Flappy bird

Antonio Maddaloni, Francesco Peluso

Intelligenza Artificiale A.A. 2024/2025

Indice

1 Introduzione

Il progetto prevede la realizzazione di un agente di Reinforcement Learning in grado di giocare al celebre gioco *Flappy Bird*. In particolare, l'agente dovrà apprendere le corrette azioni da intraprendere durante il gioco, con l'obiettivo di superare il maggior numero di ostacoli possibile accumulando un punteggio elevato. L'ambiente di gioco è rappresentato da un'interfaccia 2D in cui l'agente, raffigurato come un uccello, deve evitare collisioni con tubi verticali posti a distanze e altezze variabili. Inoltre, l'ambiente è caratterizzato da dinamiche di gioco quali la gravità e il movimento continuo verso destra, che rendono il compito di navigazione più complesso.

Per la realizzazione dell'agente è previsto l'utilizzo dell'algoritmo PPO (*Proximal Policy Optimization*), uno dei metodi di Reinforcement Learning più avanzati ed efficaci per contesti di apprendimento continuo. L'obiettivo del progetto è testare le diverse configurazioni dell'ambiente di gioco e dell'agente, al fine di identificare le configurazioni ottimali e comprendere i limiti delle capacità dell'agente nell'adattarsi a un ambiente dinamico e impegnativo come quello di *Flappy Bird*.

2 Implementazione

2.1 Interfaccia 2D

Per l'implementazione dell'interfaccia 2D del progetto è stata utilizzata la libreria *PyGame*. La finestra di gioco è rappresentata da un ambiente laterale continuo, dove l'agente, raffigurato come un uccello, deve superare ostacoli verticali costituiti da coppie di tubi. Questi tubi presentano aperture a diverse altezze, rendendo il gioco dinamico e impegnativo.
Il movimento dell'uccello è influenzato da una forza gravitazionale costante, che lo spinge verso il basso, mentre l'azione dell'agente consente di invertire temporaneamente questa forza per mantenere il volo. Gli ostacoli si muovono orizzontalmente verso sinistra, simulando lo scorrimento continuo dell'ambiente, mentre l'agente rimane fisso nella sua posizione orizzontale.
Per rendere l'interfaccia di gioco più coinvolgente, sono stati aggiunti diversi elementi visivi. Il punteggio corrente, che aumenta ogni volta che l'agente supera un ostacolo, fornisce un feedback immediato sulle prestazioni del giocatore. Quando l'uccellino collide con i tubi o colpisce il bordo superiore o inferiore dello schermo, appare una chiara scritta "Game Over", segnalando la fine della partita in modo intuitivo.
Per aggiungere dinamicità e varietà al gioco, il backgroud con il colore dell'uccellino e dei tubi cambia ad ogni nuova partita che si inizia. Questo elemento visivo migliora l'esperienza complessiva, rendendo il gioco più accattivante e stimolante.
I comandi di gioco sono stati programmati attraverso l'uso di listener di eventi scaturiti dalla tastiera. Il tasto *spazio* svolge un ruolo fondamentale: permette di avviare una nuova partita all'inizio, di far compiere un salto all'uccellino durante il gioco e, in caso di *Game Over*, di riavviare il gioco. Se nessun tasto viene premuto, l'uccellino plana in base alla velocità determinata dalla forza gravitazionale. Infine, per uscire dalla partita, il giocatore può utilizzare il tasto *ESC*. Questa configurazione semplice e intuitiva rende il controllo dell'interfaccia facilmente accessibile anche ai principianti.
Infine, sono stati fissati 60 FPS, per sessioni di gioco piu fluide.

Per poter eseguire il gioco, si può accedere alla cartella "FlappyBird" ed eseguire il file main.py. Quindi bisogna accedere col terminale alla cartella "FlappyBird", digitando il comando:

- cd path/FlappyBird/

Ed Infine digitare:

- py main.py

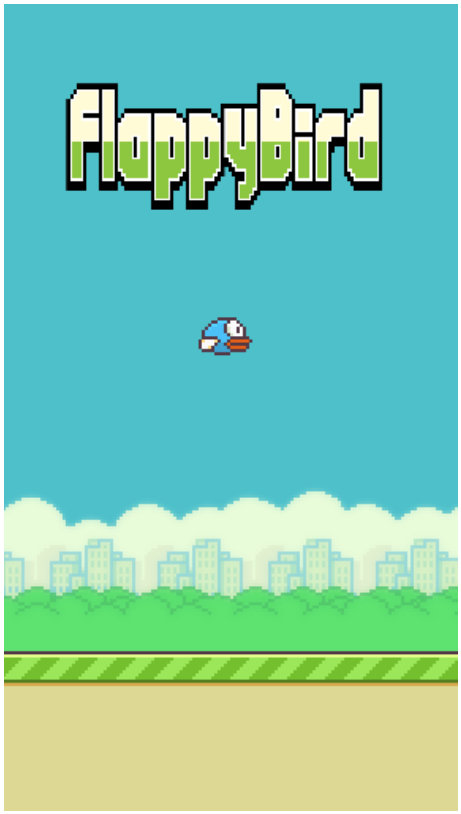

Figura 1: Interfaccia del gioco - Start Game

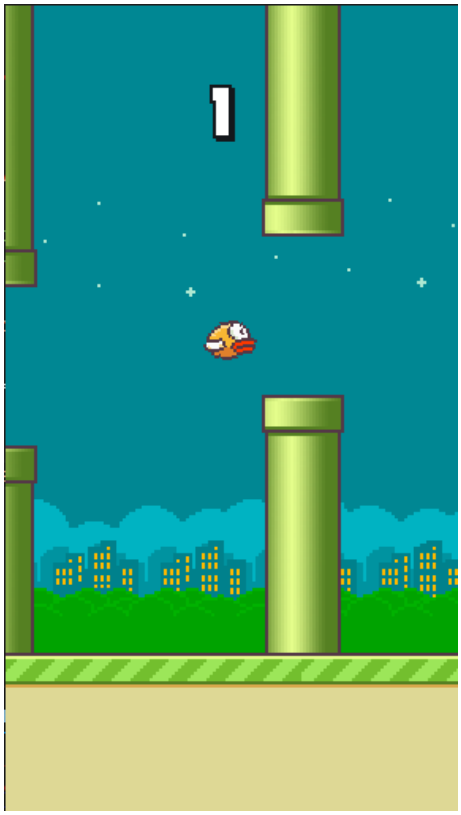

Figura 2: Interfaccia del gioco - In Game

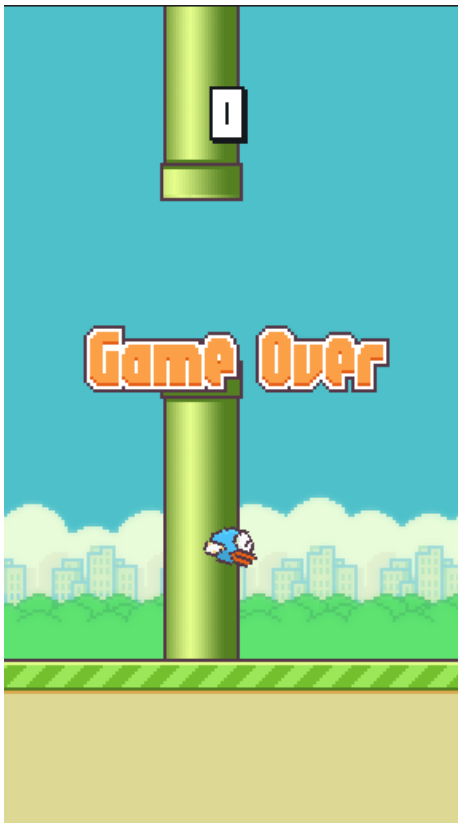

Figura 3: Interfaccia del gioco - Game Over

2.2 Ambiente

Per l'implementazione dell'ambiente e' stata scelta ed utilizzata la libreria Gymnasium. l'interfaccia grafica è stata definita senza un'ottimizzazione per Gymnasium, quindi abbiamo dovuto creare un ambiente ad hoc, apportando serie modifiche al progetto.
Le modifiche riguardavano:

1) Abbiamo reso l'ambiente eseguibile come modulo. Abbiamo definito un file '__init__.py' vuoto e il main.py trasformato in '__main__.py'.

2) L'ambiente era stato reingegnerizzato e partizionato su 4 funzioni principali:

- '__init__': serve per inizializzare l'ambiente.
- 'step()': serve per eseguire le mosse sul gioco ad ogni iterazione.
- 'reset()': serve per portare il gioco allo stato iniziale in caso di game over o inizio partita.
- 'render()': serve per renderizzare le mosse dopo ogni 'step()'.
- 'quit()': permette di chiudere il gioco in caso di game over.

3) Sono state cambiate le azioni:

- '1' per saltare.
- '0' per non fare nulla. Per chiudere il gioco ci pensa la funzione 'quit()'.

4) Ogni funzione definita nel punto (2), era necessario calcolare e ritornare delle informazioni, come le reward, lo stato dell'osservazione, e altre informazioni aggiuntive.

5) L'ambiente doveva essere dato in pasto a un algoritmo di Reinforcement Learning. Di conseguenza, ulteriori modifiche sono state apportate al 'Game Over', poiché nell'ambiente originale, il gioco non si riavviava immediatamente dopo una perdita. L'effettivo 'Game Over' avveniva solo dopo che l'uccellino fosse caduto, lasciando un margine di tempo durante il quale le azioni eseguite risultavano inutili. Questo poteva portare l'IA a eseguire azioni inefficaci.
Inoltre, i controlli sul salto bloccavano l'azione se si tentava di saltare ad ogni frame, consentendo salti solo dopo un certo intervallo di millisecondi. Per evitare che l'IA eseguisse azioni inutili e per non alterare il processo di apprendimento, abbiamo deciso di consentire un'azione per ogni frame.
Infine, un'ulteriore modifica è stata apportata durante la fase di 'Start Game'. L'ambiente originale non iniziava se non si premeva spazio (azione ora equivalente ad '1'). Per evitare di influenzare negativamente l'apprendimento, abbiamo eliminato la fase di 'Start Game': ora, quando viene applicata la funzione 'reset()', il gioco torna allo stato iniziale ed inizia immediatamente, senza la necessità di un input manuale per avviare la partita. Questo cambiamento garantisce che il processo di apprendimento non sia compromesso da una fase di avvio non necessaria. Infine per rendere il gioco non troppo veloce per l'addestramento, è stato deciso di abbasssare il limite degli FPS a 30.

2.3 Spazio di Osservazione

Come spazio di osservazione sono state definite diverse configurazioni, ma sono state scelte specificatamente due principali (questo perché le altre erano molto simili ma non ottimali). Entrambe le configurazioni principali risultano essere di tipo Box, con una che è stata modificata rispetto all'altra nel tempo per ottimizzarne l'efficacia.

1)

Listing 1: Prima Configurazione

```
self.observation_space = spaces.Box(
    low=np.array([69.9, 11, -6, 0, 11, 0, 11, -1]),
    # Limiti inferiori
    high=np.array([77, 403, 8, 340, 403, 340, 403, 1]),
    # Limiti superiori
    dtype=np.float32)
```

- Il primo valore rappresenta la posizione sull'asse x dell'uccellino.
- Il secondo valore rappresenta i limiti superiori dell'asse y dell'uccellino.
- Il terzo valore rappresenta la velocità.
- Il quarto e il quinto valore rappresentano, rispettivamente, l'asse x e y del centro del gap tra i due tubi che l'uccellino deve superare.
- Il sesto e il settimo valore rappresentano la differenza tra le coordinate x e y del centro dell'uccellino e il centro del gap dei tubi.
- L'ottavo valore indica:
  - 1 se l'uccellino si trova al di sopra di un determinato range rispetto all'asse y del centro;
  - -1 se si trova al di sotto;
  - 0 se è all'interno del range.

2)

Listing 2: Seconda Configurazione

```
self.observation_space = spaces.Box(
    low=np.array([69.9, 11, -6, 0, 11, 0]),
    # Limiti inferiori
    high=np.array([77, 403, 8, 340, 403, 340]),
    # Limiti superiori
    dtype=np.float32)
```

In questa configurazione il numero di valori è stato ridotto. I primi cinque valori sono identici a quelli della configurazione precedente, ma il sesto valore rappresenta la distanza euclidea tra il centro dell'uccellino e il centro del gap dei tubi.

2.4 Reward

Sono state implementate diverse configurazioni di reward-penality, ma sono state scelte due categorie di configurazioni principali, poiché cambiavano solo alcuni valori, quindi risultavano molto simili.

1) Prima Categoria di Configurazione:

- Se l'uccellino va a sbattere, quindi si finisce in "Game Over", allora la ricompensa assegnata è pari a -1
- Se l'uccellino si trova al di sotto dell'asse y corrispondente al centro dell'apertura del tubo e l'azione eseguita è 0 (ovvero rimanere fermo), la ricompensa assegnata è pari a -0.1.
- Se l'uccellino si trova al di sotto dell'asse y corrispondente al centro dell'apertura del tubo e l'azione eseguita è 1 (ovvero il salto), la ricompensa assegnata è pari a 0.1.
- Se l'uccellino si trova al di sopra dell'asse y corrispondente al centro dell'apertura del tubo e l'azione eseguita è 0 (ovvero rimanere fermo), la ricompensa assegnata è pari a 0.1.
- Se l'uccellino si trova al di sopra dell'asse y corrispondente al centro dell'apertura del tubo e l'azione eseguita è 1 (ovvero il salto), la ricompensa assegnata è pari a -0.1.
- Se l'uccellino si trova all'interno del range dell'apertura del tubo, la ricompensa assegnata è pari a 0.2.
- Se l'uccellino passa completamente attraverso il tubo, la ricompensa assegnata è pari a 1.

2) Seconda Categoria di Configurazione (Migliore):

- Se l'uccellino va in "Game Over", allora la ricompensa assegnata è pari a -1
- Se l'uccellino scende di valore rispetto all'asse delle y (ovvero sale troppo), fino ad un valore che sia $<= di15$, allora la ricompensa assegnata è pari a -0.8
- Se l'uccellino si trova al di sotto, o al di sopra di un range ben definito dell'asse y corrispondente al centro dell'apertura del tubo , la ricompensa assegnata è pari a 0.01. Questa ricompensa è stata pensata per far capire all'uccellino che deve sopravvivere nel tempo, anche se non si trova precisamente nel range del centro. (Vincolo Temporale)
- Il range del centro è stato suddiviso su piu sotto-range. Questi sotto-range vengono sfruttati per incoraggiare maggiormente l'uccellino, per restare o avvicinarsi al centro. Quindi se si trova nel primo strato del range, ovvero quello piu esterno, allora la ricompensa assegnata è pari a 0.3. Invece, se si trova in un sotto-range intermedio, allora la ricompensa assegnata è pari a 0.4. Infine, se si trova nell'ultimo sotto-range, ovvero quello piu interno, allora la ricompensa assegnata è pari a 0.5. Questa ricompensa è stata pensata sia per premiare l'avvicinamento al centro che la sopravvivenza nel tempo. (Vincolo Temporale)

- Se l'uccellino passa completamente attraverso il tubo, la ricompensa assegnata è pari a 1.

2.5 Criterio d'arresto

Essendo un ambiente in cui l'addestramento si basa sul tempo di vita dell'uccellino, l'unico criterio di arresto si verifica quando l'uccellino entra nello stato di game over. Questo accade in uno dei seguenti casi:

- l'uccellino collide con una pipe da qualsiasi lato;
- esce dai limiti dello schermo verso l'alto o verso il basso,
- oppure si schianta contro una superficie (ad esempio, il terreno).

In sintesi, il criterio di fine partita si attiva quando l'uccellino colpisce una superficie, considerando che l'altezza è vincolata da un limite massimo e minimo.

2.6 Algoritmi Da Utilizzare

Per decidere quale algoritmo utilizzare, abbiamo analizzato l'ambiente nella sua totalità, con particolare attenzione allo spazio di osservazione e agli stati che può assumere. È emerso che si tratta di un ambiente con uno spazio di stato continuo e dinamico, il che rende algoritmi tradizionali come il classico Q-Learning non adatti, poiché questi funzionano meglio con spazi di stato discreti. Dopo un'attenta analisi della letteratura e dello stato dell'arte, abbiamo optato per l'utilizzo del Deep Q-Learning, per la sua capacità di gestire spazi di stato continui grazie all'uso di reti neurali profonde, e del PPO (Proximal Policy Optimization), noto per la sua stabilità e robustezza in ambienti complessi. Questa combinazione ci permette di sfruttare i punti di forza di entrambi gli approcci, bilanciando efficienza e performance durante l'addestramento. Del PPO si è deciso di utilizzare la policy chiamata "MlpPolicy".

2.7 Addestramento e Agenti

Sono stati creati due moduli per l'addestramento con i due diversi algoritmi, uno per DQN e uno per PPO. Le cartelle dei moduli sono denominate rispettivamente "DQN" e "PPO". Utilizzando questi due agenti, sono stati sviluppati diversi modelli, inizialmente basati sulla configurazione 1 delle reward e sulla configurazione 1 dello spazio di osservazione. Le prestazioni degli agenti sono state valutate apportando modifiche ai valori delle reward o ai range definiti nella configurazione 1. Successivamente, è stata testata la combinazione della configurazione 1 delle reward con la configurazione 2 dello spazio di osservazione, mostrando lievi miglioramenti.
Infine, si è passati alla configurazione 2 delle reward combinata direttamente con la configurazione 2 dello spazio di osservazione, escludendo la combinazione della configurazione 2 delle reward con lo spazio 1, poiché quest'ultimo appariva già visivamente subottimale. Come previsto, la configurazione 2 delle reward abbinata alla configurazione 2 dello spazio si è dimostrata la più performante. Pur mantenendo invariata la base della configurazione 2 delle reward, sono state effettuate piccole modifiche ai valori di range e alle reward stesse per ottimizzare ulteriormente le prestazioni.
Una volta individuata la configurazione ottimale, ci si è concentrati sulla regolazione

dei parametri degli algoritmi "PPO" d "DQN", usando **stable_baseline3**, come libreria principale. Sono stati regolati parametri come il numero di passi (n_steps), i passi totali (total_timesteps) e la dimensione dei batch (batch_size), aumentandoli o diminuendoli a seconda dei risultati ottenuti. Infine, una volta individuata la configurazione migliore per l'algoritmo PPO, si è proceduto a testare la stessa configurazione con l'algoritmo DQN.

2.8 Struttura del progetto

Il progetto è stato strutturato nel seguente modo:

- FlappyBird: è una cartella contenente il gioco base di FlappyBird, senza nessun algoritmo di Reinforcement Learning.
- FlappyBird_Gym: modulo contenente l'ambiente definito ad hoc di Gymnasium, con tutti i suoi asset, suoni. Tale ambiente è stato configurato come un modulo, ma è stato aggiunto un ulteriore file per accedere ai due ambienti definiti. Il file "__main__.py" definisce l'ambiente utilizzando la configurazione 2, delle reward e dello spazio di osservazione. Mentre il file "env.py" definisce l'ambiente con la configurazione 1 delle reward e la configurazione 2 dello spazio dell'osservazione, come detto nel paragrafo (2.7).
- Execution: modulo per eseguire l'agente addestrato.
- Models: è una cartella utilizzata per salvare i modelli addestrati al fine di valutare le diverse configurazioni e iterazioni. Attraverso il plot dei dati e una valutazione empirica visiva, si è potuto osservare come, con il progredire del tempo di addestramento, il modello migliorasse visibilmente. Tuttavia, nella cartella finale è stato mantenuto esclusivamente il modello finale considerato il migliore.
- PPO: modulo per eseguire l'addestramento dell'agente usando l'algoritmo di Reinforcement Learning PPO. Sulla base di quale ambiente utilzzare per addestrare l'agente, bisogna modificare il file "__init__.py".
- DQN: modulo per eseguire l'addestramento dell'agente usando l'algoritmo di Reinforcement Learning DQN. Sulla base di quale ambiente utilzzare per addestrare l'agente, bisogna modificare il file "__init__.py".
- Logs: non è un modulo, ma una cartella contenente un file "analyze_result.py", che serve per analizzare i dati degli agenti durante l'addestramento. Infine, ci sarà un ulteriore file, chiamato "monitor.csv", contenente i dati ottenuti dall'addestramento per il plot, generato da PPO o da DQN.

3 Risultati

Per individuare la configurazione migliore degli agenti, abbiamo testato diverse configurazioni, ma in questo documento riportiamo solo le tre principali, tra cui quella che si è dimostrata la migliore. Durante le varie prove di addestramento e test, abbiamo tracciato l'andamento delle reward cumulative per ogni configurazione, il punteggio massimo ottenuto e il progresso visivo dell'agente.
A differenza di un'analisi basata esclusivamente sui dati numerici, i risultati sono stati valutati anche attraverso una valutazione empirica visiva. L'obiettivo principale era osservare un miglioramento continuo dell'agente, e abbiamo notato che più tempo trascorreva durante l'addestramento, più l'agente migliorava visibilmente.
Le configurazioni e i parametri associati agli algoritmi di reinforcement learning sono stati salvati nelle rispettive cartelle dei moduli "DQN" e "PPO". Inoltre, durante la fase di train, sono stati memorizzati solo i pesi dei modelli che hanno prodotto risultati soddisfacenti in fase di test. I migliori esiti ottenuti per le configurazioni principali testate sono riassunti nella Tabella 1.

Modello	Ambiente	Parametri	Max Train Score	Max Test Score
model-1	2-2	PPO(256; 32768; 3Mln)	730	> 400
model-2	1-2	PPO(256; 32768; 3Mln)	181	93
model-3	2-2	DQN(256; 32768; 3Mln)	115	1

Tabella 1: Risultati Finali.

Come si può osservare dalla Tabella 1, il modello migliore risulta essere il Model-1. Nella legenda della tabella, la colonna denominata "Ambiente 2-2" indica la configurazione 2 delle reward combinata con la configurazione 2 dello spazio di osservazione. I parametri indicati come tuple di tre elementi (x,y,z) rappresentano rispettivamente: x la dimensione del batch (batch_size), y il numero di passi (n_steps) e z il numero totale di passi (total_timesteps).
In aggiunta, abbiamo analizzato i dati attraverso i plot dei risultati. Tuttavia, è importante notare che, in alcune situazioni, i plot possono risultare contraddittori: si potrebbero osservare reward cumulative elevate ma punteggi finali bassi.
Per quanto riguarda il Model-1, abbiamo riscontrato, grazie a valutazioni empiriche visive, che il modello migliorava costantemente con l'aumentare del tempo di addestramento. Siamo partiti con valori iniziali modesti, come una dimensione del batch ridotta e un numero di passi (n_steps) pari a 400, per poi incrementare gradualmente questi parametri fino a raggiungere quelli ottimali riportati in Tabella 1.
Abbiamo osservato che:

- Con parametri più bassi, il modello mostrava miglioramenti significativi nelle fasi iniziali, ma tendeva a perdere efficacia nel lungo termine.
- Con parametri più alti, i progressi iniziali erano più lenti, ma il miglioramento nel tempo aumentava di molto, portando a risultati decisamente migliori nelle fasi finali dell'addestramento.

Questo comportamento conferma l'importanza di un bilanciamento tra le impostazioni iniziali e la capacità del modello di sfruttare pienamente l'aumento del tempo di addestramento.

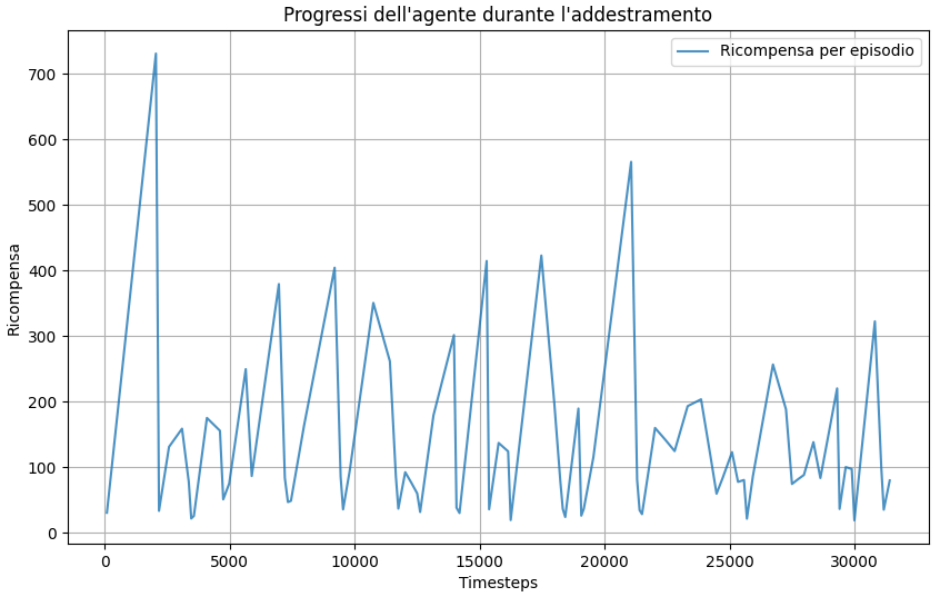

Figura 4: model-1: Migliore

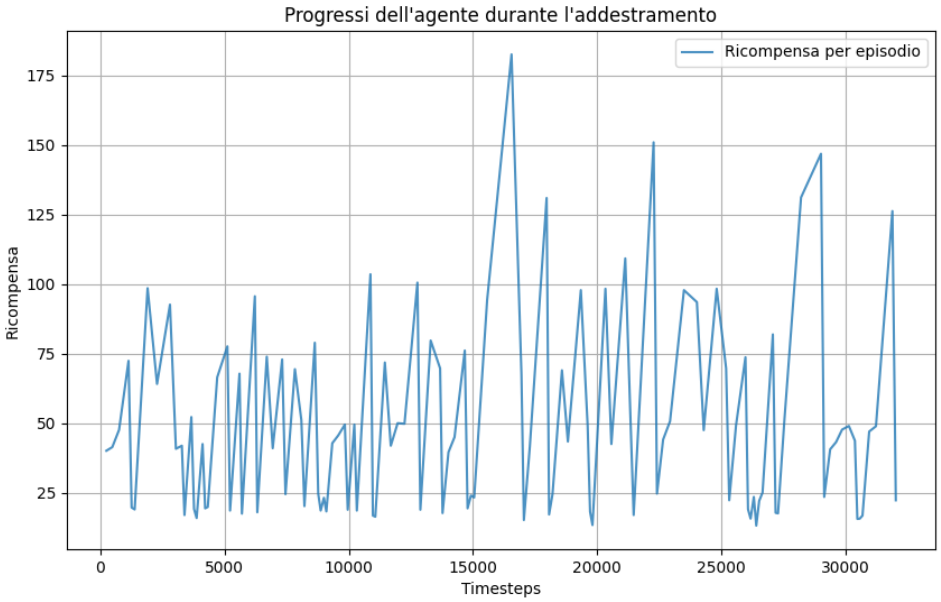

Figura 5: model-2

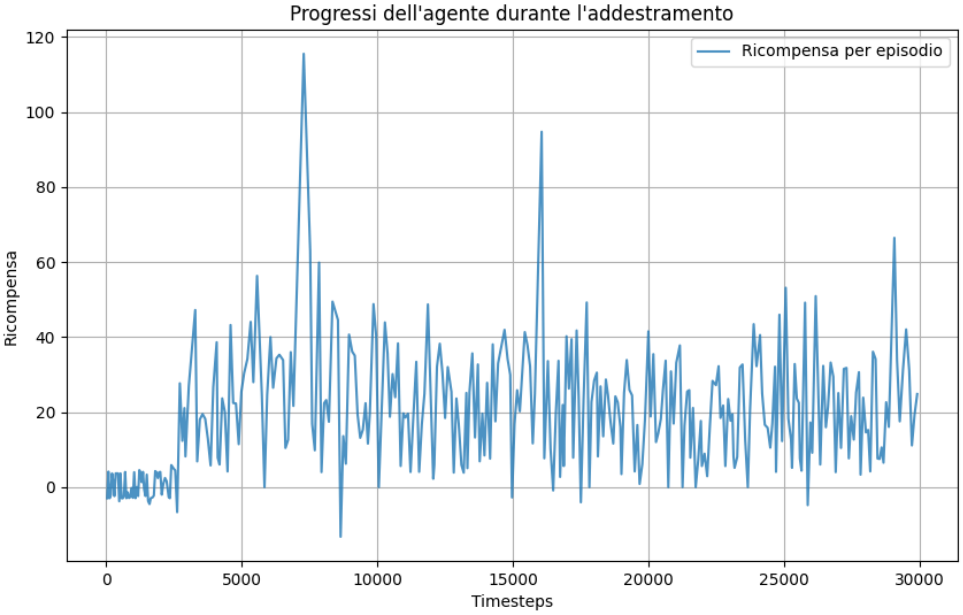

Figura 6: model-3

4 Conclusioni

Dall'analisi condotta emerge chiaramente che l'algoritmo PPO si è dimostrato superiore rispetto al DQN, anche in situazioni in cui le configurazioni di reward o di osservazioni risultavano meno efficaci. Questa robustezza ha permesso al PPO di ottenere prestazioni più consistenti e migliorare progressivamente, a differenza del DQN che tendeva a essere più sensibile alla qualità della configurazione utilizzata.
Un altro aspetto cruciale emerso è l'importanza del tempo come fattore determinante per le prestazioni dell'agente. Contrariamente a contesti dove il tempo è vincolato da parametri rigidi, come un numero massimo di passi o un "game over" causato da un limite temporale, in questo caso il tempo era legato esclusivamente alla capacità dell'agente di sopravvivere nell'ambiente. Di conseguenza, vincoli artificiali, come il parametro "truncate", si sono rivelati superflui. La gestione del tempo e della sopravvivenza dell'agente è stata quindi affidata esclusivamente ai parametri dell'algoritmo, come il numero di passi (n_steps), che si sono dimostrati fondamentali per bilanciare la rapidità iniziale di apprendimento e il miglioramento progressivo sul lungo termine.
Questo approccio ha evidenziato come l'ottimizzazione dei parametri temporali possa avere un impatto significativo sulle prestazioni finali. Con n_steps più bassi, l'agente tendeva a migliorare rapidamente nelle prime fasi, ma non riusciva a mantenere o espandere le sue capacità nel tempo. Al contrario, con n_steps più alti, il miglioramento iniziale era più lento, ma l'agente era in grado di ottenere performance nettamente superiori nelle fasi avanzate dell'addestramento. In sintesi, il successo dell'algoritmo PPO in questo contesto è legato alla sua capacità di adattarsi a diverse configurazioni e sfruttare il tempo di addestramento per migliorare continuamente. Inoltre, i risultati sottolineano l'importanza di considerare il tempo come una variabile chiave non solo nell'ambiente, ma anche nella definizione dei parametri dell'algoritmo, per ottimizzare le prestazioni dell'agente in scenari di reinforcement learning senza vincoli temporali espliciti.

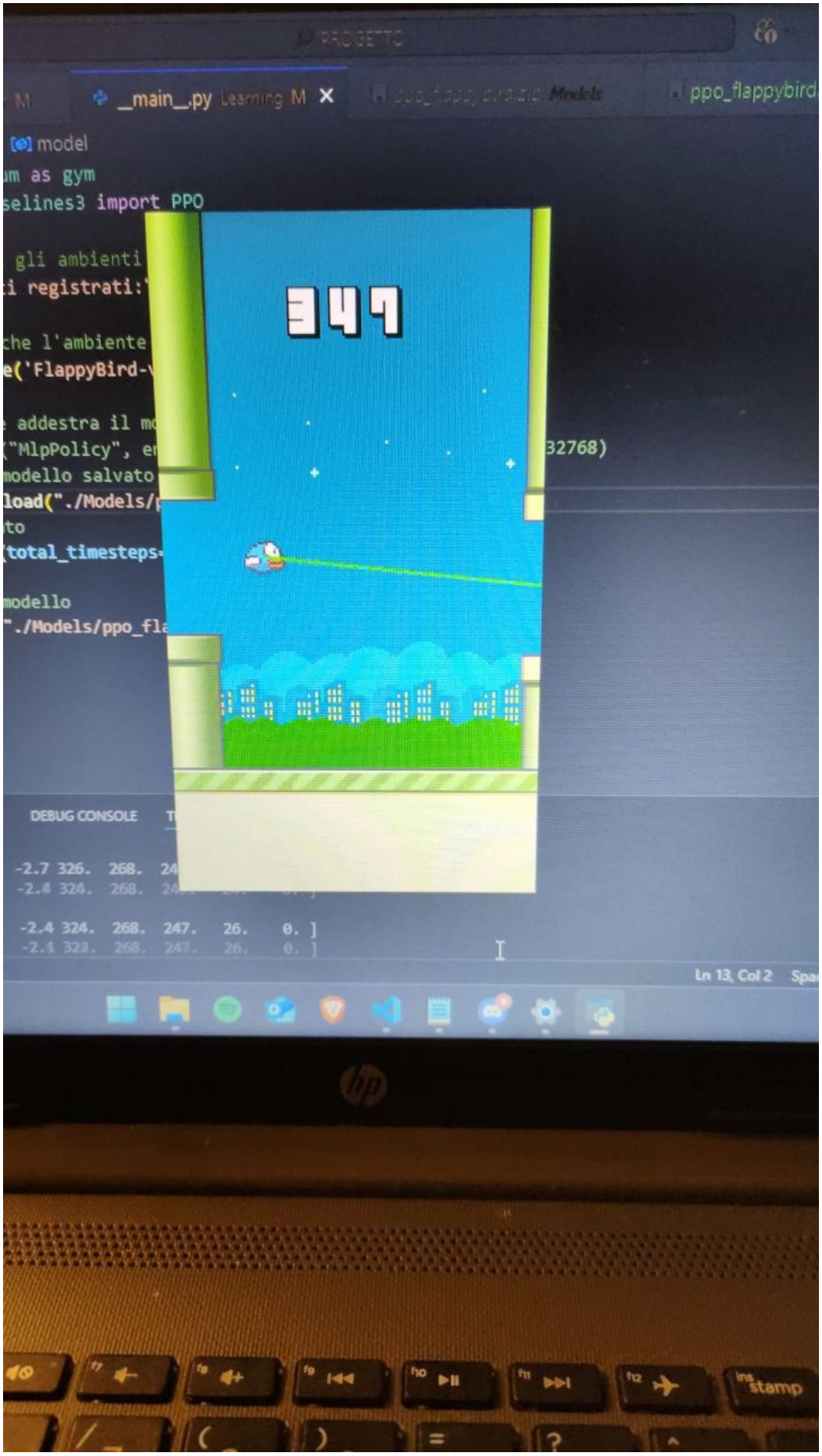

Figura 7: Best Score