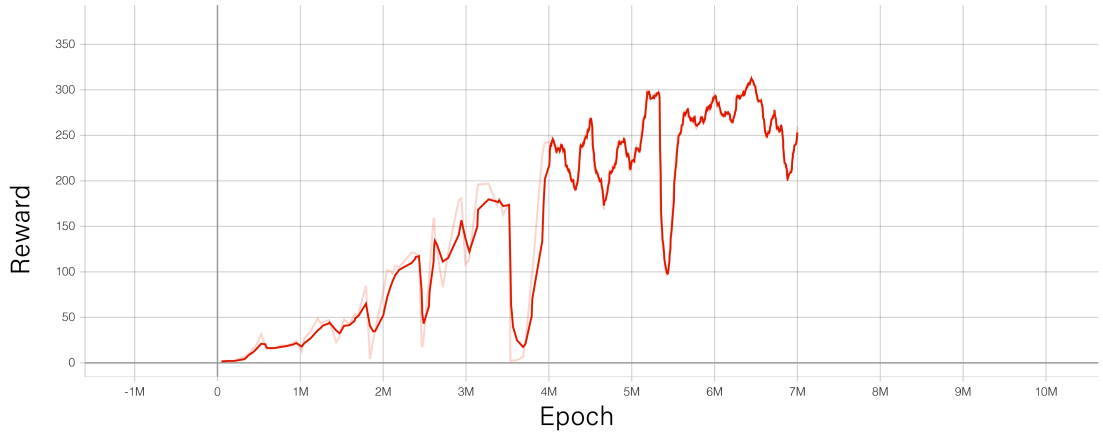

Figure 2: Training con 7mln epochs

3.3.5 Osservazioni

Una considerazione degna di nota emersa in questo step è l'enorme differenza tra l'addestrare l'agente con e senza preprocessing dell'immagine. Di seguito un grafico 3 che mostra le fasi iniziali del training in cui si nota come l'agente addestrato su un ambiente preprocessato dopo una prima fase di esplorazione randomica, inizi ad imparare e migliorare il suo reward; mentre l'agente addestrato su ambiente non preprocessato rimanga bloccato ai reward iniziali.

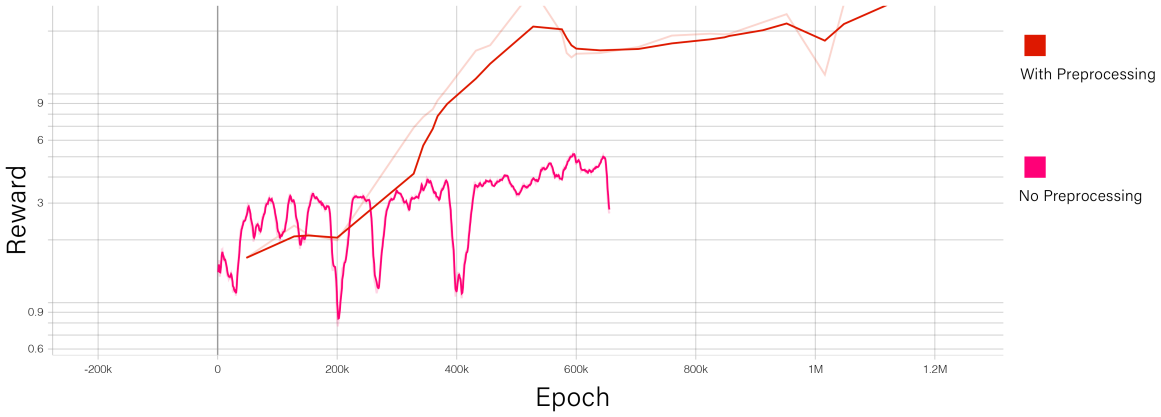

Figure 3: Comparazione tra addestramento con preprocessing (rosso) e senza preprocessing(rosa)

Il grafico in rosa è solo un campione dei primi esperimenti senza preprocessing, anche altri test hanno mostrato la stessa tendenza a non migliorare.

Degne di nota sono invece le grandi possibilità messe a disposizione da Stablebaselines3. Oltre ai vantaggi elencati sopra c'è anche quella di utilizzare gli algoritmi disponibili su ambienti non necessariamente di gym, purchè rispettino la stessa interfaccia (wrappando l'ambiente con delle classi wrapper messe a disposizione dal framework). Questa possibilità permette anche ai meno esperti del settore di addestreare agenti RL tramite algoritmi più complessi e difficili da implementare.

References

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[2] Openai baselines: Acktr a2c.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.