

Flappy Bird IA

Josef Ascione, Davide Di Sarno, Simone Tartaglia

4 dicembre 2023

Indice

1	Introduzione	3
2	Obiettivi	4
2.1	Navigazione Intelligente	5
2.2	Ottimizzazione del Volo	6
2.3	Definizione degli Stati e delle Azioni	6
2.4	Gestione delle Penalità	6
3	Metodologie ed implementazioni	7
3.1	Creazione dell'Ambiente di Flappy Bird	7
3.2	Implementazione dell'Algoritmo Q-learning	8
3.3	Definizione degli Stati e delle Azioni	9
3.4	Addestramento Iterativo	10
3.5	Valutazione delle Prestazioni	11
4	Modelli	12
4.1	Q-learning	12
4.2	Deep Q-Network	12
4.3	SARSA	13

5	Risultati e motivazioni	14
5.1	Miglior punteggio nel gioco	14
5.2	Confronto tra i modelli	15
5.3	Sfide affrontate durante l'implementazione e risoluzione	17

1 Introduzione

Flappy Bird, sviluppato da Dong Nguyen nel 2013, è un videogioco arcade che ha catturato l'attenzione per la sua semplicità e sfida. Nel gioco, il giocatore guida un uccellino attraverso un percorso di tubi verdi, simili a quelli di Super Mario, premendo lo schermo per far "volare" l'uccellino e evitare collisioni con ostacoli.

Questo documento si propone di presentare un approccio innovativo per ottimizzare le prestazioni di Flappy Bird utilizzando l'apprendimento per rinforzo, in particolare il Q-learning. L'apprendimento per rinforzo implica che un agente apprenda a eseguire un compito attraverso l'esperienza, e in Q-learning, l'agente sviluppa una funzione di valore che assegna a ogni stato e azione un valore rappresentante l'aspettativa di ricompensa futura.

Il cuore del nostro metodo è l'utilizzo di Q-learning per apprendere una strategia di gioco che consenta all'uccellino di navigare tra i tubi massimizzando le probabilità di successo. Per valutare l'efficacia del nostro approccio, confrontiamo i risultati con un'altra implementazione basata su SARSA, un algoritmo on-policy, e con un modello più avanzato, il Deep Q-Network (DQN). È importante sottolineare che Q-learning è un algoritmo off-policy, il che significa che l'agente può apprendere una strategia di gioco diversa da quella attualmente in uso, consentendo una maggiore flessibilità. In contrasto, SARSA è un algoritmo on-policy, il che implica che l'agente apprende una strategia coerente con le sue azioni correnti.

I risultati della nostra analisi indicano che Q-learning supera SARSA in termini di prestazioni nel contesto di Flappy Bird. Questo suggerisce che la capacità di apprendere strategie di gioco diverse dalle azioni correnti conferisce a Q-learning un vantaggio significativo in un ambiente dinamico come quello proposto dal gioco. Inoltre, il confronto con il DQN evidenzia l'efficacia del Q-learning anche in confronto a modelli più complessi, suggerendo che la semplicità dell'approccio può essere un vantaggio in un contesto di gioco relativamente diretto come Flappy Bird.

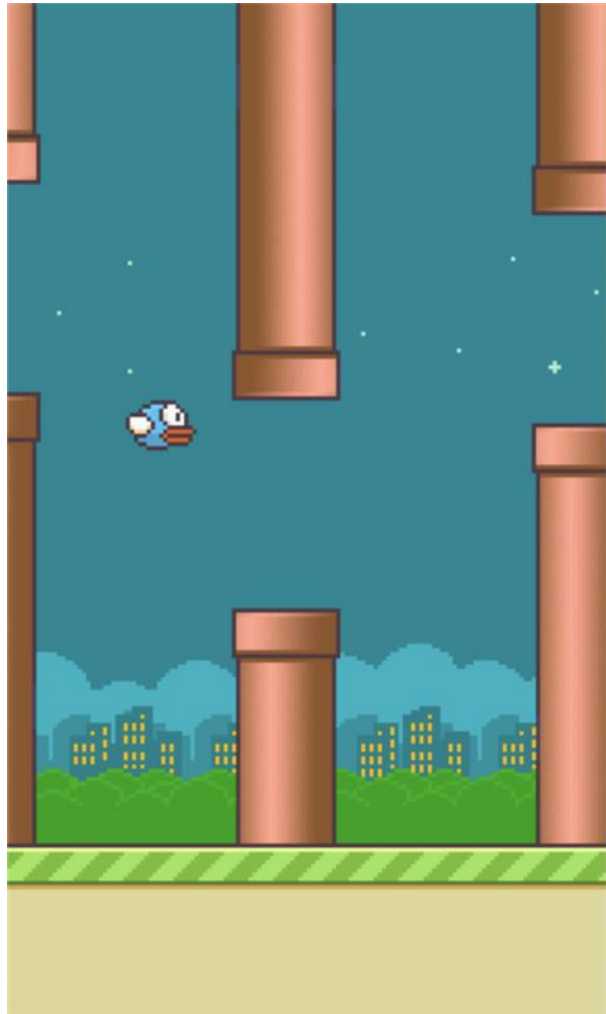


Figura 1: Schermata presa durante il testing dell'agente

2 Obiettivi

Durante l'assegnazione del progetto sono stati delineati 4 obiettivi principali:

1. **Navigazione Intelligente:** Creare un agente in grado di guidare l'uccellino attraverso gli spazi tra le tubature nel gioco Flappy Bird, evitando collisioni e mantenendo il più lungo tempo di gioco possibile.

2. **Ottimizzazione del Volo:** Addestrare l'agente a imparare un modello di volo ottimale, regolando l'altezza e la velocità per superare le tubature in modo efficiente.
3. **Definizione degli Stati e delle Azioni:** Identificare gli stati dell'ambiente, come posizione dell'uccellino, altezza e distanza dalle tubature, e definire azioni quali salto, planare o mantenere l'altitudine corrente.
4. **Gestione delle Penalità:** Implementare un sistema di ricompense e penalità per guidare l'apprendimento dell'agente, ad esempio, penalizzando collisioni con le tubature e ricompensando il superamento di nuovi record.

2.1 Navigazione Intelligente

Il modello **Q-learning** adotta una strategia basata sulla Q-table per affrontare l'obiettivo della navigazione intelligente durante una partita del gioco. La Q-table viene costantemente aggiornata attraverso le transizioni di stato, azione, ricompensa e stato successivo. Questo processo consente al modello di apprendere la migliore strategia per ogni stato, ottimizzando la navigazione del gioco. La continua iterazione sulla Q-table riflette l'adattamento dinamico alle diverse situazioni incontrate durante il gameplay.

Analogamente il **SARSA** gestisce una tabella aggiornata per tenere conto dell'azione effettivamente scelta durante l'esplorazione dell'ambiente quando aggiorna la sua tabella. Diversamente da Q-learning, SARSA segue un approccio on-policy, il che significa che apprende una strategia di gioco coerente con le azioni attualmente in esecuzione.

Il **Deep Q-Network**, che condivide l'approccio off-policy con il Q-learning, si concentra sull'ottimizzazione del volo catturando le transizioni ed salvandole nella replay memory. Questo approccio permette di preservare ed utilizzare esperienze passate, contribuendo al miglioramento progressivo delle ricompense in risposta alle varie situazioni di gioco. L'utilizzo della replay memory nel DQN riflette la capacità del modello di imparare da esperienze passate, contribuendo così a una gestione più avanzata delle dinamiche di volo.

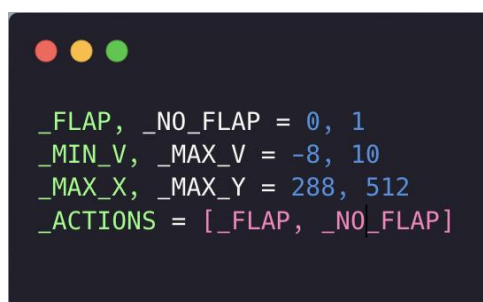
2.2 Ottimizzazione del Volo

Per l'ottimizzazione del volo, il modello **Q-learning** si avvale dell'aggiornamento della Q-table per ottimizzare la scelta delle azioni in base alle ricompense associate. In **SARSA** l'aggiornamento della tabella avviene in modo coerente con le azioni attuali dell'agente.

Il **DQN** si distingue nell'utilizzo di una funzione di valore implementata nella rete neurale. La selezione delle azioni è guidata dalla ricerca delle azioni che massimizzano questa funzione di valore. L'ottimizzazione del modello si traduce nell'addestramento della rete neurale per apprendere una strategia di volo ottimale, migliorando così le prestazioni nel gioco.

2.3 Definizione degli Stati e delle Azioni

I modelli implementati definiscono gli stati e le azioni nel metodo **state encoder**, considerando le informazioni rilevanti per la navigazione, come la posizione dell'uccellino, l'altezza dei tubi, la velocità e la distanza tra l'uccellino e i tubi. Le **azioni** durante una partita sono semplicemente la scelta tra "flap" e "no flap", scelta determinata in base alla conoscenza accumulata nella Q-table (**Q-learning** e **SARSA**) oppure seguendo la logica di una rete neurale (**DQN**).

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light green/cyan color and defines several variables: `_FLAP, _NO_FLAP = 0, 1`, `_MIN_V, _MAX_V = -8, 10`, `_MAX_X, _MAX_Y = 288, 512`, and `_ACTIONS = [_FLAP, _NO_FLAP]`.

```
_FLAP, _NO_FLAP = 0, 1
_MIN_V, _MAX_V = -8, 10
_MAX_X, _MAX_Y = 288, 512
_ACTIONS = [_FLAP, _NO_FLAP]
```

Figura 2: Variabili relative agli stati e alle azioni

2.4 Gestione delle Penalità

Le **penalità** sono un elemento chiave nella definizione delle strategie ottimali di azione in base alle transizioni di stato. In particolare, le penalità sono applicate in risposta a situazioni sfavorevoli, come le **collisioni** con i tubi, e hanno un impatto negativo sul punteggio complessivo dell'agente. Questo meccanismo incentiva l'agente a evitare comportamenti dannosi per il suo

successo nel gioco. D'altra parte, comportamenti desiderati, come superare nuovi record, vengono ricompensati positivamente. Le **ricompense** fungono da incentivo per l'agente, motivandolo a perseguire strategie che portino a risultati positivi e a migliorare le proprie prestazioni nel gioco. Nel contesto di Q-learning e SARSA, questo processo di penalizzazione e ricompensa è integrato nell'aggiornamento della Q-table. Le transizioni di stato, azione, ricompensa e stato successivo sono utilizzate per calcolare gli aggiornamenti della Q-table, riflettendo le esperienze passate dell'agente. Per quanto riguarda il modello DQN, il concetto di ricompensa rimane centrale, ma la gestione avviene attraverso la definizione e l'applicazione di ricompense nel processo di aggiornamento della rete neurale. In questo modo, la rete neurale impara a valutare le azioni in base alle ricompense associate, contribuendo a una strategia di gioco più sofisticata e adattabile.

A screenshot of a code editor with a dark background and syntax-highlighted text. At the top left, there are three colored window control buttons: red, yellow, and green. The code defines a dictionary for reward values.

```
# Valori reward
REWARD_Values = {
    "positive": 1.0,
    "tick": 0.0,
    "loss": -10.0,
}
```

Figura 3: Combinazione parametrica utilizzata dal DQN

3 Metodologie ed implementazioni

3.1 Creazione dell'Ambiente di Flappy Bird

L'implementazione dell'ambiente simulato di gioco basata sulla libreria **pygame** offre una solida base per la creazione di una replica fedele delle dinamiche di gioco di Flappy Bird. Questa libreria fornisce un insieme di strumenti e funzionalità che semplificano la gestione di aspetti cruciali del gioco, garantendo una rappresentazione accurata e realistica delle sfide affrontate dal

giocatore. La gestione della fisica del volo all'interno dell'ambiente riveste un ruolo fondamentale, replicando le leggi del movimento dell'uccellino nel gioco originale. La disposizione delle tubature e la generazione dinamica di elementi durante il corso della partita sono gestite in modo coerente con l'esperienza di gioco di Flappy Bird. Questo approccio assicura che l'agente affronti le stesse condizioni fisiche e strutturali del gioco originale, offrendogli un contesto di apprendimento realistico. La fedeltà nella rappresentazione dell'ambiente sottolinea l'importanza di garantire che le abilità acquisite durante l'addestramento trovino un'applicazione efficace e coerente nel contesto del gioco effettivo.

3.2 Implementazione dell'Algoritmo Q-learning

L'agente **Q-learning** è stato progettato per apprendere strategie per superare le tubature e massimizzare il punteggio attraverso la gestione della **Q-table** e le implementazioni di addestramento basate su transazioni di stato, azione e ricompensa. Durante la fase di addestramento, l'agente osserva le transizioni di stato, azione, ricompensa e stato successivo. La funzione **observe** si occupa di aggiornare dinamicamente la Q-table utilizzando l'equazione di aggiornamento caratteristica del modello, fondamentale è l'attenzione al valore massimo previsto dalla Q-table per determinare le sue aspettative di ricompensa. Il metodo **training_policy** definisce la politica di apprendimento che gestisce l'esplorazione dell'ambiente. La strategia di apprendimento dell'agente è delineata dalla funzione **training_policy**, che gestisce l'esplorazione dell'ambiente. L'agente deve scegliere tra il seguire una politica casuale o selezionare l'azione con il massimo valore Q associato allo stato corrente. Durante la fase di valutazione, la funzione **policy** guida l'agente nel selezionare l'azione ottimale in base alla massimizzazione dei valori Q disponibili. Questo approccio aiuta a garantire una giusta esplorazione dell'ambiente durante la fase di addestramento, consentendo all'agente di apprendere strategie più efficaci. L'implementazione include anche metodi per il salvataggio e il caricamento della Q-table, consentendo di preservare gli stati appresi tra le sessioni di gioco. Il metodo **_state_encoder** converte le informazioni dello stato del gioco (posizione dell'uccellino, altezza dei tubi, distanza dalle tubature e velocità) in intervalli discreti, permettendo una rappresentazione adatta per la Q-table.


```

class QLearning:
    def __init__(self):
        self.eps_start = Q_Parameters["epsilon_start"]
        self.eps_decay = Q_Parameters["epsilon_decay"]
        self.eps_end = Q_Parameters["epsilon_end"]
        self.alpha = Q_Parameters["alpha"]
        self.alpha_end = Q_Parameters["alpha_end"]
        self.alpha_decay = Q_Parameters["alpha_decay"]
        self.gamma = Q_Parameters["gamma"]
        self.partitions = Q_Parameters["partitions"]
        self.steps_done = Q_Parameters["steps_done"]

        self._ACTIONS = _ACTIONS
        self._FLAP = _FLAP
        self._NO_FLAP = _NO_FLAP
        self._MIN_V = _MIN_V
        self._MAX_V = _MAX_V
        self._MAX_X = _MAX_X
        self._MAX_Y = _MAX_Y

        self.Q_table = {
            ((y_pos, pipe_top_y, x_dist, velocity), action): 0
            for y_pos in range(self.partitions)
            for pipe_top_y in range(self.partitions)
            for x_dist in range(self.partitions)
            for velocity in range(-8, 11)
            for action in range(2)
        }

```

Figura 4: Configurazione del modello Q-learning

3.3 Definizione degli Stati e delle Azioni

Gli stati sono rappresentati come variabili di interi relative ad informazioni quali la posizione del giocatore, la distanza dalle tubature e la velocità. Questa rappresentazione dettagliata consente all'agente di percepire con precisione il contesto di gioco, facilitando la presa di decisioni informate durante il volo dell'uccellino. Le azioni disponibili, "flap" e "no flap", delineano le scelte di movimento dell'uccellino in risposta a ciascuno di questi stati. Per ogni istante della partita il giocatore può decidere di volare aumentando la propria altezza oppure non può decidere di non volare e quindi diminuirla gradualmente.



```
state = {
    "player_y": self.player.pos_y,
    "player_vel": self.player.vel,

    "next_pipe_dist_to_player": next_pipe.x + next_pipe.width/2 -
self.player.pos_x ,
    "next_pipe_top_y": next_pipe.gap_start,
    "next_pipe_bottom_y": next_pipe.gap_start + self.pipe_gap,

    "next_next_pipe_dist_to_player": next_next_pipe.x +
next_next_pipe.width/2 - self.player.pos_x ,
    "next_next_pipe_top_y": next_next_pipe.gap_start,
    "next_next_pipe_bottom_y": next_next_pipe.gap_start + self.pipe_gap
}
```

Figura 5: Sezione del codice relativa agli stati del gioco

3.4 Addestramento Iterativo

Durante l'addestramento, l'agente apprende le conoscenze di gioco dalle esperienze passate aggiornando la Q-table o il modello neurale in risposta alle transizioni affrontate. Parte degli esperimenti affrontati ha avuto come obiettivo l'ottimizzazione degli iperparametri dell'agente in fase di addestramento. Durante ciascun episodio di gioco, l'agente seleziona azioni sulla base della sua politica di addestramento, osserva gli effetti di tali azioni, e aggiorna la sua conoscenza in merito alle strategie più efficaci. Questo ciclo iterativo di selezione, osservazione e aggiornamento costante consente all'agente di affinare le sue risposte alle diverse situazioni di gioco, creando una progressiva evoluzione delle strategie implementate. In questo modo, l'agente si avvicina sempre di più a una performance ottimale nel superare le sfide proposte da Flappy Bird.

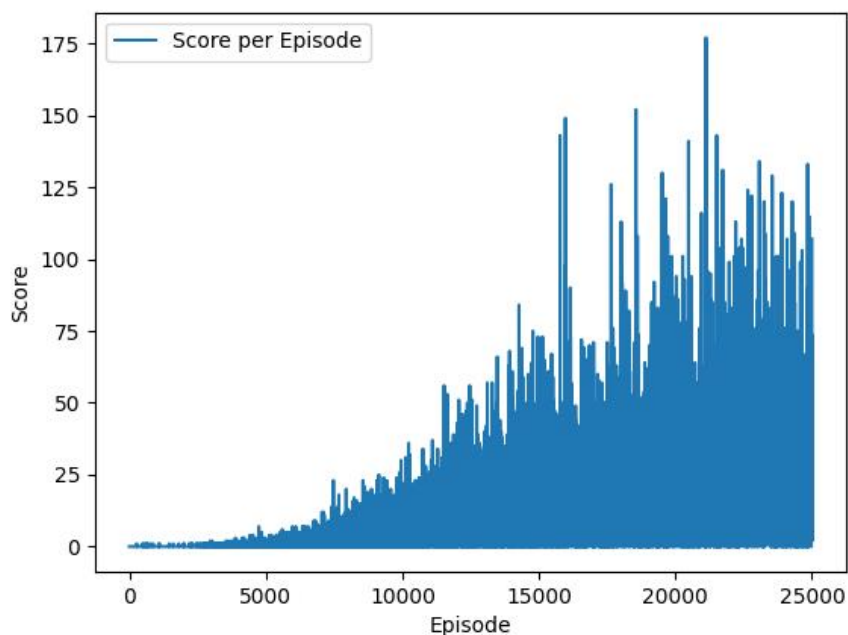


Figura 6: All'aumentare degli episodi dell'addestramento la media dei punti migliora

3.5 Valutazione delle Prestazioni

Le prestazioni dell'agente sono state monitorate e valutate considerando il punteggio del giocatore durante la partita, simulando lo svolgimento reale del gioco. La fase di testing dell'agente è stata eseguita su più episodi poiché la natura dinamica e mutevole del gioco rende necessario l'avere una visione più approfondita delle abilità apprese durante le diverse situazioni di gioco per poter valutare le prestazioni dell'agente. Per visualizzare in modo chiaro e intuitivo le performance dell'agente, abbiamo impiegato la libreria **matplotlib**, utilizzando **pyplot** per generare grafici ed avere una rappresentazione visuale delle performance dell'agente durante la fase di train o di test.

4 Modelli

Il modello viene addestrato sfruttando un agente di apprendimento per rinforzo, quindi sfruttando un meccanismo di ricompense al fine di addestrare l'agente a effettuare le azioni corrette. Nel nostro caso, sono stati sfruttati tre diversi algoritmi di apprendimento per rinforzo: **Q-Learning**, **Deep Q-Network** e **SARSA**. Sono stati valutate accuratamente le prestazioni di tutti e tre i modelli, selezionando infine quello più adatto nel nostro contesto applicativo.

4.1 Q-learning

Per **Q-Learning** intendiamo un algoritmo di apprendimento automatico, basato sulla programmazione dinamica, particolarmente efficace in ambienti discreti. Il suo funzionamento è basato sull'utilizzo di una **funzione Q**, la quale associa un valore a ogni coppia stato-azione, in modo da massimizzare la ricompensa cumulativa nel lungo termine. Avendo uno spazio delle azioni discreto, nonché una rappresentazione semplice degli stati, il Q-Learning ha prodotto risultati migliori rispetto agli altri due algoritmi.

4.2 Deep Q-Network

Il **Deep Q-Network**, invece, estende il Q-Learning con l'utilizzo di reti neurali profonde per approssimare la funzione Q in ambienti di grandi dimensioni. Nel nostro caso, data la scarsa complessità degli stati, nonché le piccole dimensioni dello spazio delle azioni, l'impiego di una rete neurale risulta superfluo, e anzi causa overfitting, producendo quindi risultati peggiori rispetto al Q-Learning.

```

class DQN(nn.Module):
    def __init__(self, n_observations, n_actions) -> None:
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

```

4.3 SARSA

SARSA, acronimo di **State-Action-Reward-State-Action**, prevede anch'esso la ricerca di una funzione Q ottimale, esattamente come il Q-Learning. La differenza principale tra i due algoritmi sta nell'aggiornamento della funzione Q in base all'azione successiva dell'agente, che avviene nel SARSA, anziché limitarsi a selezionare la migliore azione possibile nel nuovo stato come nel Q-Learning tradizionale. Rispetto agli altri algoritmi, il SARSA ha presentato risultati decisamente peggiori, per via della politica **on-policy** su cui si basa: essa prevede il miglioramento della politica corrente, prendendo decisioni in base alle azioni eseguite. Questo approccio risulta però inefficiente in un ambiente caratterizzato da una forte componente casuale come il nostro, in quanto limita la capacità da parte dell'agente di scoprire strategie ottimali.

```

def policy(self, state):
    state = self._state_encoder(state)

    if self.Q_table[(state, self._FLAP)] == self.Q_table[(state, self._NO_FLAP)]:
        return random.choice(self._ACTIONS)
    elif self.Q_table[(state, self._FLAP)] > self.Q_table[(state, self._NO_FLAP)]:
        return self._FLAP
    else:
        return self._NO_FLAP

```

5 Risultati e motivazioni

5.1 Miglior punteggio nel gioco

L'agente **Q-learning** ha dimostrato grande abilità nel superare le tubature in modo intelligente. Attraverso un processo di addestramento iterativo, l'agente ha acquisito strategie altamente efficaci, dimostrando la sua capacità di adattarsi dinamicamente alle mutevoli condizioni di gioco al fine di evitare collisioni e prolungare il tempo di gioco in modo significativo. Le ottimizzazioni delle prestazioni sono emerse chiaramente nel corso dell'addestramento iterativo, manifestandosi con miglioramenti sia nel tempo massimo di gioco raggiunto, che nella gestione delle penalità. Il punteggio massimo ottenuto, pari a **5780**, riflette il successo delle strategie apprese e l'efficacia del processo di ottimizzazione delle prestazioni dell'agente.

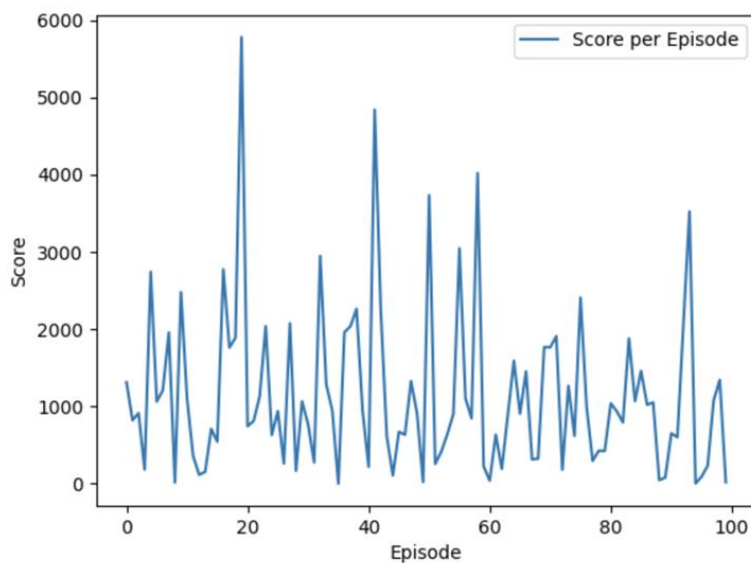
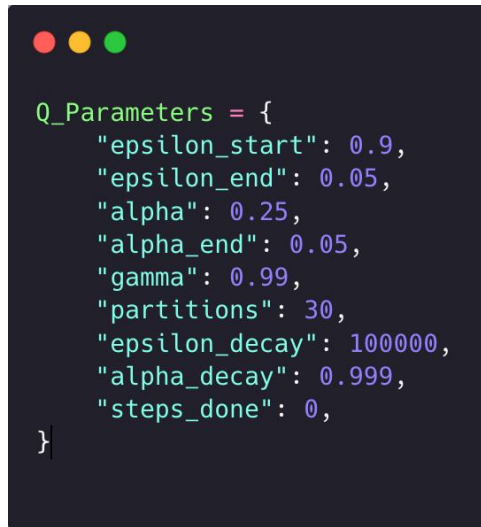


Figura 7: Grafico del miglior punteggio ottenuto in fase di test

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a Python dictionary of hyperparameters for Q-learning, with syntax highlighting: keywords in green, strings in blue, and numbers in purple.

```
Q_Parameters = {  
    "epsilon_start": 0.9,  
    "epsilon_end": 0.05,  
    "alpha": 0.25,  
    "alpha_end": 0.05,  
    "gamma": 0.99,  
    "partitions": 30,  
    "epsilon_decay": 100000,  
    "alpha_decay": 0.999,  
    "steps_done": 0,  
}
```

Figura 8: Miglior combinazione degli iperparametri per il Q-learning

5.2 Confronto tra i modelli

I risultati ottenuti dal confronto degli algoritmi Q-learning, SARSA e DQN in Flappy Bird hanno evidenziato una significativa superiorità dell'approccio Q-learning. Questa differenza può essere attribuita alla natura del gioco, caratterizzato da una sequenza di tubature con disposizione indipendente e randomica. L'ambiente, infatti, risulta essere **sequenziale** ma data la disposizione randomica dei tubi deve essere affrontato come se fosse **episodico**. D'altra parte, SARSA, con la sua politica specifica durante l'addestramento, risulta essere eccessivamente cauto, limitando la capacità dell'agente di esplorare efficacemente lo spazio delle azioni. Inoltre, considerando che Flappy Bird è un gioco relativamente semplice con dinamiche di gioco dirette, l'utilizzo di un modello più complesso come una rete neurale profonda, come nel caso del DQN, potrebbe essere eccessivo per un problema così lineare. Infine, il DQN con una rete neurale profonda potrebbe essere più incline all'overfitting rispetto a un modello più semplice come il Q-learning.

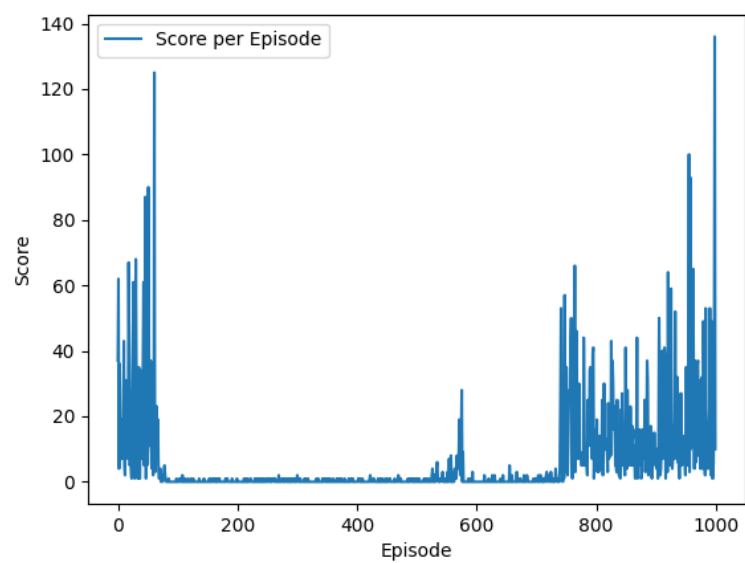


Figura 9: Grafico dell'addestramento con DQN

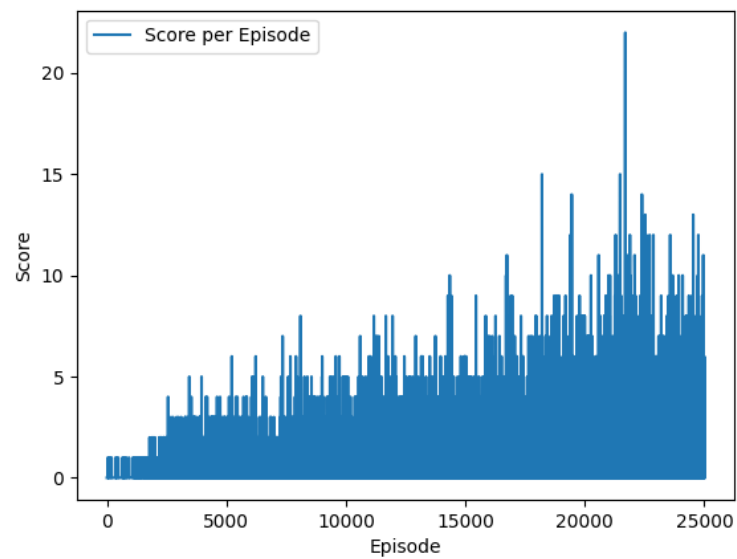


Figura 10: Grafico dell'addestramento con SARSA

5.3 Sfide affrontate durante l'implementazione e risoluzione

Le principali sfide durante l'implementazione riguardavano la gestione della Q-table. Inizialmente, il modello aggiornava la tabella, ma non la utilizzava effettivamente nella successiva fase di gioco. Questo portava a una gestione non ottimale delle altezze dei tubi, che variano in modo casuale nel gioco. La necessità di coordinare l'aggiornamento della tabella con le azioni effettive dell'agente è stata cruciale per ottenere risultati significativi. Successivamente, l'implementazione del DQN è stata introdotta dopo i primi test con il Q-learning. Questo passaggio è stato effettuato per affrontare le limitazioni riscontrate nella gestione della Q-table. Tuttavia pur essendo il DQN generalmente più efficace, per via della natura molto semplice dell'ambiente di gioco Flappy Bird, l'impiego di una rete neurale risulta essere eccessivamente complesso, causando problemi di overfitting entro poche epoche. Di conseguenza, la scelta finale è ricaduta sul Q-learning.