UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Applicazione del Deep Reinforcement Learning a videogiochi di piattaforme 2D

Application of Deep Reinforcement Learning to 2D platform video games

Candidato:
Emanuele Balloni

Relatore:
Prof. Primo Zingaretti

Correlatore:
Dott. Marco Mameli

Anno Accademico 2020-2021

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Sommario

Il Deep Reinforcement Learning (DRL) permette ad un agente (di un processo di Reinforcement Learning) di ricevere input ad alta dimensionalità ad ogni passo ed effettuare azioni in accordo con una policy basata su una rete neurale artificiale deep. Questo meccanismo di apprendimento si sposa perfettamente con il mondo dei videogiochi, dove gli ambienti sono sempre dinamici e all'IA è richiesto di apprendere molte informazioni. Questo lavoro di tesi porta l'attenzione su una categoria specifica di videogiochi, i platform 2D, andando a mostrare come sia possibile partire da prodotti già realizzati (non per un task di DRL) per creare delle reti neurali performanti, soffermandosi, in particolare, sulla distinzione tra platform a livelli ed endless, dove il primo ha come obiettivo il raggiungimento di una meta prefissata, il secondo, invece, prevede la sopravvivenza il più longeva possibile in un ambiente generato proceduralmente, cioè diverso ad ogni partita. I risultati ottenuti evidenziano che è possibile allenare delle IA per entrambi i casi, con risultati soddisfacenti; mostrano, inoltre, come i due giochi necessitino di reti allenate separatamente, in quanto differiscono ampiamente nell'obiettivo e nell'ambiente in cui gli agenti sono inseriti.

Indice

Elenco delle figure

[1] `https://en.wikipedia.org/wiki/Deep_reinforcement_learning/`

Elenco delle tabelle

Capitolo 1.

Introduzione

1.1. Contesto applicativo

L'intelligenza artificiale (IA) da tempo è entrata nel settore dei videogiochi, in particolare per ottenere performance in gioco comparabili a quelle di un essere umano. Più in generale, studia le complesse interazioni tra gli agenti e gli ambienti di gioco. Parallelamente, la varietà degli ambienti e dei problemi da risolvere rendono i videogiochi un campo perfetto per la ricerca in ambito IA.

L'IA nei videogiochi si basa su percezione e decision-making all'interno di un ambiente di gioco. Le sfide presenti in questo campo sono diverse: lo spazio degli stati può assumere dimensioni notevoli, specialmente in giochi strategici; inoltre, apprendere una policy che prenda le giuste decisioni in ambienti con dinamiche sconosciute è un'operazione non semplice; infine, una sfida notevole è quella di riuscire a trasferire l'abilità di una IA da un gioco ad un altro.

Per molto tempo, queste sfide sono state risolte tramite Reinforcement Learning (RL). Negli ultimi anni, abbiamo assistito allo sviluppo del Deep Learning (DL), capace di ottenere risultati notevoli in ambiti quali la computer vision ed il natural language processing. La combinazione di RL e DL ha portato alla nascita del Deep Reinforcement Learning (DRL), che ha permesso di allenare agenti in spazi ad alta dimensionalità, incrementando drasticamente la generalizzazione e scalabilità degli algoritmi di RL classici. In particolare, in ambito videogiochi, il DRL ha fatto notevoli progressi, grazie anche allo sviluppo di piattaforme di benchmark sia generali (e.g. ALE, OpenAI Gym, ML-Agents) che specifiche di un videogioco (e.g. ViZDoom, StarCraft, Dota 2).

1.2. Obiettivi e contributi

L'obiettivo di questo lavoro di tesi è l'implementazione del Deep Reinforcement Learning nell'ambito dei videogiochi platform 2D, con particolare attenzione alla distinzione tra platform a livelli ed endless runner (cioè senza fine). Ciò che si vuole provare è come, a partire da videogiochi già esistenti, sia possibile implementare il Deep Reinforcement Learning e la semplicità (o meno) nel farlo. Inoltre, si vuole dimostrare la differenza sostanziale che c'è tra una rete allenata con scopo di

superamento di un livello rispetto alla generalizzazione ed adattamento necessari per un ambiente endless.

Lo scopo terminale è quello di allenare delle reti neurali artificiali il più possibile performanti.

1.3. Struttura della tesi

Nel capitolo 2 viene fatto un excursus su cosa è il Deep Reinforcement Learning, le tipologie di algoritmi presenti nello stato dell'arte e viene introdotto ML-Agents. Nel capitolo 3 vengono spiegate le motivazioni che hanno portato alla scelta dei casi studio, le metriche utilizzate ed i casi studio stessi. Nel capitolo 4 vengono mostrati gli esperimenti effettuati ed i risultati ottenuti. Infine, il capitolo 5 prevede delle note conclusive sul lavoro effettuato e dei possibili sviluppi futuri.

Capitolo 2.

Stato dell'arte

2.1. Introduzione al Deep Reinforcement Learning

Il Deep Reinforcement Learning (DRL) si basa sull'utilizzo complementare del Deep Learning e del Reinforcement Learning.

2.1.1. Deep Learning

Il Deep Learning (DL) è una metodologia di apprendimento automatico basata su reti neurali artificiali ed è utilizzato per apprendere delle funzioni matematiche che siano in grado di approssimare la rappresentazione dei dati su cui esse vengono applicate. L'apprendimento può essere di tre tipi: supervisionato, non supervisionato e semi-supervisionato. Il DL è stato introdotto per la prima volta nel 1986 [1], senza però poter essere applicato, per mancanza di dati e di potenza computazionale sufficiente; questi due problemi sono stati superati negli anni e ad oggi il DL è ampiamente utilizzato ed in continua evoluzione.

Le classi di reti neurali deep più importanti sono: la Convolutional Neural Network (CNN), applicata ampiamente nella computer vision, la Recurrent Neural Network (RNN), molto utilizzata nei problemi di natural language processing, e la Long Short Term Memory (LSTM), che è un tipo particolare di RNN, capace di apprendere dipendenze a lungo termine.

Il DL ha trovato spazio in molti campi e continua anche attualmente ad essere oggetto di ricerca.

2.1.2. Reinforcement Learning

Il Reinforcement Learning (RL) è una metodologia di machine learning che si basa sull'utilizzo di uno o più agenti, i quali, inseriti all'interno di un ambiente (environment), cercano di apprendere una politica (policy) ottimale attraverso un processo di trial and error, ottenendo ricompense (reward) sulla base di azioni intraprese per passare da uno stato al successivo [2].

Si differenzia dal Deep Learning in quanto non si occupa di trovare rappresentazioni all'interno di insiemi di dati (etichettati o meno), ma cerca una traiettoria di apprendimento che vada a massimizzare i reward ricevuti.

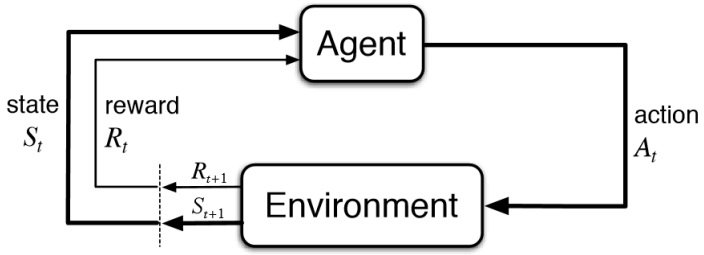


Figura 2.1.: Interazione tra agente ed environment nel RL[1].

È basato sul processo decisionale di Markov (MDP) ad episodi con fattore di sconto (S,A,γ,P,r): l'agente sceglie un'azione a_t seguendo una policy $\pi(a_t|s_t)$ nello stato s_t. L'environment riceve l'azione, produce un reward r_{t+1} e transita allo stato successivo s_{t+1} seguendo la probabilità di transizione $P(s_{t+1}|s_t, a_t)$. Il processo continua fino a che l'agente raggiunge uno stato terminale o un tempo massimo stabilito.

L'obiettivo è quello di massimizzare i reward cumulativi scontati previsti

$$\mathbb{E}_\pi[R_t] = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i r_{t+1}], \tag{2.1}$$

dove $\gamma \in (0, 1]$ è il fattore di sconto.

Gli algoritmi di RL possono dividersi in off-policy e on-policy: negli algoritmi off-policy la policy utilizzata per selezionare le azioni è diversa dalla policy di apprendimento, in quelli on-policy le due coincidono. Inoltre, possiamo anche categorizzare gli algoritmi di RL in algoritmi basati sul valore, sulla policy o sul modello: nel primo caso gli agenti aggiornano la funzione di valore per apprendere una policy appropriata, nel secondo gli agenti apprendono la policy direttamente, nel terzo esiste (o viene creato) un modello dell'environment da cui partire. Quest'ultima distinzione verrà approfondita nella sezione 2.2.

Il Q-learning è un tipico metodo off-policy; la sua regola di aggiornamento è

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma arg\max_a Q(s_{t+1}, a) - Q(s_t, a_t)), \tag{2.2}$$

dove α rappresenta il tasso di apprendimento (learning rate).

Lo State–action–reward–state–action (SARSA) è un tipico algoritmo on-policy; la sua regola di aggiornamento è

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)), \tag{2.3}$$

dove α rappresenta il learning rate.

2.1.3. Deep Reinforcement Learning

Quando lo spazio degli stati è ad alta dimensionalità gli algoritmi di RL tradizionali non sono efficaci; gli algoritmi di Deep Reinforcement Learning incorporano il DL

per risolvere questo tipo di problemi, rappresentando la policy $\pi(a|s)$ (o un'altra funzione di apprendimento) come una rete neurale.

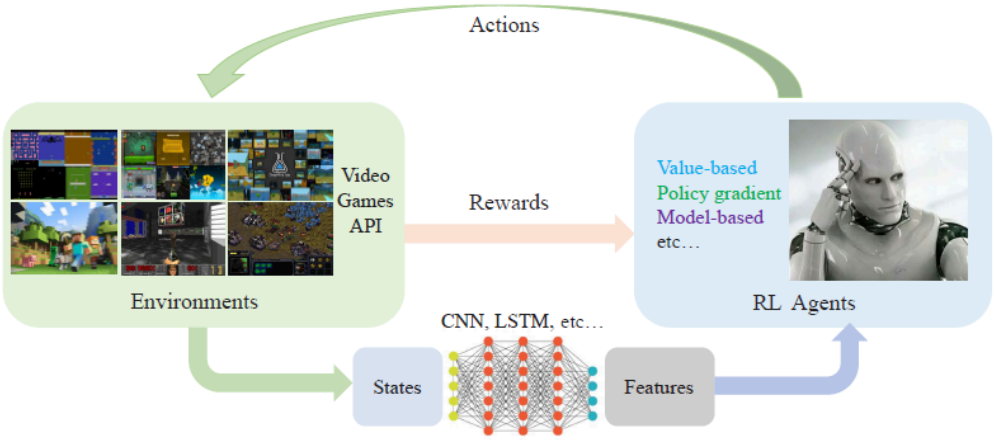


Figura 2.2.: Tipica struttura di un framework di DRL per un videogioco [3].

Questi concetti verranno approfonditi nelle sezioni successive.

2.2. Lavori recenti

Nell'ambito del Deep Reinforcement Learning possiamo distinguere 3 categorie di metodi principali:

2.2.1. Metodi di DRL basati sul valore

Gli algoritmi di DRL basati sul valore apprendono sulla base dello stato o della coppia stato-azione. Agiscono seguendo la migliore azione nello stato. Il più famoso è il Deep Q-network (DQN) [4], il quale riceve pixel raw in input ed invia in output una funzione di valore per stimare i reward futuri.

Alcuni recenti algoritmi di questo tipo includono: Rainbow [5], che combina 6 degli algoritmi basati sul DQN, mostrando come ognuno di essi apporti un contributo al miglioramento della perfomance generale; RUDDER [6], utilizzato per trattare MDPs finiti con reward ritardati, che propone due metodi per portare i reward futuri verso lo zero, cercando, in questo modo, di rimuovere i problemi di bias ed alta varianza; Ape-X DQfD [7], che utilizza un nuovo operatore di Bellman trasformato per processare reward di densità e scale variabili, una loss ausiliaria per estendere l'orizzonte di pianificazione e delle dimostrazioni umane per guidare l'agente verso stati con reward maggiori; Soft DQN [8], che è una versione del Q-learning per ambienti regolarizzati dall'entropia.

2.2.2. Metodi di DRL basati sul modello

Gli algoritmi di DRL basati sui modelli creano un modello dell'environment e lo utilizzano per apprendere.

Alcuni recenti algoritmi questo tipo includono: World Model [9], che allena una RNN generativa tramite apprendimento non supervisionato per modellare ambienti di RL attraverso rappresentazioni spazio-temporali compresse. Le feature estratte vengono inserite in policy semplici e compatte ed allenate tramite evoluzione; Value Propagation (VProp) [10], che si basa sull'iterazione del valore ed è un insieme di moduli di pianificazione differenziale efficienti; MuZero [11], che è un algoritmo che combina la ricerca basata su alberi con un modello già allenato; quando applicato iterativamente riesce a predire reward, policy di selezione dell'azione e la funzione di valore.

2.2.3. Metodi di DRL basati sulla policy

Gli algoritmi di DRL basati sulla policy (chiamati anche policy gradient) [12] apprendono direttamente la funzione di policy che mappa lo stato all'azione ed agiscono sulla base della policy migliore, cercando sempre di migliorarla. Alla base, il policy gradient parametrizza la policy ed aggiorna il parametro θ. Nella sua forma generale, la funzione obiettivo del policy gradient è

$$J(\theta) = \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \log \pi_{\theta}(a_t|s_t)R], \tag{2.4}$$

dove R è il ritorno totale accumulato.

L'algoritmo Actor-Critic [2] migliora il policy gradient con una valutazione basata sul valore

$$J(\theta) = \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \Psi_t \log \pi_{\theta}(a_t|s_t)], \tag{2.5}$$

dove Ψ_t è il "Critic", cioè la parte che si occupa di stimare la funzione di valore; questo può essere la funzione di valore stato-azione $Q^{\pi}(s_t, a_t)$, la funzione di vantaggio $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ (algoritmo A2C) o l'errore di differenza temporale $r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$. L'"Actor" aggiorna la distribuzione della policy nella direzione suggerita dal Critic.

Asynchronous Advantage Actor-Critic (A3C) [13] migliora l'Actor-Critic implementando il training parallelo, tramite il quale multipli agenti in environment paralleli aggiornano una funzione di valore globale, aumentando così l'efficienza e l'efficacia dell'esplorazione dello spazio degli stati.

Trust Region Policy Optimization (TRPO) [14] è un algoritmo di ottimizzazione delle policy che garantisce miglioramenti monotoni. Utilizza l'Actor-Critic nella sua versione A2C. TRPO computa una direzione di ascesa per migliorare il policy

gradient e questo può assicurare piccoli cambiamenti nella distribuzione della policy. Il problema di ottimizzazione con vincoli del TRPO ad ogni epoca è

$$\max_{\theta} \quad E_{s\sim\rho_{\theta'},a\sim\pi_{\theta'}}[\frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)}A_{\theta'}(s|a)], \tag{2.6a}$$

$$\text{subject to:} \quad E_{s\sim\rho_{\theta'}}[D_{KL}(\pi_{\theta'}(\cdot|s))] \leq \delta_{KL}. \tag{2.6b}$$

Proximal Policy Optimization (PPO) [15] migliora TRPO, effettuando il clipping della funzione obiettivo. TRPO non limita la distanza tra le policy $\frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)}$ (l'attuale e la vecchia); questo può portare a instabilità nell'allenamento. PPO utilizza un valore di soglia ε per limitare il rateo di probabilità di modifica della funzione obiettivo. È formulato come

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \tag{2.7a}$$

$$L(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t], \tag{2.7b}$$

dove:

- θ è il parametro della policy
- $\hat{\mathbb{E}}_t$ rappresenta il reward atteso per uno step temporale
- r_t è il rateo di probabilità tra la nuova e la vecchia policy
- $\hat{A}_t$ è il vantaggio stimato al tempo t
- ε è l'iperparametro di clipping

PPO verrà citato nuovamente nelle sezioni successive, in quanto è l'algoritmo utilizzato negli esperimenti.

2.3. Deep Reinforcement Learning nei videogiochi

Negli ultimi anni, nell'ambito dei videogiochi è nato un interesse sempre maggiore verso il DRL. Sono state create svariate piattaforme di allenamento e benchmark nel tempo; in particolare, possono essere divise in piattaforme generali e piattaforme specifiche (per un singolo videogioco). Nelle piattaforme generali troviamo:

- Arcade Learning Environment (ALE)[16], che include una moltitudine di videogiochi dell'Atari 2600
- OpenAI Gym[17], che contiene una serie di videogiochi dell'Atari, giochi da tavolo, robot 2D e 3D
- OpenAI Universe[2], anch'esso contenente una vasta gamma di videogiochi

[2] `https://github.com/openai/universe`

- OpenAI Gym Retro[3], che estende Gym con videogiochi NEC, Nintendo e Sega
- DeepMind Lab[18], che è un ambiente di apprendimento in prima persona
- Unity ML-Agents Toolkit[19], che è uno strumento per creare ed integragire con ambienti simulati. Questa è la piattaforma utilizzata per gli esperimenti

Ognuna di queste piattaforme, nonostante non siano specifiche per un singolo videogioco, si specializza in una categoria o piattaforma differente; ad esempio, ALE tratta solo i giochi dell'Atari 2600, DeepMind Lab i giochi in prima persona, Unity ML-Agents Toolkit tutti quelli creati in Unity; la versatilità di quest'ultima piattaforma rispetto alle altre è anche il motivo per cui è stata scelta per gli esperimenti.

Nelle piattaforme specifiche troviamo:

- videogiochi con prospettiva in prima persona:
  - Malmo[4], che presenta un ambiente basato sul videogioco Minecraft
  - TORCS[20], che è un simulatore di corse
  - ViZDoom[21], che è una piattaforma basata sullo sparatutto in prima persona Doom
- videogiochi strategici in tempo reale (RTS):
  - TorchCraft[22], piattaforma di Facebook per StarCraft I
  - DeepMind[23], che presenta un'ambiente per l'allenamento di reti in StarCraft II
- Google Research Football[24], che è un ambiente basato sul videogioco open-source Gameplay Football

Queste piattaforme si focalizzano su un solo videogioco, con il vantaggio di poter creare un ambiente di training il più specifico e personalizzato possibile per quel determinato gioco; dall'altro lato, però, non offrono la possibilità di scelta e generalizzazione fornita da piattaforme come OpenAI Gym o Unity ML-Agents Toolkit.

2.4. Unity Machine Learning Agents Toolkit

Unity Machine Learning Agents Toolkit (ML-Agents)[5][19] è una piattaforma open-source di Unity[6] che ha lo scopo di abilitare videogiochi e simulazioni ad essere utilizzati come ambienti per allenare agenti intelligenti. Fornisce, tramite delle API Python, implementazioni di alcuni algoritmi dello stato dell'arte nel Deep Reinforcement Learning (PPO [15], SAC, MA-POCA, self-play) per l'allenamento di un singolo agente, multi-agente cooperativo e multi-agente competitivo.

[3] `https://github.com/openai/retro`
[4] `https://github.com/Microsoft/malmo`
[5] `https://github.com/Unity-Technologies/ml-agents`
[6] `https://unity.com/`

2.4.1. Proximal Policy Optimization in ML-Agents

Proximal Policy Optimization è una delle tecniche di Deep Reinforcement Learning implementate in ML-Agents. PPO utilizza una rete neurale per approssimare una funzione ideale che mappi le osservazioni di un agente con la migliore azione che l'agente può effettuare in un determinato stato. L'algoritmo è implementato in PyTorch[7] ed è eseguito in un processo Python separato, comunicando con Unity tramite un socket.

2.4.2. Il ciclo di RL in ML-Agents

Un agente è un'entità che può osservare il suo environment, decidere la migliore azione da intraprendere sulla base di queste osservazioni ed eseguirla. Gli agenti in Unity sono creati estendendo la classe `Agent`. Il ciclo di apprendimento di un agente in ML-Agents è composto da 4 elementi Fig.2.3:

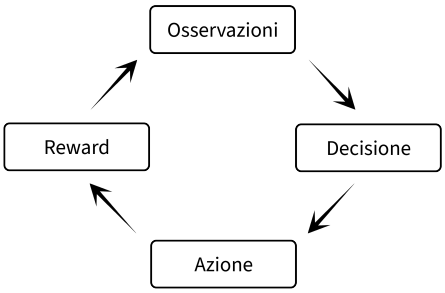


Figura 2.3.: Ciclo di apprendimento tramite Deep Reinforcement Learning in Unity.

2.4.2.1. Osservazioni

Le osservazioni rappresentano le informazioni che un agente ha dell'ambiente. In ML-Agents esistono 4 tipi di osservazioni:

- Osservazioni vettoriali: sono delle liste di `float` che contengono le variabili che l'agente dovrebbe conoscere (e.g. la posizione dell'agente, la posizione dell'obiettivo da raggiungere). Possono essere implementate effettuando l'override del metodo `Agent.CollectObservations()` ed utilizzando il metodo `VectorSensor.AddObservation()` per aggiungere un'osservazione. È presente anche un parametro, *Stacked Vectors*, che indica il numero di stati da memorizzare prima di inviarli alla rete neurale. Può essere impostato con un valore ≥ 1, dove il numero indica la quantità di stati da memorizzare (di default è 1, cioè nessuna "memoria").
- Osservazioni visuali: utilizzano un oggetto `CameraSensor` per catturare le immagini e trasformarle in un tensore 3D che viene elaborato dalla CNN della policy. Questo è il tipo di osservazioni meno efficiente e più lento nell'allenamento.

[7] `https://pytorch.org/`

- Osservazioni raycast: sono dei raggi emessi a partire dalla posizione dell'agente per permettergli di osservare ciò che lo circonda. Possono essere implementati aggiungendo il GameObject `RayPerceptionSensorComponent3D` per ambienti 3D e `RayPerceptionSensorComponent2D` per ambienti 2D Fig.2.4 come componente dell'agente.

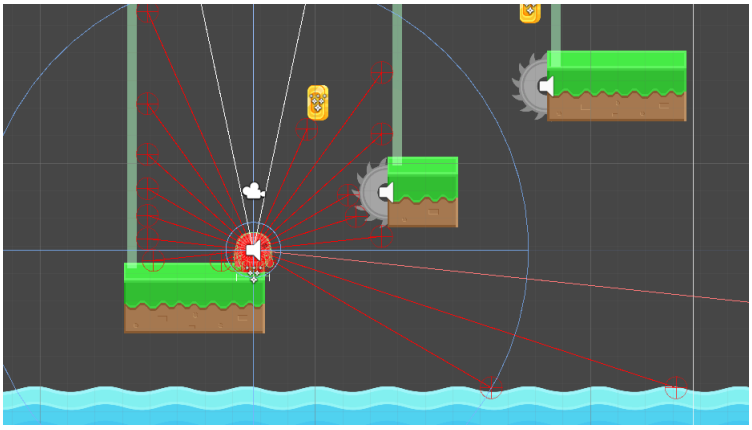


Figura 2.4.: Esempio di osservazioni raycast 2D (raggi rossi) nel gioco Red Runner

  È possibile personalizzare vari parametri del sensore, tra cui gli oggetti da considerare e da ignorare, il numero di raggi, la direzione e la lunghezza. Come nelle osservazioni vettoriali, anche qui è presente un parametro, *Stacked Raycasts*, che permette di memorizzare più stati prima di inviarli alla rete.

- Osservazioni a griglia: combinano le osservazioni visuali con quelle raycast; in particolare, il sensore utilizza una griglia per dividere l'ambiente in cubi ed ha una visione 2D dall'alto dell'agente. Durante il movimento dell'agente, il sensore rileva la presenza di oggetti nelle celle vicine a quelle dell'agente e passa le informazioni di ogni cella ad un tensore 3D che verrà elaborato dalla CNN della policy.

2.4.2.2. Decisione

La decisione viene presa da una policy a cui l'agente passa le osservazioni. Un agente richiederà una decisione ogni volta che `Agent.RequestDecision()` verrà richiamato, o, in caso le decisioni vengano prese ad intervalli regolari, è possibile inserire il GameObject `DecisionRequester` come componente dell'agente.

2.4.2.3. Azione

L'azione è un'istruzione eseguita da un agente sulla base della decisione presa dalla policy. È implementata tramite l'override del metodo `Agent.OnActionReceived()`. Né la policy né l'algoritmo di training conoscono il significato delle azioni; l'algoritmo semplicemente le prova ed osserva come cambiano i reward accumulati nel tempo e negli episodi.

Esistono 2 tipi di azioni in ML-Agents: continue e discrete. Le azioni continue sono un array di lunghezza personalizzabile che contengono valori `float`. Le azioni discrete sono degli array di interi di lunghezza personalizzabile; queste possono essere suddivise in *Branches*: ogni *Branch* è un array di interi e il *Branch* "padre" definisce il numero di possibili valori per ogni *Branch*. Questa suddivisione è utile in caso l'agente necessiti di effettuare diverse operazioni contemporaneamente (e.g. muoversi e saltare).

2.4.2.4. Reward

Un reward è una ricompensa data ad un agente per aver eseguito un'azione corretta (è anche possibile che un'azione non porti a nessun reward). I reward sono la metrica di riferimento utilizzata dall'algoritmo PPO per apprendere; in particolare, l'algoritmo opera ottimizzando le scelte prese in modo tale che un'agente guadagni il più alto reward cumulativo medio possibile nel tempo.

I reward possono distinguersi in reward estrinseci ed intrinseci: i primi sono i reward definiti dall'environment, cioè definiti esternamente all'algoritmo di apprendimento; i secondi sono definiti esternamente all'environment, sono quindi interni all'algoritmo stesso. Il reward cumulativo rappresenta la media delle due tipologie di reward. La logica di assegnazione dei reward estrinseci è definita dal programmatore ed è un elemento fondamentale per un'apprendimento corretto.

Per quanto concerne i reward instrinseci, all'interno di ML-Agents sono presenti diversi algoritmi per la loro assegnazione:

- Generative Adversarial Imitation Learning (GAIL)
- Behavioral Cloning (BC)
- Curiosity
- Random Network Distillation (RND)

I primi due utilizzano l'*Imitation Learning*, cioè basano il loro apprendimento sul cercare di replicare delle dimostrazioni, cioè delle azioni pre-registrate dal programmatore; in particolare, GAIL utilizza una rete generativa avversaria dove una seconda rete, il discriminatore, è allenato per distinguere le azioni delle dimostrazioni da quelle dell'agente; BC cerca di copiare esattamente le azioni nelle dimostrazioni.

Curiosity e RND sono due algoritmi che possono essere utilizzati in caso di reward scarsi, cioè casi in cui l'agente necessita di molta esplorazione prima di ricevere un reward.

2.4.3. Iperparametri in ML-Agents

Un altro dei processi più importanti per l'allenamento di una rete neurale è la scelta degli iperparametri; una corretta selezione di questi è fondamentale per creare una rete che soddisfi il task richiesto. In ML-Agents ne sono presenti molti; i più rilevanti sono:

- *Gamma*: rappresenta il fattore di sconto per i reward futuri
- *Lambda*: è il parametro utilizzato per calcolare il Generalized Advantage Estimate (GAE) [25] (che fa parte del PPO), il quale indica quanto un agente sia dipendente dalla stima del valore corrente rispetto al reward corrente
- *Buffer size*: è la quantità di esperienza (osservazioni dell'agente, azioni e reward ottenuti) che dovrebbe essere acquisita prima di aggiornare il modello
- *Batch size*: è il numero di esperienze utilizzate per una iterazione dell'aggiornamento del grandient descent
- *Numero di epoche*: è il numero di passaggi attraverso il buffer dell'esperienza durante il gradient descent
- *Learning rate*: rappresenta l'intensità di ogni passo di aggiornamento del gradient descent
- *Beta*: corrisponde all'intensità della regolarizzazione dell'entropia, utilizzata per fare in modo che l'agente esplori propriamente lo spazio delle azioni durante l'allenamento
- *Espilon*: è il valore di soglia accettabile nella divergenza tra la vecchia e la nuova policy durante l'aggiornamento del gradient descent
- *Numero di layers*: è il numero di hidden layers presenti dopo le osservazioni in input (o dopo il CNN encoding delle osservazioni visuali)
- *Hidden Units*: rappresenta il numero di unità Fully Connected per livello della rete neurale
- *Normalize*: permette di scegliere se normalizzare le osservazioni vettoriali in input o meno

Capitolo 3.

Metodologia

3.1. Motivazioni scelta casi studio

Sono stati presi in analisi due videogiochi platform 2D come casi studio: Super Sparty Bros.[1] e Red Runner[2]. Sono state diverse le motivazioni che hanno portato alla scelta di questi due videogiochi: in primo luogo, si voleva sperimentare l'applicazione del Deep Reinforcement Learning in videogiochi già realizzati, per carpirne la facilità (o meno) di adattamento ad esso. In un contesto di diffusione del Deep Reinfocement Learning in questo ambito, infatti, è importante pensare alla sua flessibilità di implementazione in prodotti già presenti sul mercato (o già in fase di sviluppo avanzato), piuttosto che in prodotti creati appositamente per questo task. Questa scelta ha portato alla luce anche la necessità di effettuare delle modifiche al codice ed alla struttura del gioco originale per renderli compatibili con il Deep Reinfocement Learning e velocizzare l'allenamento dell'agente. In secondo luogo, i due videogiochi sono stati scelti in modo da differenziarsi all'interno dello stesso genere: in particolare, Super Sparty Bros. è un platform 2D a livelli, mentre Red Runner è un platform 2D di tipo endless runner, cioè dove non esiste una vera e propria fine del livello ma si cerca di avanzare il più possibile, superando ostacoli in un ambiente generato proceduralmente, cioè differente ad ogni partita. L'obiettivo è di mostrare le differenze di allenamento delle reti neurali nei due giochi, evidenziando come un ambiente a livelli sia meno oneroso da allenare, da un punto di vista computazionale, rispetto ad uno endless.

3.2. Metriche di valutazione

Le metriche utilizzate per la valutazione dei risultati sono le seguenti:

- **Reward cumulativo**: come già discusso precedentemente, questa rappresenta la metrica di apprendimento dell'agente, che mette insieme reward estrinseci ed intrinseci. Ci si aspetta che aumenti nel tempo fino a stabilizzarsi a training completo

[1] `https://github.com/Zj-Lan/Unity_Platform-game`
[2] `https://github.com/BayatGames/RedRunner`

- **Policy loss**: rappresenta la media della loss della funzione di policy; rappresenta quanto la policy cambia negli episodi. Ci si aspetta che diminuisca durante il training
- **Value loss**: questa metrica rappresenta la media della loss dell'aggiornamento della funzione di valore; rappresenta quanto correttamente il modello riesce a predire il valore di ogni stato. Ci si aspetta che aumenti durante l'apprendimento e diminuisca quando il reward si stabilizza
- **Entropia**: stabilisce quanto casuali sono le decisioni. Ci si aspetta che decresca lentamente durante il training
- **Extrinsic reward**: rappresenta il valore medio dei soli reward estrinseci
- **Extrinsic value estimate**: rappresenta il valore medio stimato per tutti gli stati visitati dall'agente. Ci si aspetta che cresca durante l'apprendimento

3.3. Caso studio: Super Sparty Bros.

Super Sparty Bros. è un videogioco open-source sviluppato in Unity. È un platform 2D a scorrimento orizzontale a livelli, in cui un personaggio, controllato dal giocatore, deve spostarsi all'interno di un ambiente con l'obiettivo di raggiungere e toccare un oggetto (una rosa), collezionando le monete lungo il percorso. Sono presenti dei nemici (sia statici che in movimento) da evitare e dei punti di caduta letali. Toccare i nemici o cadere in uno dei punti provoca un game over istantaneo. Gli input che il giocatore può dare sono: movimento a destra, movimento a sinistra e salto (con possibilità di doppio salto).

Figura 3.1.: Screenshot di Super Sparty Bros.

Sono state apportate delle modifiche alla struttura originale del gioco per adattarla all'utilizzo con ML-Agents e per migliorare l'allenamento stesso, oltre che sistemare dei bug riscontrati; in particolare, inizialmente, è stato rimosso l'aumento di punteggio per le monete prese (per evitare errori dovuti a punteggi troppo elevati durante il training), è stato rimosso il sistema di vite (inutile ai fini dell'allenamento) ed è stato fatto in modo che il personaggio non effettuasse il passaggio al livello successivo in

caso di vittoria, ma si resettasse nello stesso livello (per poter allenare l'agente nello stesso livello in automatico). In più è stato fixato il sistema di *groundCheck* (cioè se il personaggio collide terreno o meno), in quanto inconsistente. Infine, dato che le monete sono la fonte di reward principale (oltre il reward finale per il raggiungimento dell'obiettivo) ne sono state rimosse alcune che influivano negativamente sul comportamento dell'agente, in particolare portavano l'agente a compiere un percorso non necessario, sviandolo dall'obiettivo.

Successivamente, data la necessità di gestire l'agente, il suo comportamento ed il processo di DRL è stata creata la classe `CharacterAgent.cs`, che va a definire tutta la logica di apprendimento, comprese osservazioni, azioni e reward. La classe è stata inserita come componente dell'oggetto rappresentante il personaggio.

Per quanto riguarda le osservazioni, sono state utilizzate osservazioni vettoriali ed osservazioni raycast. Le osservazioni vettoriali sono 7; si compongono di:

- le componenti (*x, y, z*) relative alla posizione del personaggio (`Vector3 localPosition()`),
- le componenti (*x, y, z*) relative alla posizione della rosa (l'oggetto da raggiungere)
- un `int` rappresentante il numero di salti disponibili (utilizzato per aiutare l'agente nell'apprendimento del doppio salto).

Per le osservazioni raycast, è stato usato un `RayPerceptionSensorComponent2D()` con 7 tag (che si riferiscono agli oggetti a cui il sensore è interessato):

- le monete,
- la zona di caduta,
- il terreno,
- le piattaforme fisse,
- le piattaforme mobili,
- i nemici.

Sono stati utilizzati 20 raggi con un ampiezza massima di 180°, una lunghezza pari a 20 ed un raggio di grandezza delle sfere a fine raggio di 0.15.

Per quanto concerne le azioni, ne sono state utilizzate solo di discrete; in particolare, sono stati inseriti 2 *branch* da 3 azioni l'uno: il primo rappresenta il movimento dell'agente (0 = fermo, 1 = movimento a destra, -1 = movimento a sinistra), il secondo lo stato del salto (0 = non sta saltando, 1 = il pulsante del salto è premuto, 2 = il pulsante del salto è rilasciato, quest'ultimo resosi necessario a causa della logica del gioco stesso). Inoltre, è stata modificata la classe già presente che gestisce movimento e salto per renderla compatibile con l'utilizzo di queste azioni (`CharacterController2D.cs`).

Per quanto riguarda i reward, è stato attribuito:

- +0.1 per ogni moneta presa,
- +5 per il raggiungimento dell'obiettivo,
- -1 per game over (per contatto con nemici o caduta),
- -0.0005 (-1/2000) ad ogni step, come penalità esistenziale per stimolare il movimento dell'agente (per evitare che l'agente decida che stare fermo è l'azione migliore da prendere) (2000 è il numero massimo di passi prima del reset).

Inoltre, l'environment viene resettato dopo 2000 passi.

Per richiedere le decisioni è stato inserito un `DecisionRequester` tra i componenti dell'agente, che permette di prendere decisioni ad intervalli regolari (in questo caso ad ogni passo).

Infine, è stato effettuato l'override della funzione `Agent.Heuristic()`, implementando l'input manuale di azioni per finalità di testing.

3.4. Caso studio: Red Runner

Il secondo videogioco preso in analisi ed utilizzato per il task di apprendimento tramite DRL con ML-Agents è Red Runner. Anch'esso è un gioco open-source e sviluppato in Unity. Condivide con Super Sparty Bros. la caratteristica di essere un platform 2D a scorrimento orizzontale; ciò che lo differenzia è il fatto di far parte del genere "endless runner", cioè videogiochi in cui lo scopo è proseguire il più possibile senza perdere, non c'è, quindi, uno stato finale di vittoria. I livelli sono generati proceduralmente (cioè costruiti randomicamente ad ogni giocata), creando così una sfida sempre diversa. La tipologia di gioco è sembrata molto interessante per un task di DRL, in quanto, a differenza di un platform a livelli, qui la struttura dell'ambiente cambia ad ogni partita, richiedendo alla rete una maggiore capacità di generalizzazione. Sono presenti diversi ostacoli che il personaggio deve superare o evitare: ad esempio acqua, spuntoni, lame (fisse e mobili), oggetti che cadono e così via. Sono presenti delle monete collezionabili, ma ciò che più conta è la distanza percorsa prima di perdere.

Il gioco è strutturato in modo da avere 17 "blocchi", che rappresentano ognuno una possibile parte del percorso; il primo è fisso ed è il punto di partenza, gli altri 16 sono scelti casualmente per essere aggiunti al percorso mano a mano che il personaggio prosegue nella sua corsa. Ogni blocco ha la stessa probabilità di essere selezionato e può essere selezionato più volte (idealmente anche sempre).

Anche in questo caso sono state effettuate delle modifiche ed aggiunte al codice di gioco per renderlo idoneo al DRL con ML-Agents: per prima cosa, è stato fatto in modo che, al lancio, il gioco avviasse direttamente una nuova partita (senza passare per il menu principale); anche in caso di morte, è stato fatto in modo che non venisse mostrata alcuna schermata di game over ma si tornasse direttamente al punto di partenza.

Figura 3.2.: Screenshot di Red Runner

Ciò che è stato inserito nel percorso è un sistema di checkpoint, sfruttato per dare dei reward all'agente Fig.3.3.

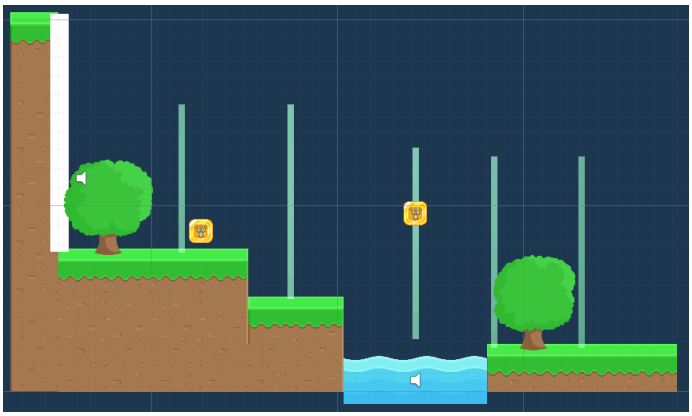


Figura 3.3.: Esempio di checkpoint (rettangoli verdi) presenti in un blocco

Più nel dettaglio, sono stati creati, in ogni blocco, dei GameObject rettangolari e ad ognuno è stata aggiunta la classe `CheckpointSingle.cs` come componente, creata con lo scopo di rilevare il contatto con l'agente Alg.1 Fig.3.4.

Tutti i checkpoint di un blocco sono stati raggruppati all'interno di un GameObject `Checkpoints`. La classe `CheckpointSingle.cs` comunica con un'altra (`TrackCheckpoints.cs`, anche questa creata appositamente), la quale si occupa di tenere traccia dei checkpoint che sono stati oltrepassati e si accorge se l'agente commette un errore (e.g. torna indietro e prende un checkpoint già preso). Quest'ultima

Algorithm 1 CheckpointSingle.cs

```
1: if collisione tra checkpoint e giocatore then
2:     richiama metodo della classe TrackCheckpoints GiocatoreAttraversaCheck-
   point (passando come parametro il checkpoint)
3: end if
```

classe è inserita come componente del blocco stesso Alg.2 Fig.3.5.

Algorithm 2 TrackCheckpoints.cs

```
1: procedure ISTANZIAZIONE CLASSE:
2:     cerca componente "Checkpoints"
3:     crea lista ordinata dei checkpoint presenti in "Checkpoints": checkpointsList
4:     imposta indiceProssimoCheckpoint = 0
5: end procedure
6: procedure GIOCATOREATTRAVERSACHECKPOINT(checkpoint):
7:     if indice checkpointsList(checkpoint) == indiceProssimoCheckpoint then
8:         invoca evento PassaggioCheckpointCorretto
9:     else
10:        invoca evento PassaggioCheckpointErrato
11:    end if
12: end procedure
```

Infine, data la necessità di gestire l'agente ed il processo di DRL, compresa l'assegnazione di reward (o penalità) in caso di passaggio per un checkpoint giusto (o sbagliato), è stata creata una terza classe, `RedAgent.cs`, inserita come componente dell'agente Alg.3 Fig.3.6.

Algorithm 3 RedAgent.cs

```
1: ...
2: procedure PASSAGGIOCHECKPOINTCORRETTO
3:     assegna reward positivo
4: end procedure
5: procedure PASSAGGIOCHECKPOINTERRATO
6:     assegna reward negativo (penalità)
7: end procedure
8: ...
```

Per quanto riguarda le osservazioni, sono state utilizzate osservazioni vettoriali ed osservazioni raycast. Le osservazioni vettoriali sono 4: le componenti (x, y, z) relative alla posizione dell'agente (`Vector3 localPosition()`) e la velocità sull'asse x dell'agente (`float x`).

Per le osservazioni raycast, è stato usato un `RayPerceptionSensorComponent2D()` con 8 tag:

- il terreno,
- l'acqua,

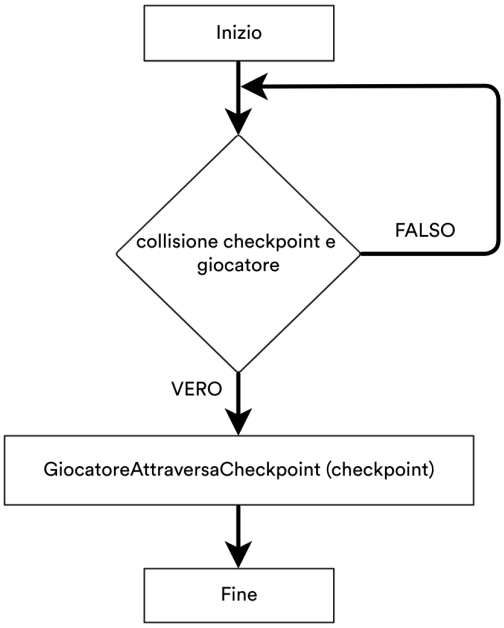


Figura 3.4.: Diagramma di flusso `CheckpointSingle.cs`

- gli spuntoni,
- gli oggetti dall'alto,
- le lame circolari,
- i checkpoint,
- le monete,
- dei particolari punti che generano game over, inseriti da me per facilitare l'allenamento.

I valori dei parametri raycast differiscono rispetto a quelli utilizzati in Super Sparty Bros. poiché l'ambiente è diverso; in particolare, sono stati utilizzati 15 raggi (diminuiti rispetto ai 20 dell'altro gioco poiché gli ostacoli qui risultano essere più grandi e, quindi, bastano meno raggi per rilevarli) con un ampiezza massima di 180°, una lunghezza pari a 69 (lunghezza aumentata poiché l'ambiente è più grande) ed un raggio di grandezza delle sfere a fine raggio di 0.4 (anche in questo caso aumentato perchè gli ostacoli sono di dimensioni maggiori).

Per quanto concerne le azioni, ne sono state utilizzate 1 continua ed un *branch* da 3 per le discrete; in particolare, quella continua è un `float` tra -1 e 1 che indica la direzione di movimento dell'agente (0 = fermo, > 0 = movimento a destra, < 0 = movimento a sinistra), quella discreta rappresenta lo stato del salto (0 = non sta saltando, 1 = salto). Inoltre, ho modificato la classe `RedCharacter.cs` per renderla compatibile con l'utilizzo di queste azioni.

Per quanto riguarda i reward, è stato attribuito:

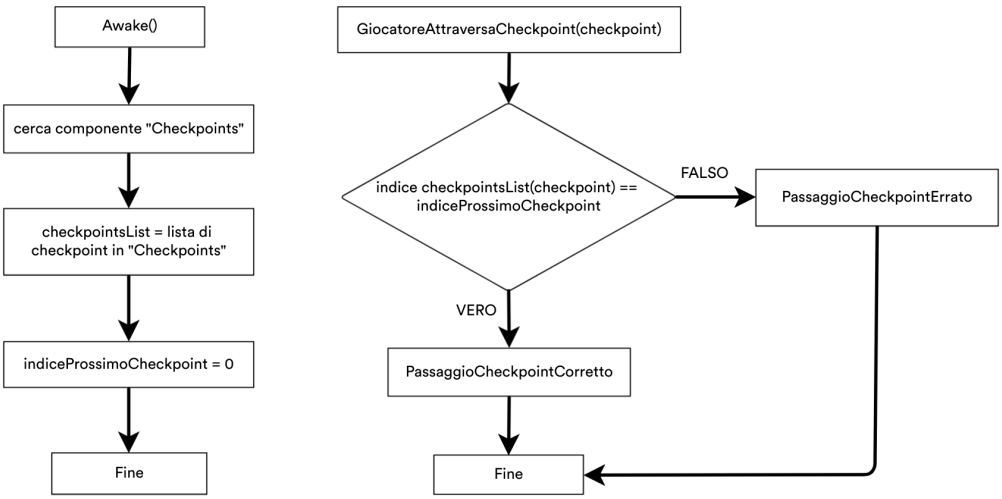


Figura 3.5.: Diagramma di flusso `TrackCheckpoints.cs`

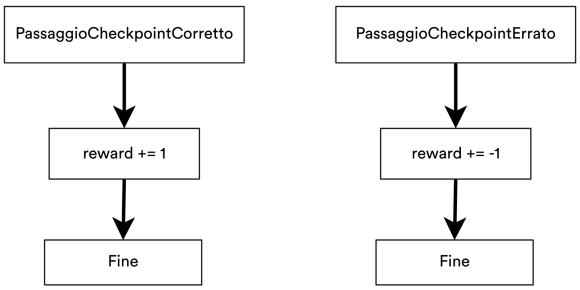


Figura 3.6.: Diagramma di flusso `RedAgent.cs` (solo gestione passaggio di checkpoint)

- +1 per ogni checkpoint corretto passato,
- -1 per passaggio su checkpoint sbagliato (da notare che dopo l'assegnazione della penalità viene generato un game over),
- -1 per collisione con ostacoli,
- -0.0005 (-1/2000) ad ogni step, come penalità esistenziale per stimolare il movimento dell'agente (2000 è il numero massimo di passi prima del reset),
- -0.1 per salto (in modo da stimolare l'agente a saltare solo quando necessario, poiché alcuni ostacoli possono essere superati solo acquisendo una certa velocità, che aumenta correndo senza saltare)

Inoltre, l'environment viene resettato se l'agente non attraversa un checkpoint entro 2000 passi.

Per richiedere le decisioni è stato inserito un `DecisionRequester` tra i componenti dell'agente.

Infine, anche qui è stato effettuato l'override della funzione `Agent.Heuristic()`, implementando l'input manuale di azioni per finalità di testing.

Capitolo 4.

Risultati e discussioni

4.1. Configurazioni reti

Con lo scopo di ricercare la combinazione migliore di iperparametri della rete sono stati effettuati diversi esperimenti. Gli iperparametri degli esperimenti sono riassunti nelle sezioni 4.1.1 e 4.1.2; i risultati ottenuti per ogni test sono presentati nella sezione 4.2.

In entrambi i casi gli iperparametri di *Learning rate*, *Beta* ed *Epsilon* hanno uno scheduling lineare, cioè descrescono linearmente durante il processo di training a partire dal valore impostato. Inoltre, sono stati impostati *Stacked Vectors* a 2 e *Stacked Raycasts* a 1.

4.1.1. Super Sparty Bros.

I test effettuati sul videogioco Super Sparty Bros. sono stati 11.
Gli iperparametri costanti in ogni esperimento sono stati:

- *Lambda*: 0.95
- *Normalize*: `true`

Nelle tabelle 4.1 e 4.2 sono riportati gli iperparametri utilizzati per gli esperimenti.

Test	1	2	3	4	5	6
Batch size	128	512	128	128	512	512
Buffer size	1280	5120	2560	2560	5120	5120
Learning rate	$3 \cdot 10^{-4}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$
Beta	0.005	0.005	0.01	0.0001	0.001	0.001
Epsilon	0.2	0.2	0.3	0.3	0.3	0.3
Epoche	3	3	3	3	3	3
Hidden units	128	256	256	256	256	256
Layers	2	3	5	4	4	4

Tabella 4.1.: Tabella configurazioni iperparametri Super Sparty Bros. (1)

Test	7	8	9	10	11
Batch size	512	512	1024	1024	512
Buffer size	5120	5120	10240	10240	5120
Learning rate	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$3 \cdot 10^{-5}$
Beta	0.001	0.001	0.001	0.001	0.001
Epsilon	0.3	0.3	0.4	0.4	0.3
Epoche	3	3	25	25	3
Hidden units	256	256	256	256	256
Layers	4	4	10	10	4

Tabella 4.2.: Tabella configurazioni iperparametri Super Sparty Bros. (2)

4.1.2. Red Runner

I test effettuati sul videogioco Red Runner sono stati 14.
Gli iperparametri costanti in ogni esperimento sono stati:

- *Batch size*: 512
- *Buffer size*: 5120
- *Lambda*: 0.95
- *Normalize*: `true`

Nelle tabelle 4.3 e 4.4 sono riportati gli iperparametri utilizzati per ogni esperimento.

Test	1	2	3	4	5	6
Learning rate	$3 \cdot 10^{-5}$	$3 \cdot 10^{-6}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$
Beta	0.005	0.005	0.005	0.0001	0.01	0.01
Epsilon	0.2	0.2	0.2	0.2	0.2	0.3
Epoche	3	3	3	3	3	3
Hidden units	256	256	256	256	256	256
Layers	2	2	2	2	2	2

Tabella 4.3.: Tabella configurazioni iperparametri Red Runner (1); test 4 e 5 effettuati a partire dalla rete trainata dalla configurazione 3; test 6 effettuato a partire dalla rete con configurazione 5

Test	7	8	9	10
Learning rate	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$
Beta	0.01	0.01	0.01	0.01
Epsilon	0.3	0.3	0.3	0.3
Epoche	3	3	3	3
Hidden units	256	256	256	512
Layers	4	4	4	5

Tabella 4.4.: Tabella configurazioni iperparametri Red Runner (2)

Test	11	12	13	14
Learning rate	$3 \cdot 10^{-5}$	$3 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$3 \cdot 10^{-5}$
Beta	0.01	0.01	0.01	0.01
Epsilon	0.3	0.3	0.3	0.3
Epoche	5	5	5	3
Hidden units	512	512	256	512
Layers	5	5	10	10

Tabella 4.5.: Tabella configurazioni iperparametri Red Runner (3)

4.2. Risultati

Vengono di seguito riportati i risultati ottenuti dagli esperimenti effettuati. Per la scelta dei risultati migliori sono state prese in considerazione le metriche di reward cumulativo, policy loss, value loss, entropia ed extrinsic value estimate. L'extrinsic reward non è stato considerato poiché negli esperimenti non sono stati utilizzati reward intrinseci, in quanto non si prestavano alle casistiche prese in considerazione. Quest'ultima metrica corrisponde, quindi, al reward cumulativo (non viene perciò replicata all'interno dei grafici che seguono in 4.2.1 e 4.2.2).

4.2.1. Super Sparty Bros.

Per quanto riguarda Super Sparty Bros., il gioco, inizialmente, non era affatto strutturato per supportare il DRL, presentando delle scelte di codice che hanno impattato sia sull'efficienza che sull'efficacia del training. Per una completa risoluzione di questa problematica sarebbe stato necessario riscrivere gran parte del codice di gioco, operazione che andava al di fuori dell'obiettivo degli esperimenti. Sono state, quindi, effettuate delle scelte nella creazione (e modifica) del codice per il DRL (descritte precedentemente e di cui viene riportato il codice in A), che fossero un compromesso tra la logica iniziale di funzionamento del gioco ed una corretta

implementazione, al fine di generare una rete neurale funzionante e che prenda le corrette decisioni.

Nella tabella 4.6 sono riassunti i risultati in termini di metriche finali.

Test	Reward cumulativo	Policy loss	Value loss	Entropia	Extrinsic value estimate
1	1.637	0.067	0.234	0.611	1.131
2	0.038	0.034	2.026	2.093	4.338
3	0.996	0.067	7.844	0.692	0.391
4	0.863	0.067	4.112	0.572	0.292
5	1.737	0.033	0.262	0.364	1.206
6	0.639	0.034	8.108	1.127	0.136
7	0.603	0.033	4.392	1.351	0.098
8	0.608	0.033	1.711	0.686	0.109
9	0.078	0.027	34.96	0.158	0.044
10	0.929	0.037	0.28	0.505	0.617
11	2.061	0.033	0.013	0.485	1.446

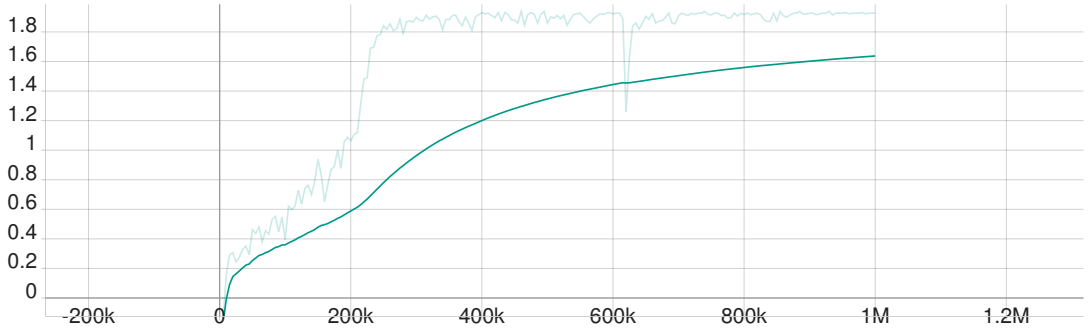
Tabella 4.6.: Tabella risultati metriche Super Sparty Bros.

I risultati migliori ottenuti per Super Sparty Bros. sono quelli del test 11 Fig.4.11; in particolare, si può notare come il secondo allenamento (in blu) presenti un reward cumulativo essenzialmente costante per tutti gli episodi, segno che la rete non ha possibilità di miglioramento ulteriore. Il reward cumulativo finale del secondo training ha un valore di 2.061 (valore più alto di tutti gli allenamenti effettuati). La correttezza dell'allenamento è confermata anche dalla policy loss e dalla value loss, i cui valori finali sono rispettivamente di 0.03338 e 0.01346, valori piuttosto bassi. Anche il valore di entropia finale è basso, attestando a 0.4845; l'extrinsic value estimate finale è 1.446, con un andamento in linea con il reward cumulativo.

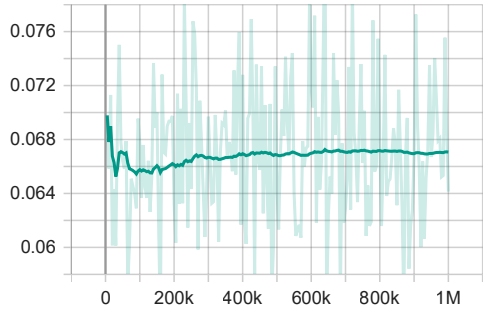
Viene riportata una dimostrazione video[1] che documenta il comportamento della rete. Come si può notare dal video, l'agente arriva all'obiettivo principale con una sua particolare strategia, cercando di prendere le monete presenti e di evitare i nemici (o saltar loro in testa per bloccarli temporaneamente).

Il tempo medio di allenamento è stato stimato in 3 ore e 15 minuti per 1 milione di steps, utilizzando una macchina con GPU NVIDIA Quadro P3200 e 32GB di RAM.

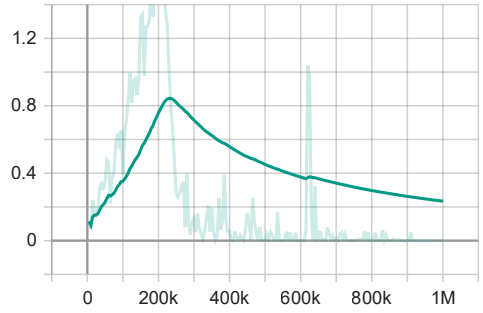
[1] `www.youtube.com/watch?v=SlU0ctOt6kU`



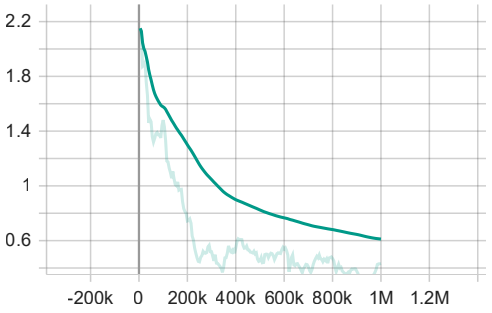
(a) Reward cumulativo



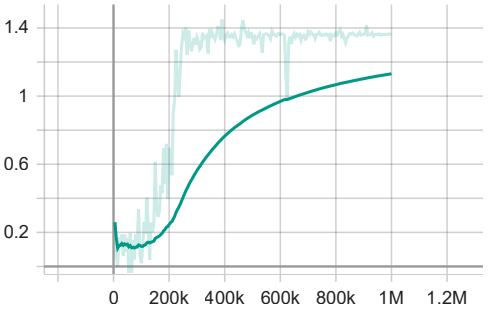
(b) Policy loss



(c) Value loss

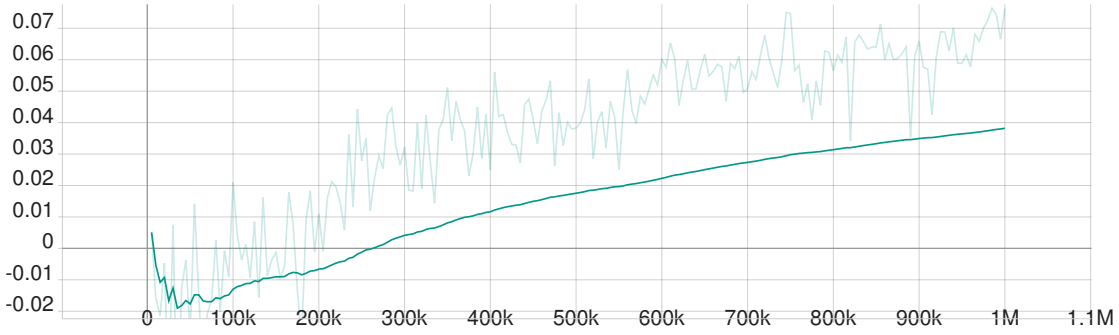


(d) Entropia

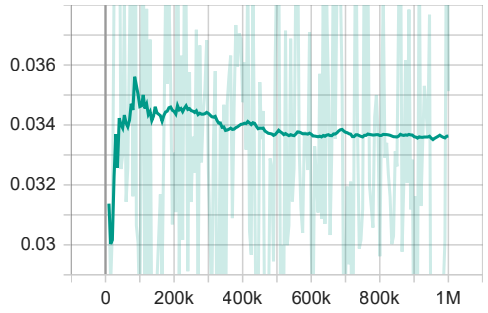


(e) Extrinsic value estimate

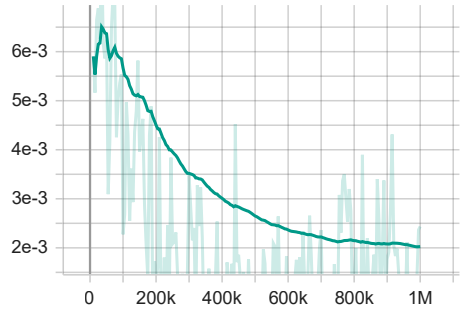
Figura 4.1.: Grafici test 1; numero passi: 1 milione



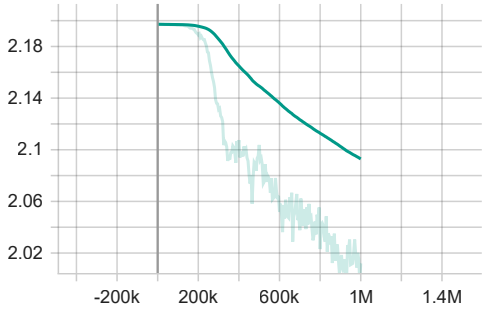
(a) Reward cumulativo



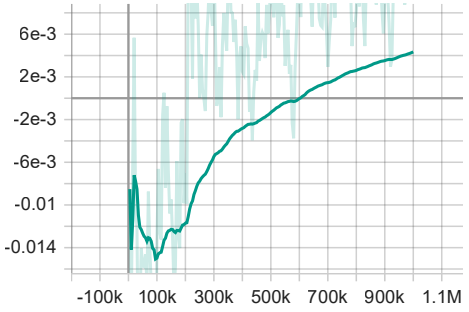
(b) Policy loss



(c) Value loss

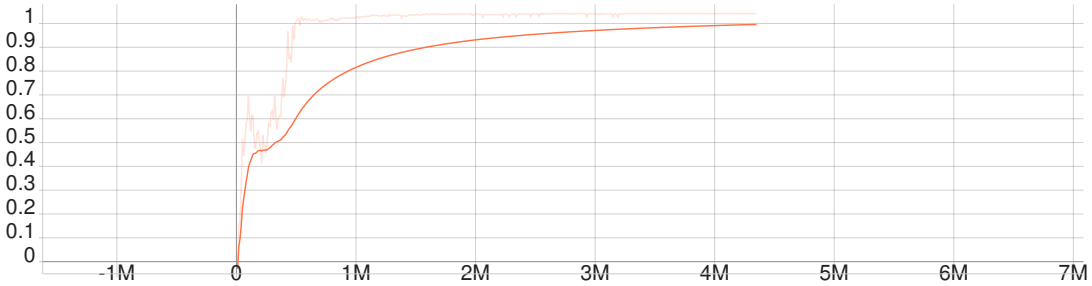


(d) Entropia

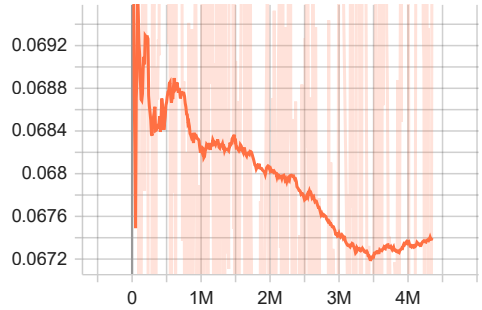


(e) Extrinsic value estimate

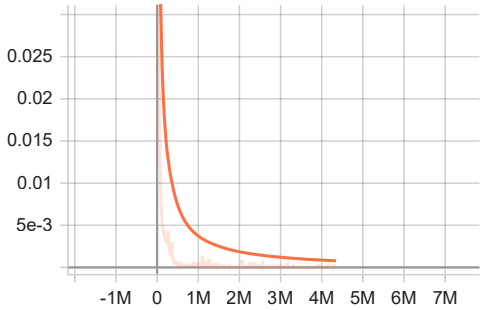
Figura 4.2.: Grafici test 2; numero passi: 1 milione



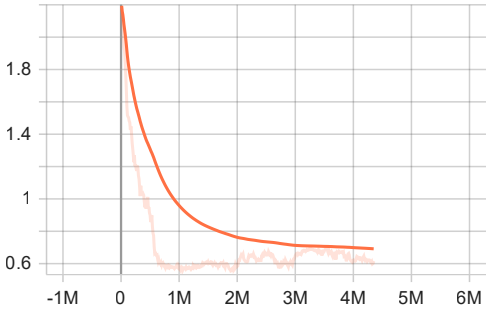
(a) Reward cumulativo



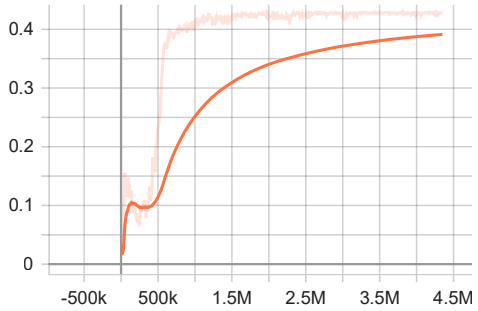
(b) Policy loss



(c) Value loss

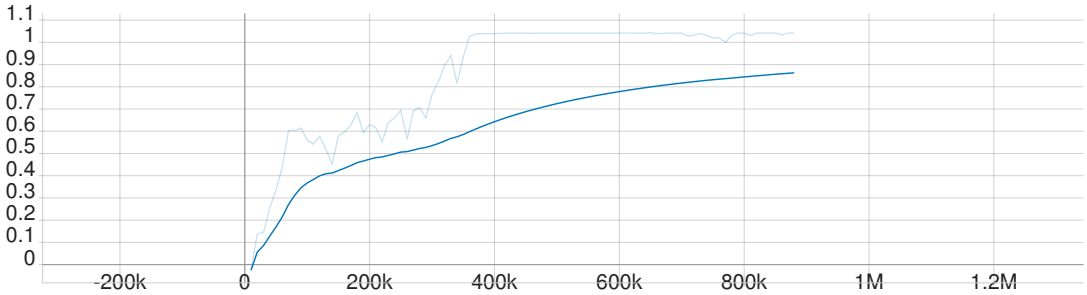


(d) Entropia

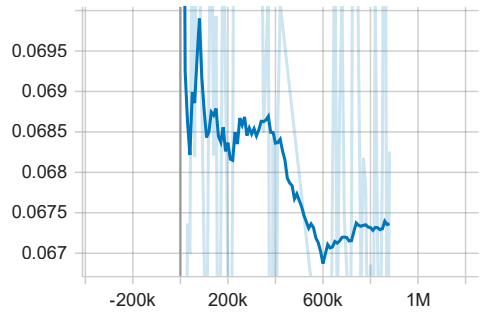


(e) Extrinsic value estimate

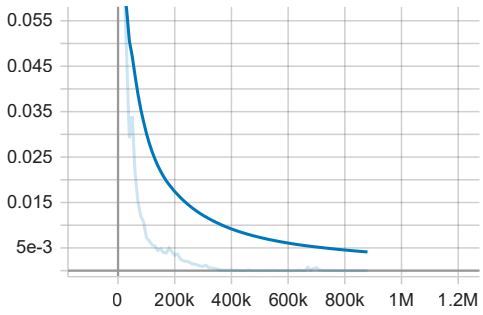
Figura 4.3.: Grafici test 3; numero passi: 4.35 milioni



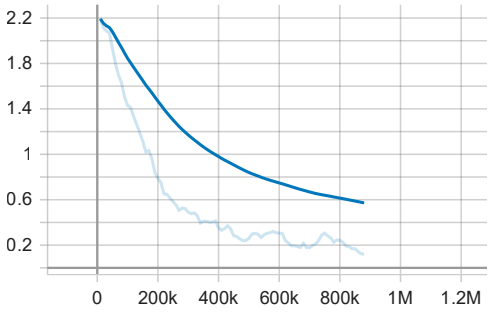
(a) Reward cumulativo



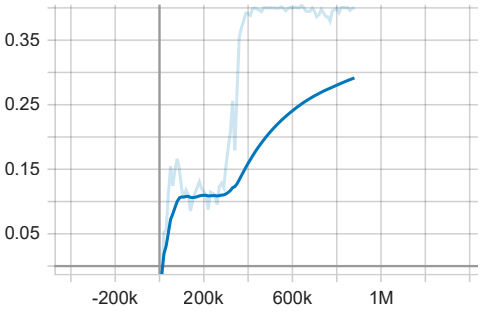
(b) Policy loss



(c) Value loss

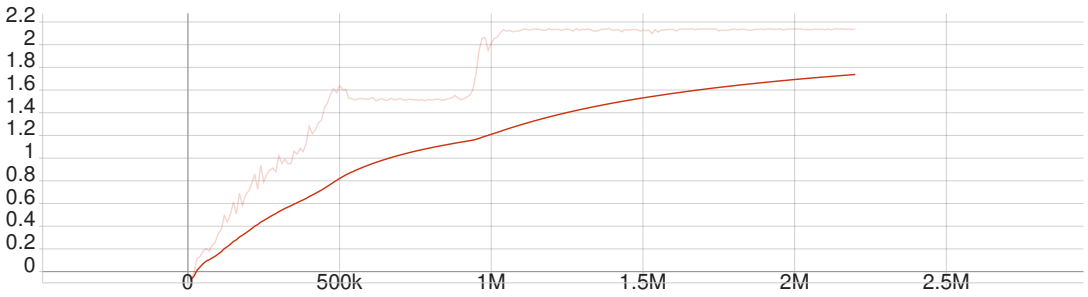


(d) Entropia

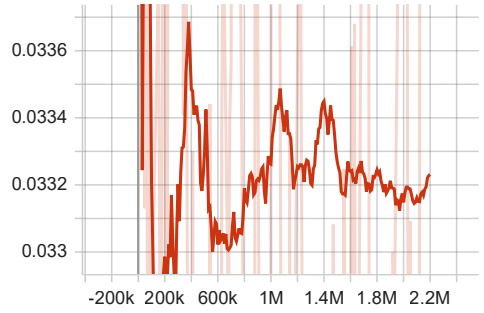


(e) Extrinsic value estimate

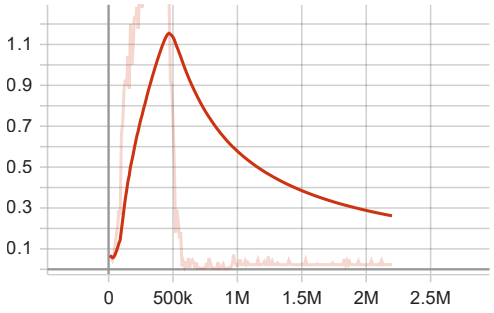
Figura 4.4.: Grafici test 4; numero passi: 880k



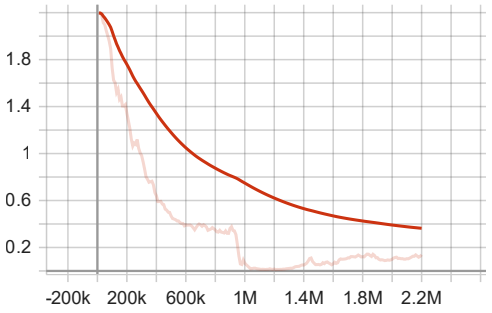
(a) Reward cumulativo



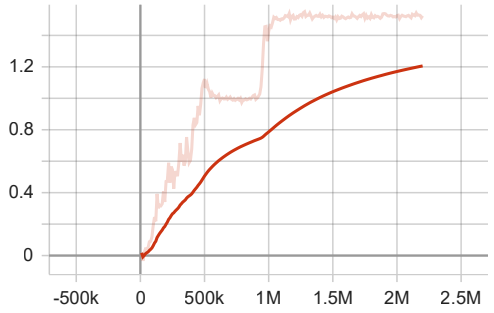
(b) Policy loss



(c) Value loss

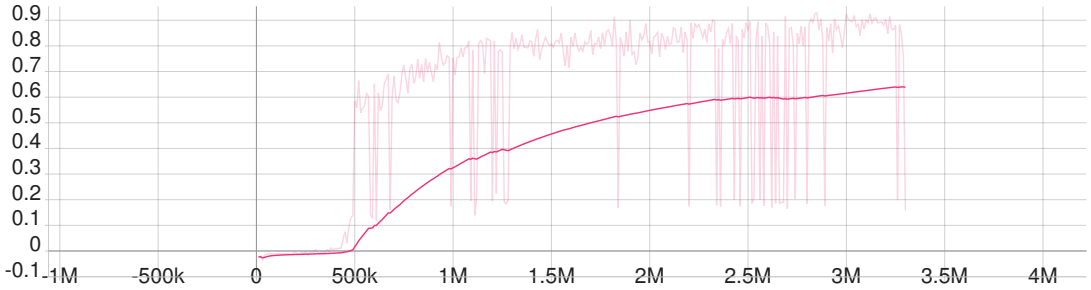


(d) Entropia

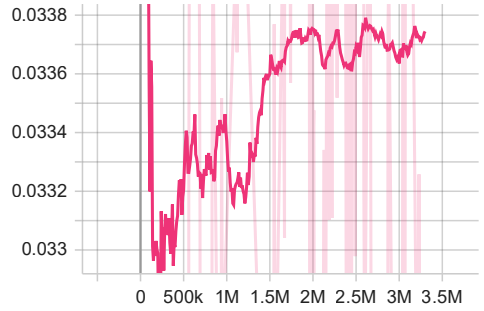


(e) Extrinsic value estimate

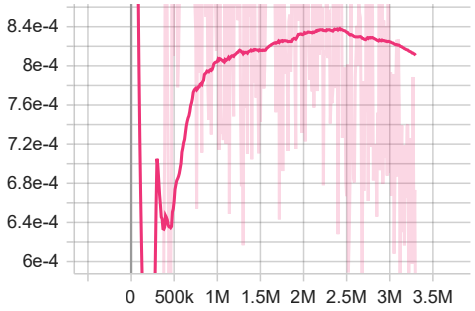
Figura 4.5.: Grafici test 5; numero passi: 2.2 milioni



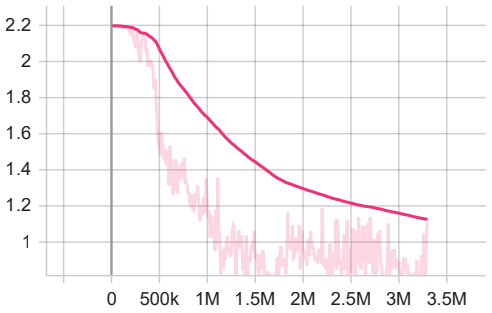
(a) Reward cumulativo



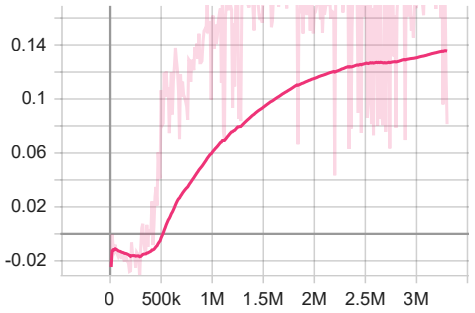
(b) Policy loss



(c) Value loss

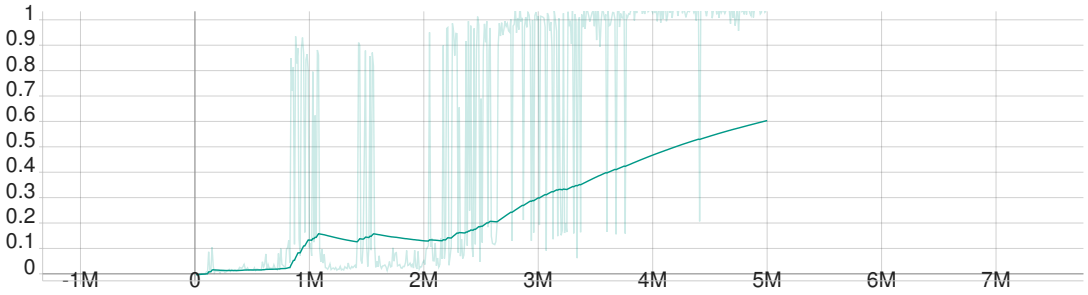


(d) Entropia

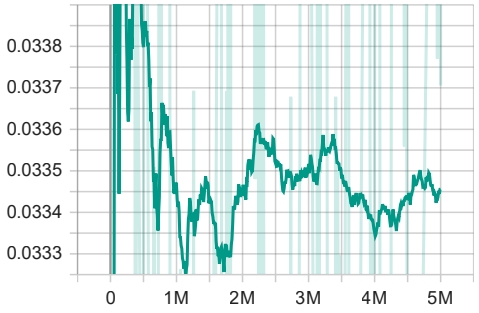


(e) Extrinsic value estimate

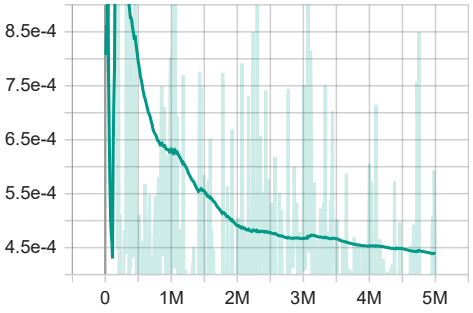
Figura 4.6.: Grafici test 6; numero passi: 3.3 milioni



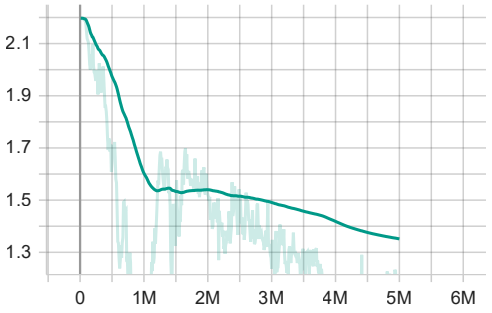
(a) Reward cumulativo



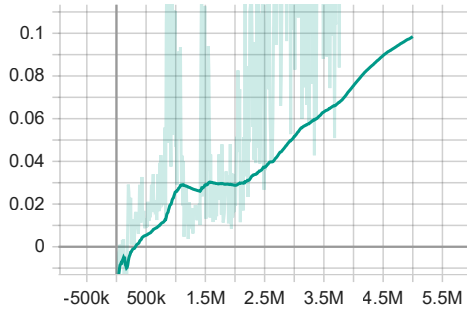
(b) Policy loss



(c) Value loss

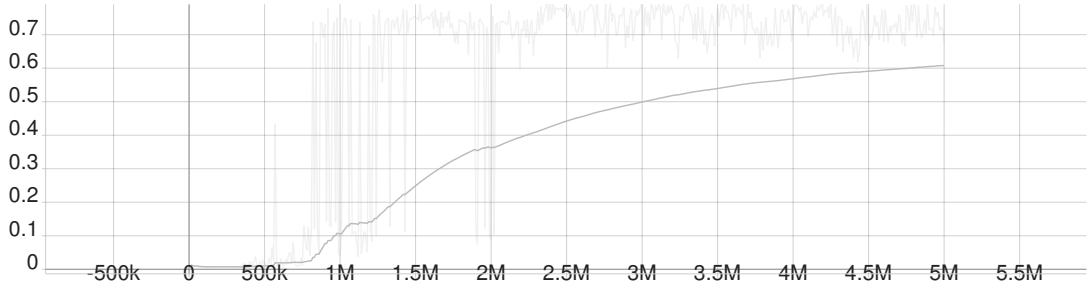


(d) Entropia

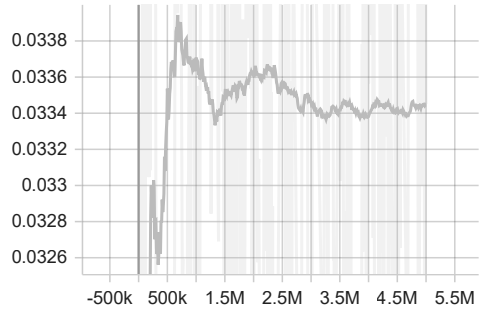


(e) Extrinsic value estimate

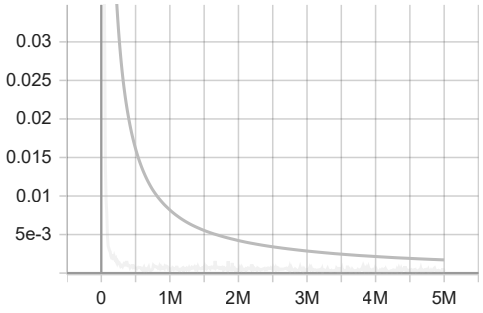
Figura 4.7.: Grafici test 7; numero passi: 5 milioni



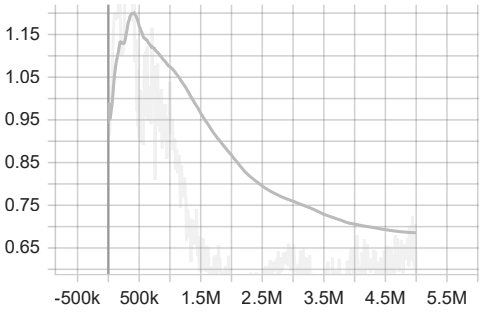
(a) Reward cumulativo



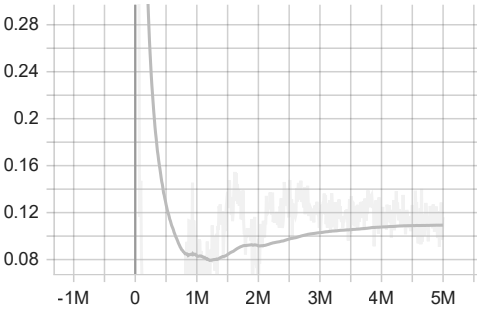
(b) Policy loss



(c) Value loss

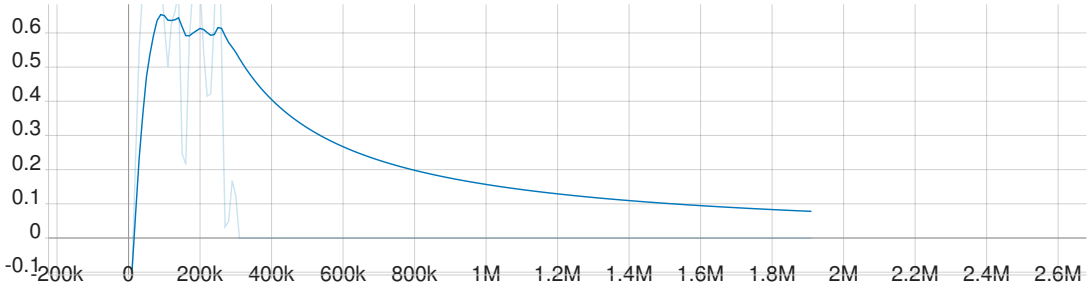


(d) Entropia

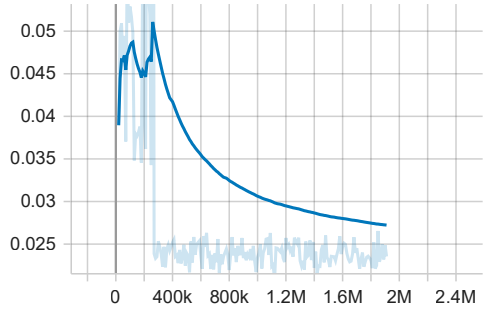


(e) Extrinsic value estimate

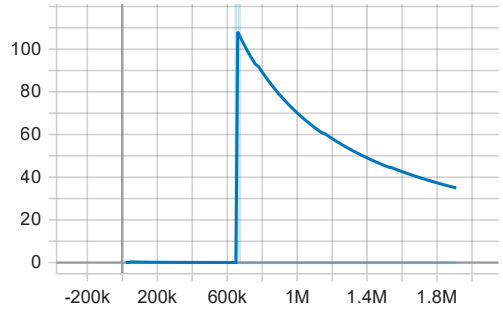
Figura 4.8.: Grafici test 8; numero passi: 5 milioni



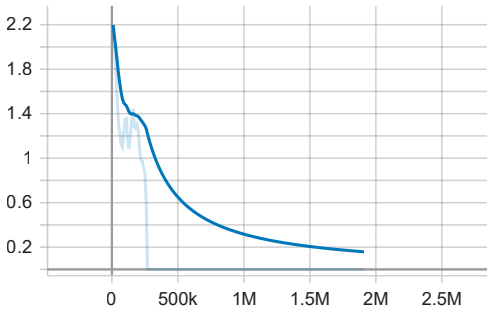
(a) Reward cumulativo



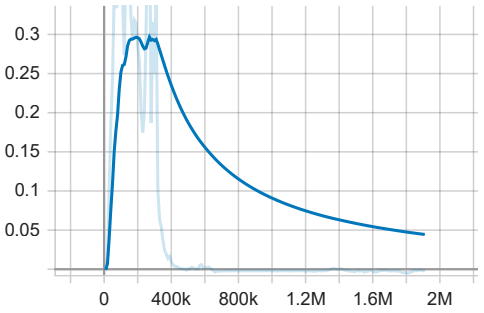
(b) Policy loss



(c) Value loss

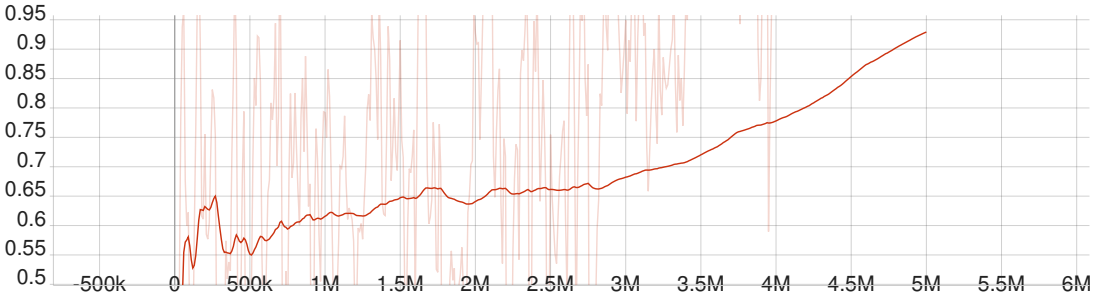


(d) Entropia

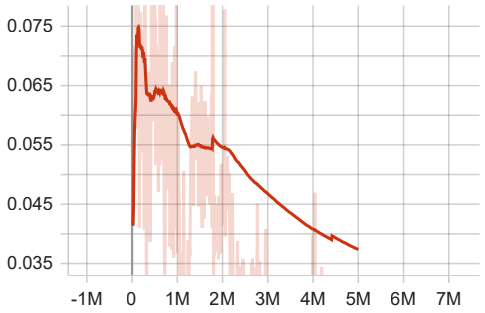


(e) Extrinsic value estimate

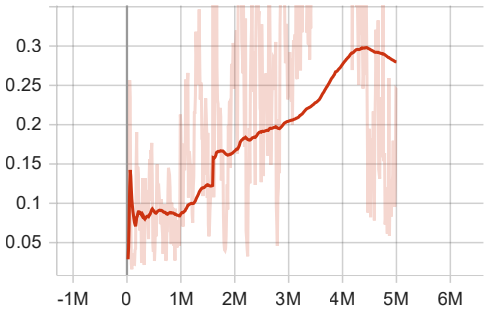
Figura 4.9.: Grafici test 9; numero passi: 1.91 milioni



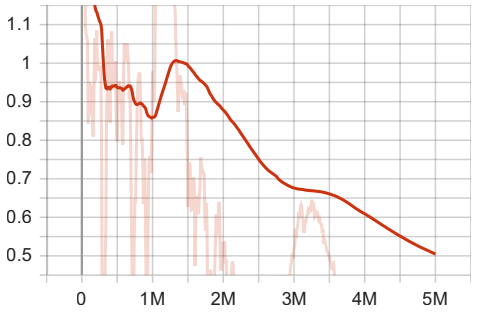
(a) Reward cumulativo



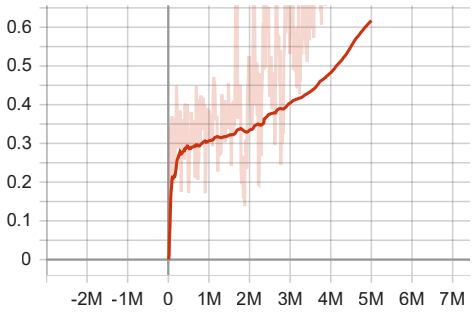
(b) Policy loss



(c) Value loss

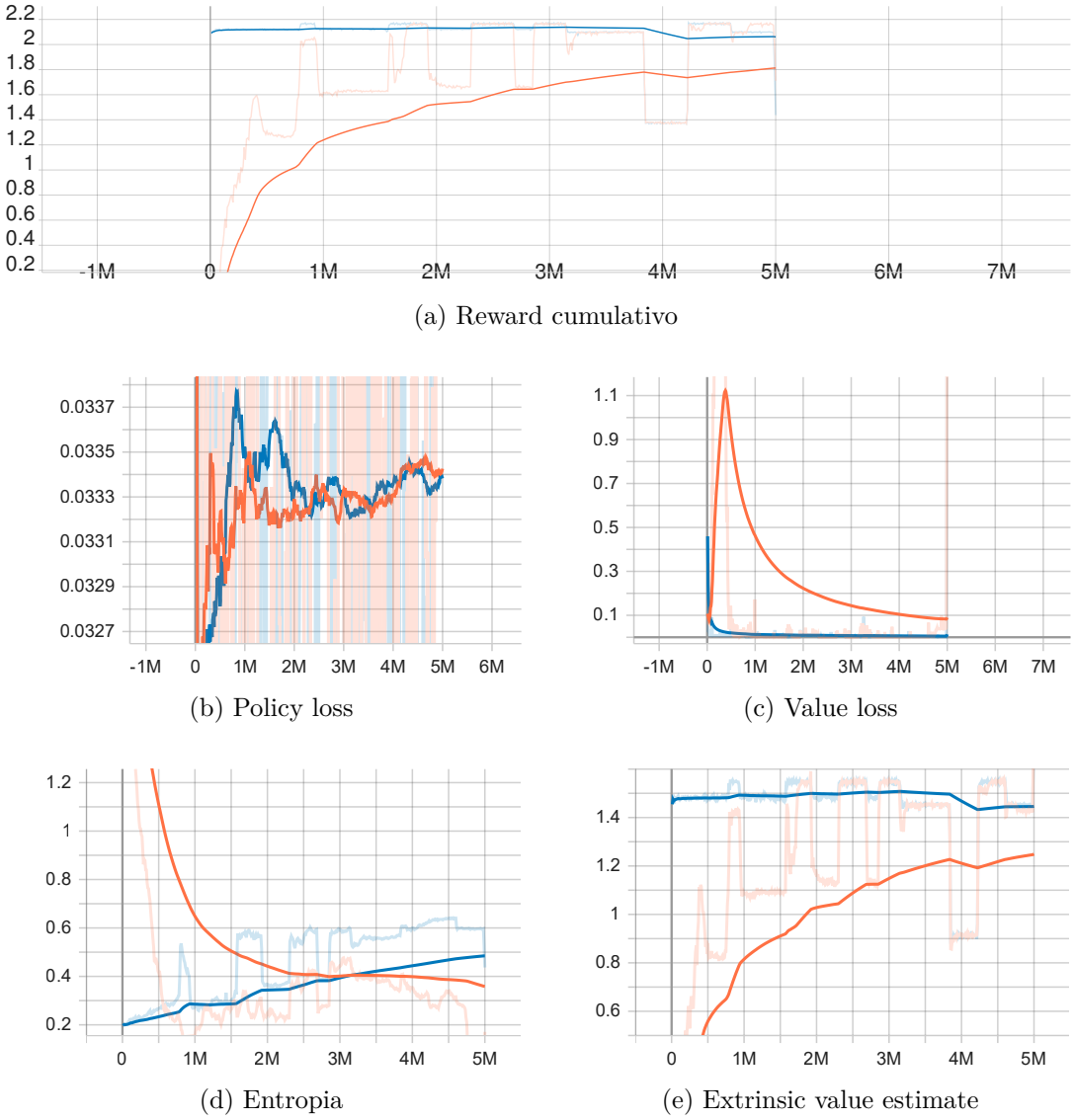


(d) Entropia



(e) Extrinsic value estimate

Figura 4.10.: Grafici test 10; numero passi: 5 milioni



(a) Reward cumulativo

(b) Policy loss

(c) Value loss

(d) Entropia

(e) Extrinsic value estimate

Figura 4.11.: Grafici test 11; numero passi: 5 milioni. Il secondo allenamento è realizzato a partire dalla rete precedente (non da zero)

4.2.2. Red Runner

Per quanto concerne Red Runner, l'allenamento è stato decisamente più complesso di Super Sparty Bros., non tanto per la struttura iniziale che il codice di gioco presentava (più articolata ma meglio realizzata di Super Sparty Bros.) ma quanto più per la natura stessa del gioco di presentare un ambiente generato proceduralmente (che cambia ad ogni partita), rendendo, quindi, la capacità di generalizzazione della rete una necessità.

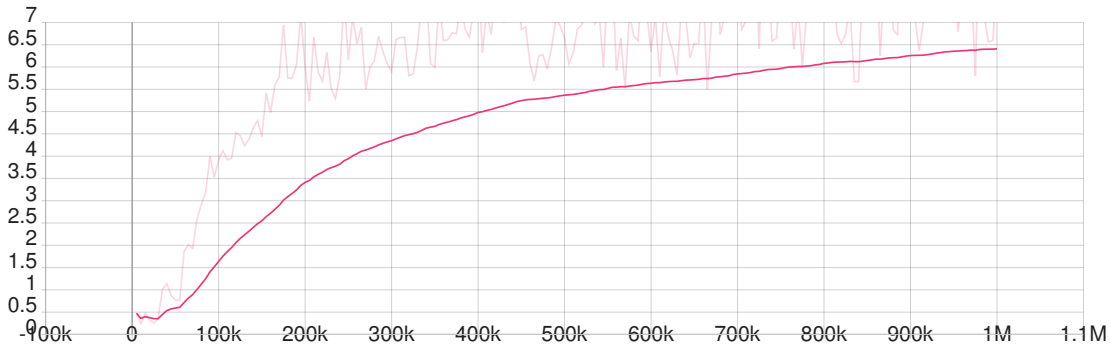
Nella tabella 4.7 sono riassunti i risultati in termini di metriche finali.

Test	Reward cumulativo	Policy loss	Value loss	Entropia	Extrinsic value estimate
1	6.407	0.033	1.834	2.47	3.65
2	1.6	0.033	0.379	2.518	0.787
3	22.87	0.034	10.13	2.285	11.6
4	22.96	0.033	10.22	2.246	11.63
5	23.52	0.034	10.34	2.302	11.73
6	23.75	0.033	10.31	2.298	11.75
7	18.71	0.033	3.913	2.086	5.754
8	15.2	0.034	2.338	2.136	3.764
9	22.9	0.034	8.25	2.119	9.854
10	22.24	0.033	4.632	2.123	6.102
11	9.454	0.033	0.371	2.02	0.957
12	-0.42	0.123	192.9	16.95	-0.09
13	-6.57	0.13	19.62	3.834	-2.78
14	10.83	0.034	1.529	1.946	3.176

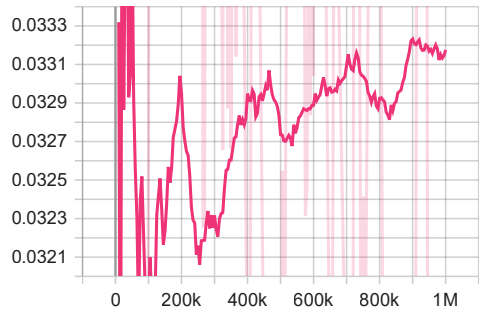
Tabella 4.7.: Tabella risultati metriche Red Runner

Data la complessità dell'ambiente, sono stati effettuati molti esperimenti, arrivando a trovare più di una configurazione soddisfacente. I test 9 (training 2) Fig.4.20 e 10 (training 3) Fig.4.21 risultano essere i migliori, con un reward cumulativo di 22.9 e 22.24, rispettivamente. In realtà, questi, come si può notare dai grafici in 4.2, non sono i valori più alti riscontrati tra tutti gli esperimenti effettuati: i valori del test 3 (training 6) Fig.4.14, del test 4 Fig.4.15, del test 5 Fig.4.16 e del test 6 Fig.4.17 superano i precedenti in termini di reward cumulativo. Il problema dei suddetti esperimenti è che nonostante la rete di partenza (test 3 e di conseguenza test 4, 5 e 6 che da essa derivano) si comporti bene durante l'allenamento, in fase di inferenza (utilizzo concreto della rete) mostra un comportamento errato.

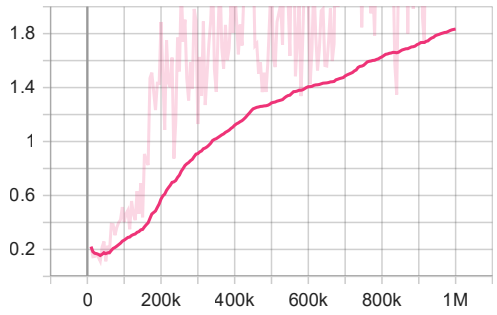
Il tempo medio di allenamento è stato stimato in 4 ore e 51 minuti per 1 milione di steps, utilizzando una macchina con GPU NVIDIA RTX 2070 SUPER e 16GB di RAM.



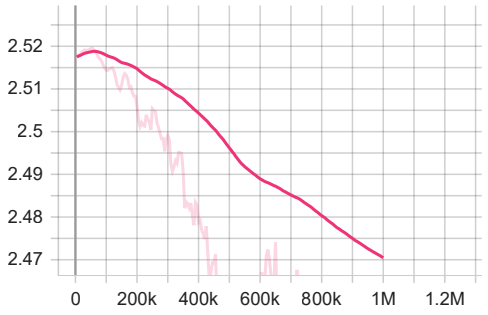
(a) Reward cumulativo



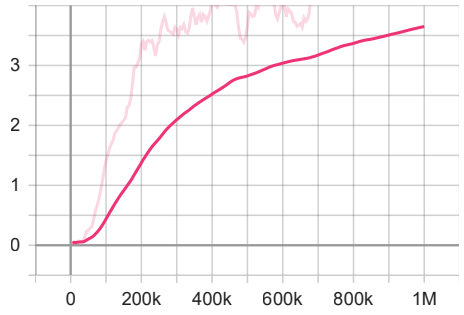
(b) Policy loss



(c) Value loss

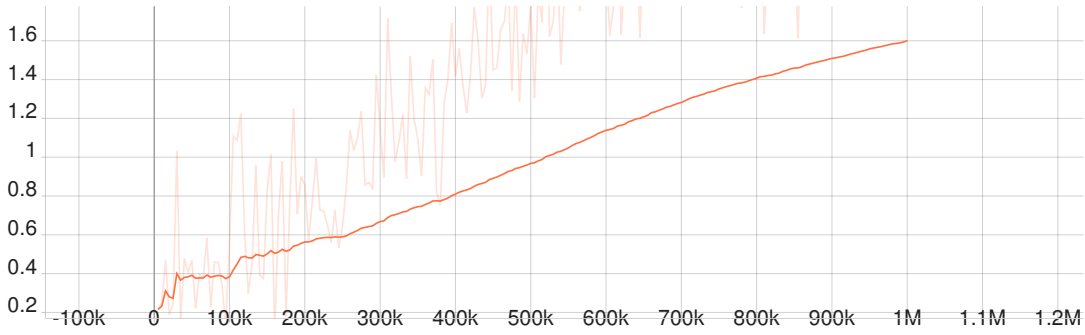


(d) Entropia

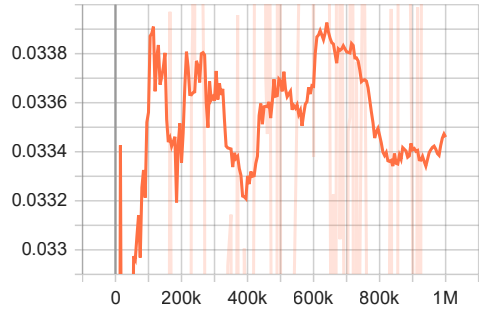


(e) Extrinsic value estimate

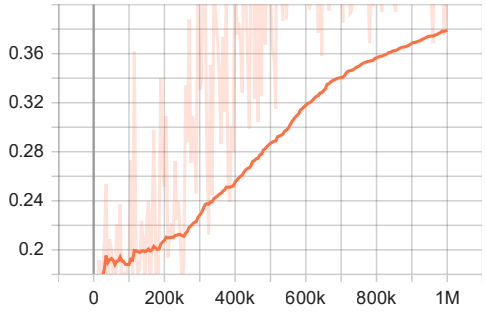
Figura 4.12.: Grafici test 1; numero passi: 1 milione



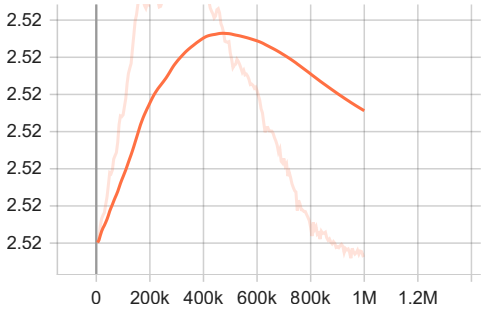
(a) Reward cumulativo



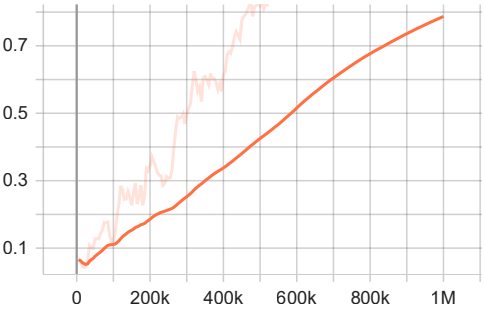
(b) Policy loss



(c) Value loss

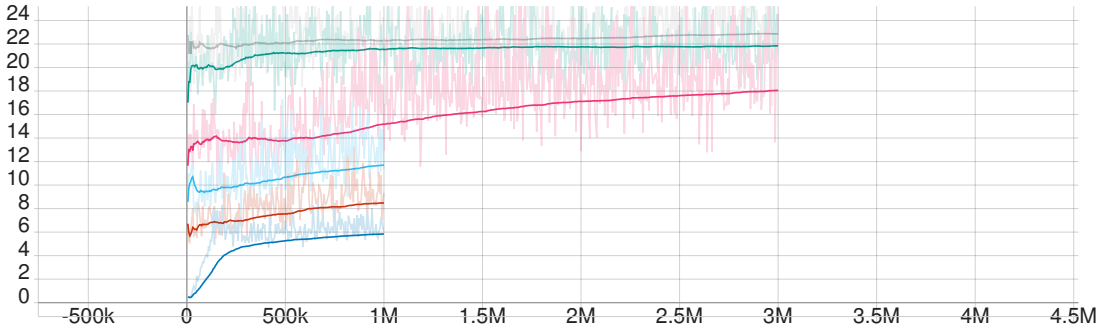


(d) Entropia

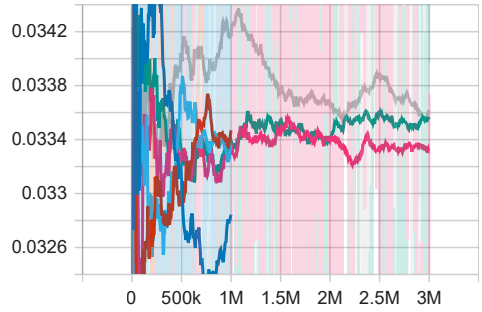


(e) Extrinsic value estimate

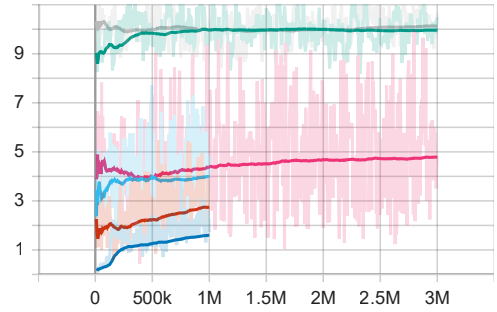
Figura 4.13.: Grafici test 2; numero passi: 1 milione



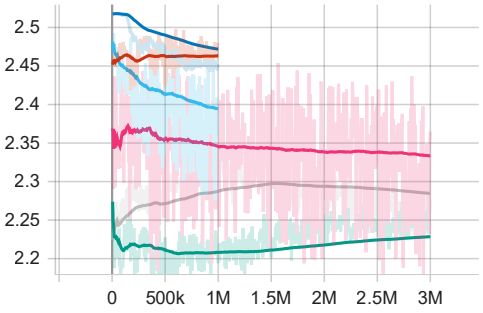
(a) Reward cumulativo



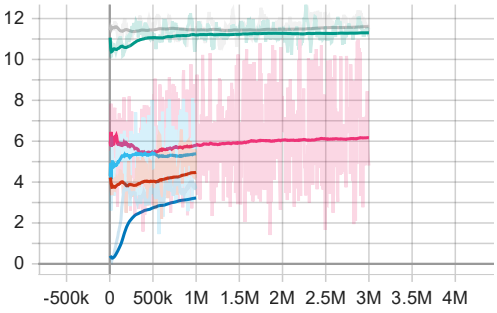
(b) Policy loss



(c) Value loss



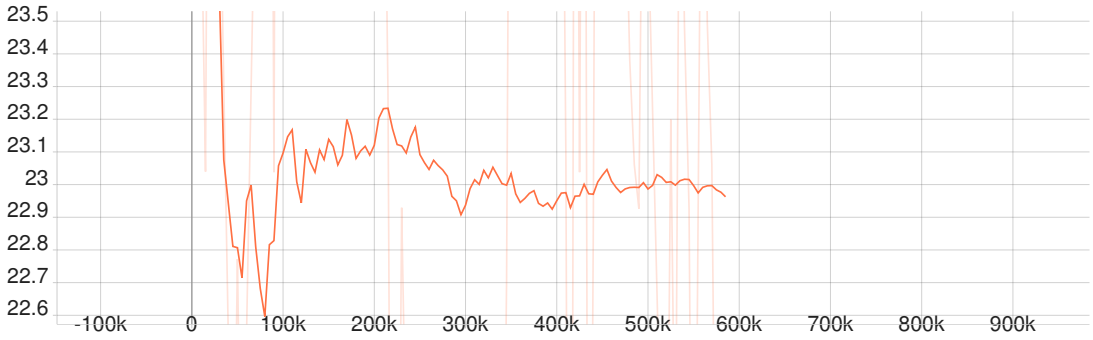
(d) Entropia



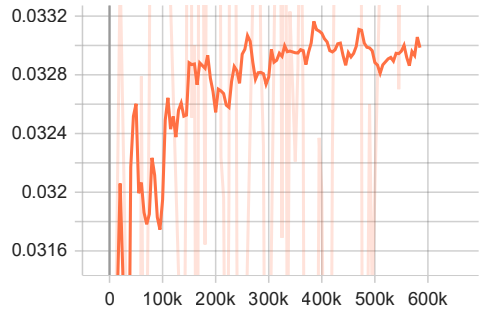
(e) Extrinsic value estimate

- Training 1 (1M passi)
- Training 2 (da rete 1) (1M passi)
- Training 3 (da rete 2) (1M passi)
- Training 4 (da rete 3) (3M passi)
- Training 5 (da rete 4) (3M passi)
- Training 6 (da rete 5) (3M passi)

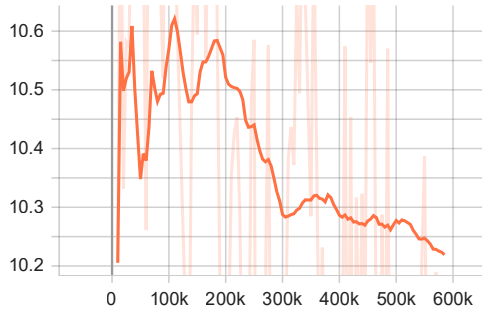
Figura 4.14.: Grafici test 3; gli allenamenti da 2 a 6 sono realizzati a partire dalla rete precedente (non allenati da zero)



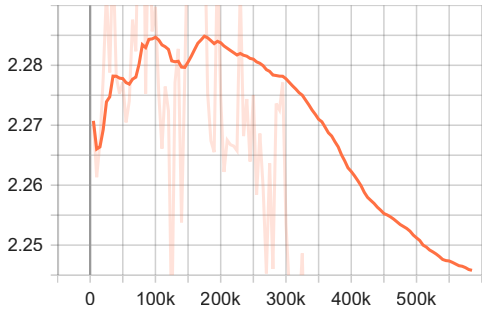
(a) Reward cumulativo



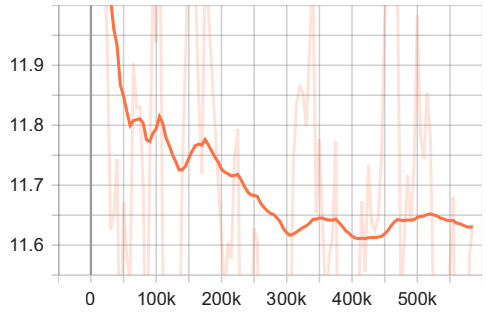
(b) Policy loss



(c) Value loss

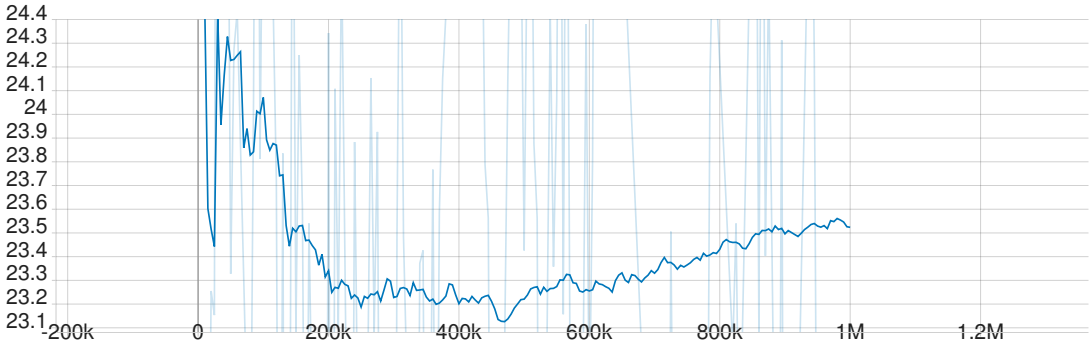


(d) Entropia

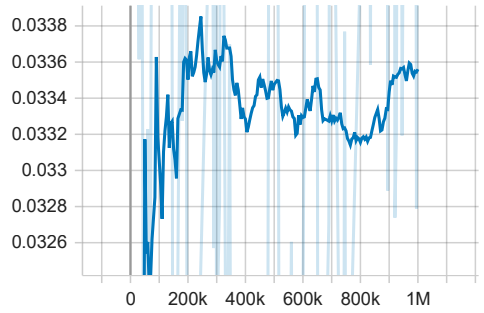


(e) Extrinsic value estimate

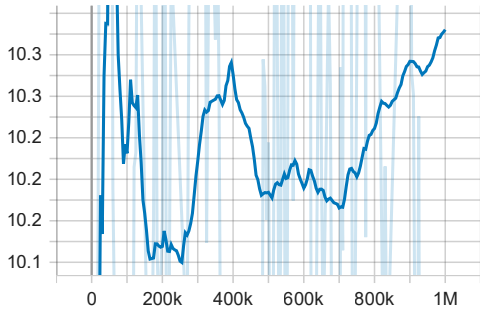
Figura 4.15.: Grafici test 4; numero passi: 585k. Rete trainata a partire dall'ultimo risultato del test 3 Fig.4.14



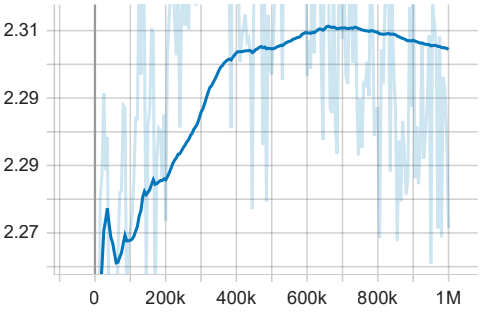
(a) Reward cumulativo



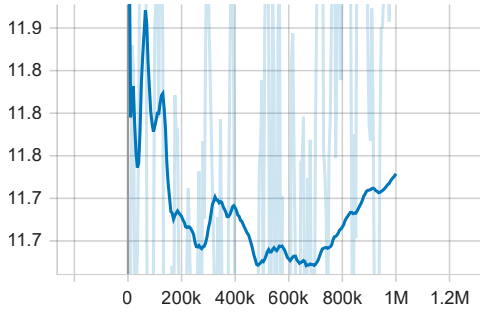
(b) Policy loss



(c) Value loss

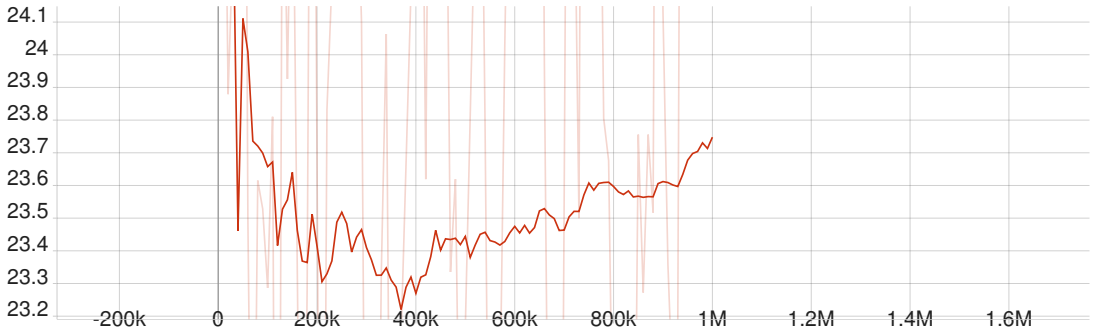


(d) Entropia

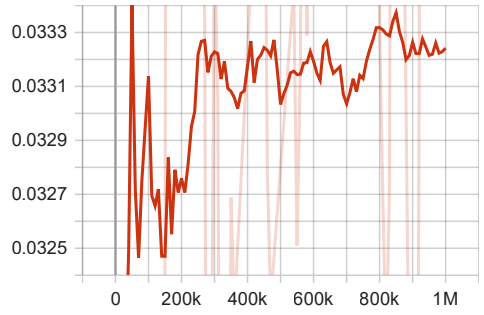


(e) Extrinsic value estimate

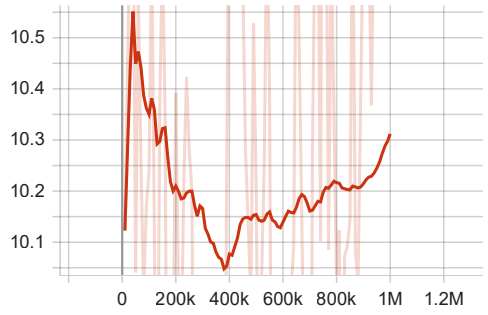
Figura 4.16.: Grafici test 5; numero passi: 1 milione. Rete trainata a partire dall'ultimo risultato del test 3 Fig.4.14



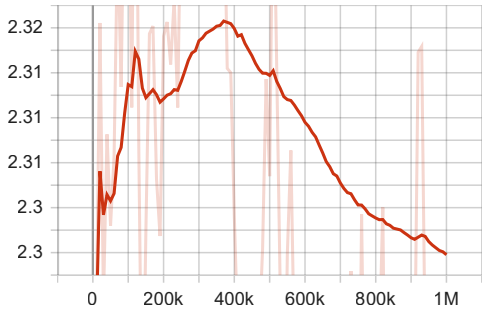
(a) Reward cumulativo



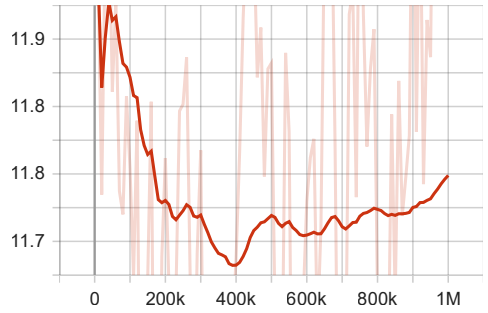
(b) Policy loss



(c) Value loss

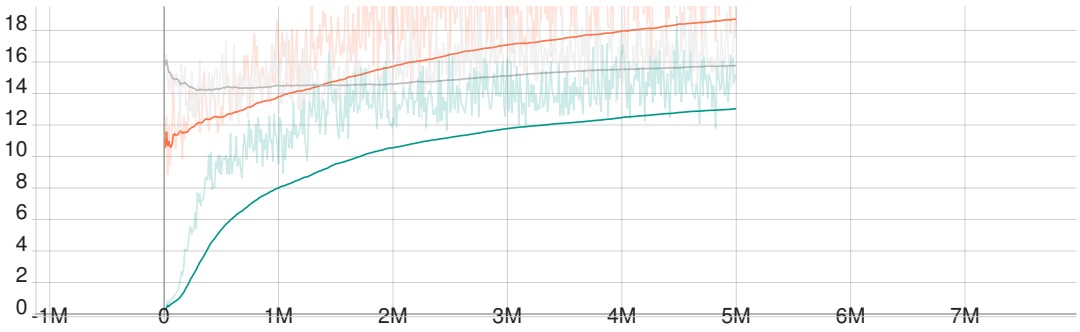


(d) Entropia

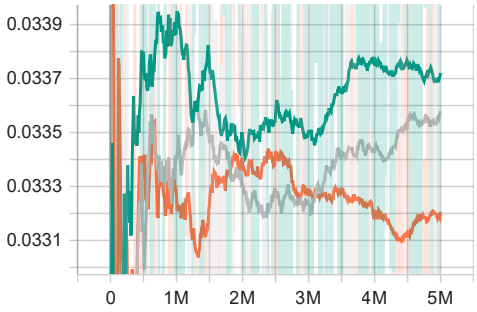


(e) Extrinsic value estimate

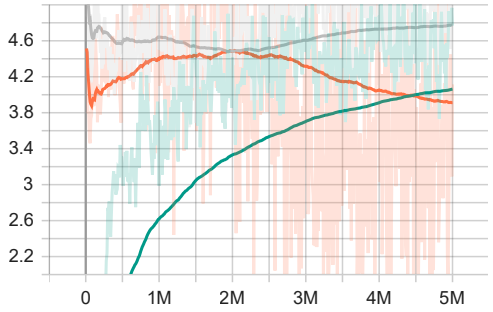
Figura 4.17.: Grafici test 6; numero passi: 1 milione. Rete trainata a partire dal risultato del test 5 Fig.4.16



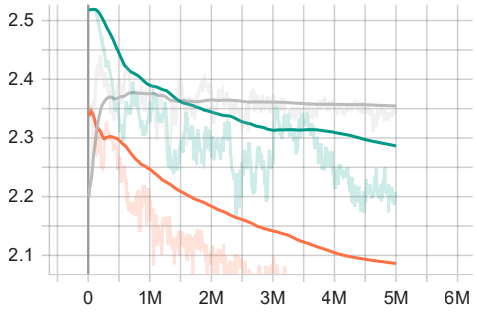
(a) Reward cumulativo



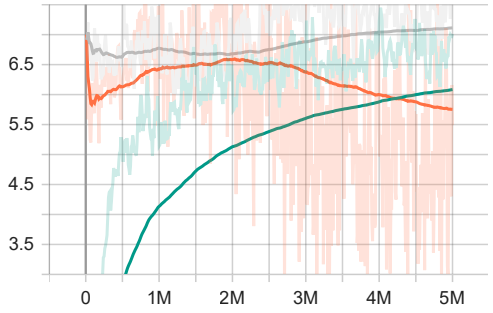
(b) Policy loss



(c) Value loss



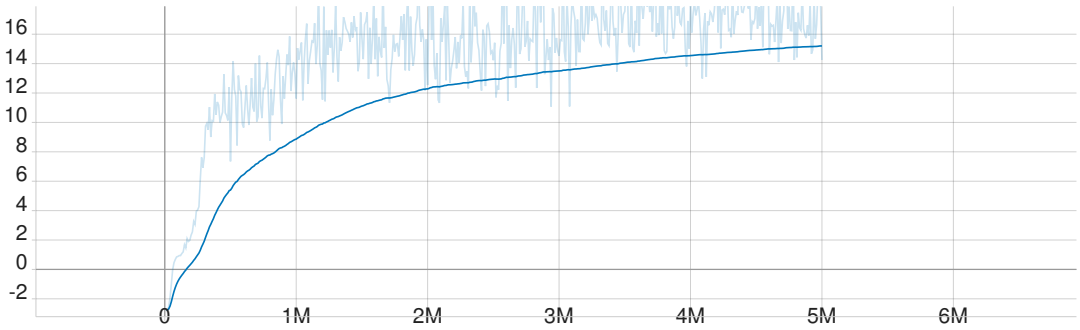
(d) Entropia



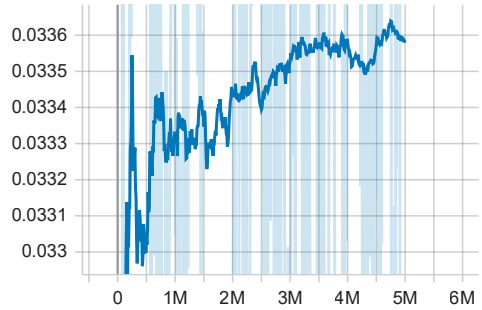
(e) Extrinsic value estimate

Training 1 Training 2 (da rete 1) Training 3 (da rete 2)

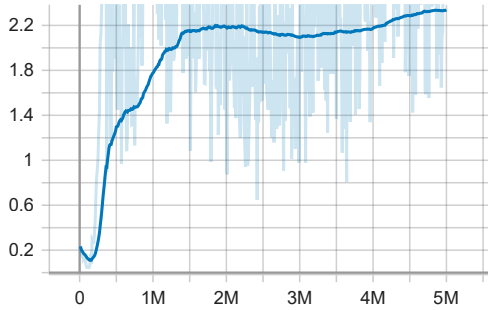
Figura 4.18.: Grafici test 7; numero passi: 5 milioni. Gli allenamenti da 2 e 3 sono realizzati a partire dalla rete precedente (non da zero)



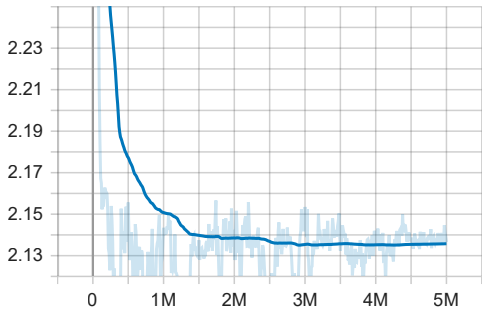
(a) Reward cumulativo



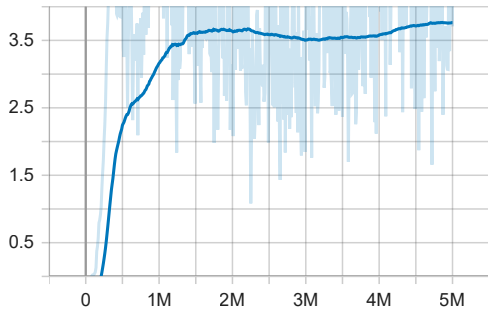
(b) Policy loss



(c) Value loss

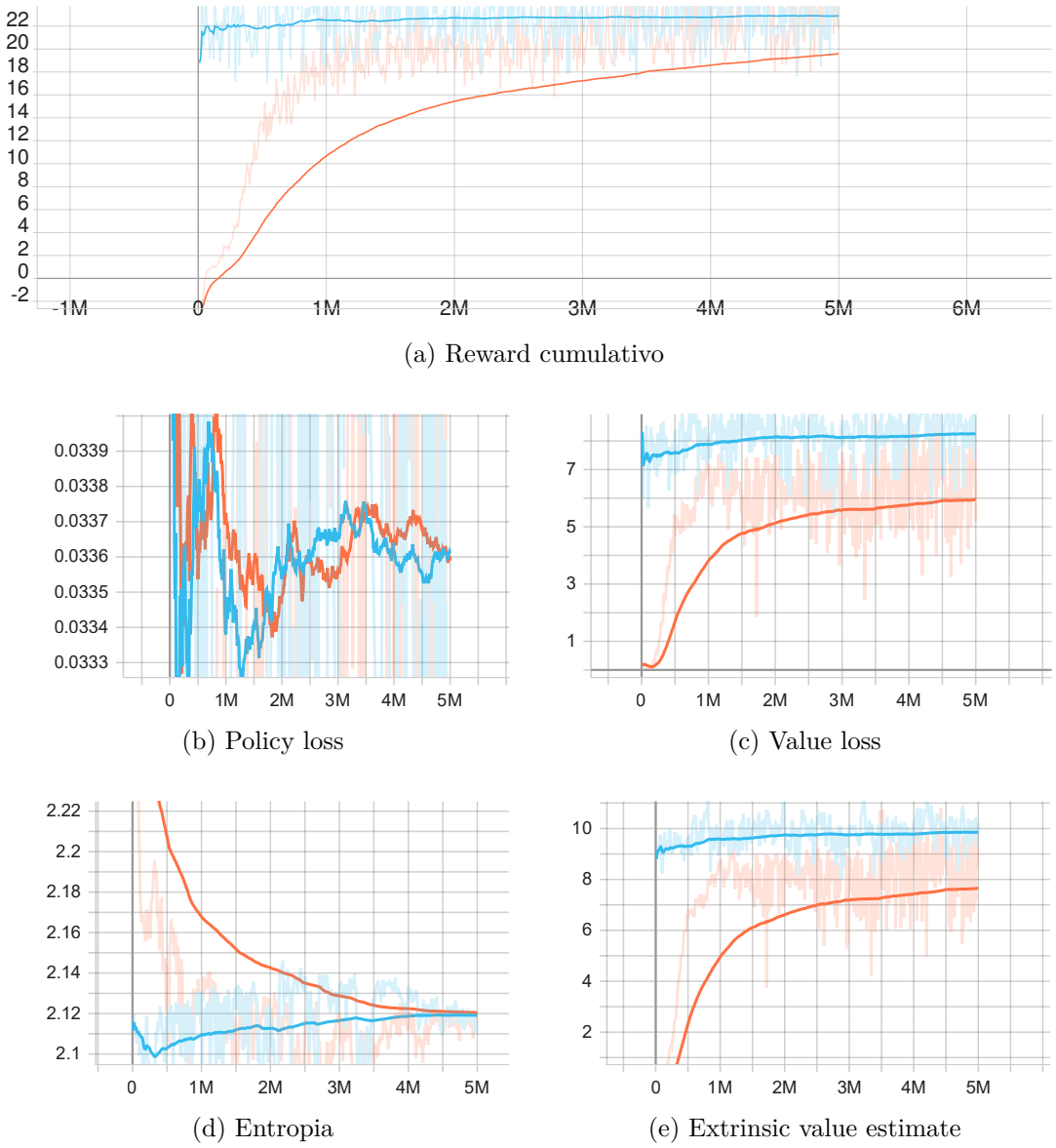


(d) Entropia



(e) Extrinsic value estimate

Figura 4.19.: Grafici test 8; numero passi: 5 milioni



(a) Reward cumulativo

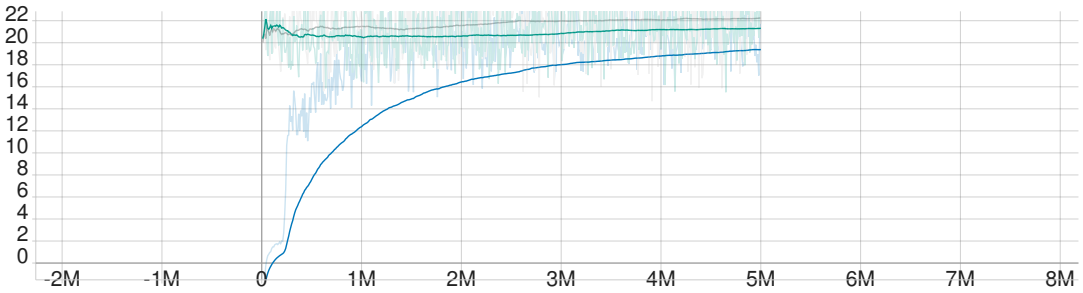
(b) Policy loss

(c) Value loss

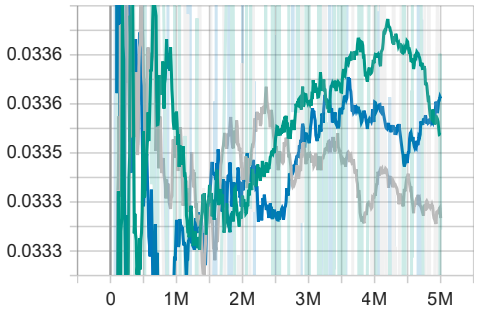
(d) Entropia

(e) Extrinsic value estimate

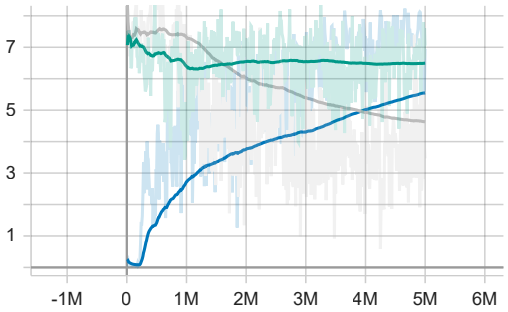
Figura 4.20.: Grafici test 9; numero passi: 5 milioni. Il secondo allenamento è realizzato a partire dalla rete precedente (non da zero)



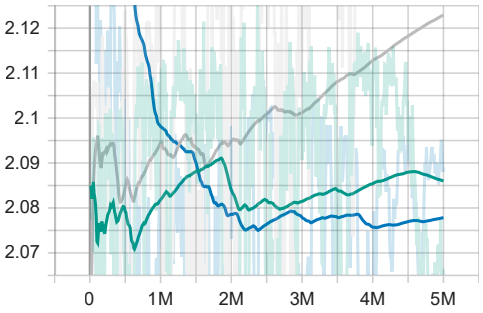
(a) Reward cumulativo



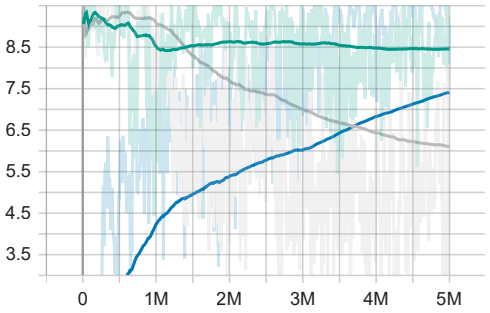
(b) Policy loss



(c) Value loss



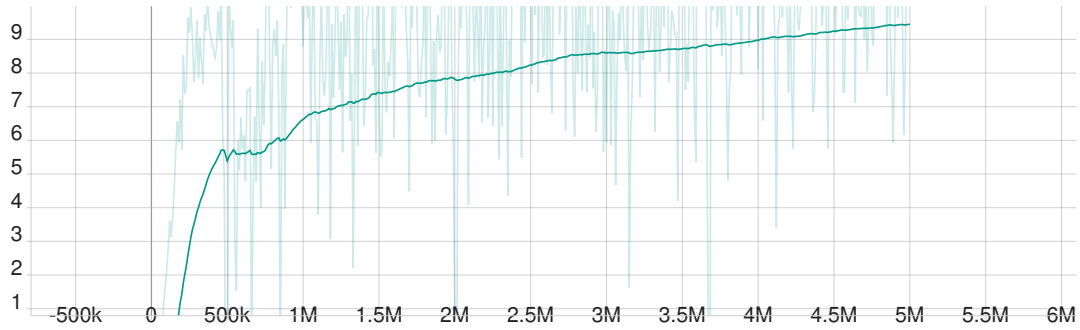
(d) Entropia



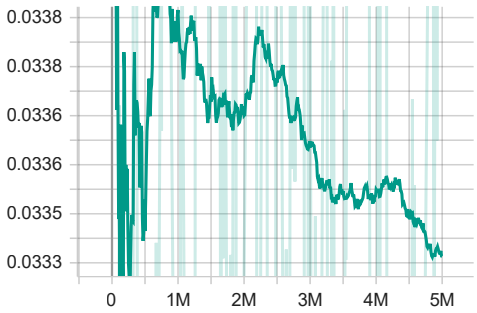
(e) Extrinsic value estimate

Training 1 Training 2 (da rete 1) Training 3 (da rete 2)

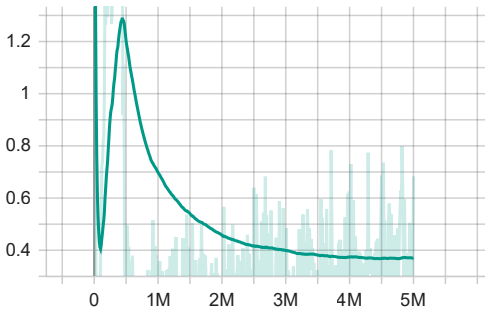
Figura 4.21.: Grafici test 10; numero passi: 5 milioni. Gli allenamenti da 2 e 3 sono realizzati a partire dalla rete precedente (non da zero)



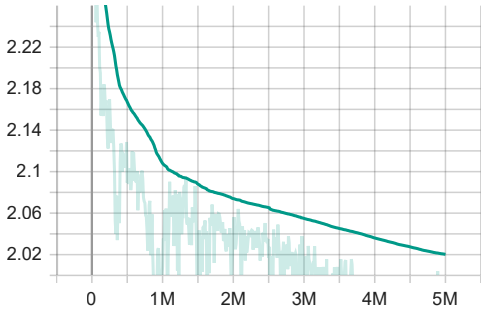
(a) Reward cumulativo



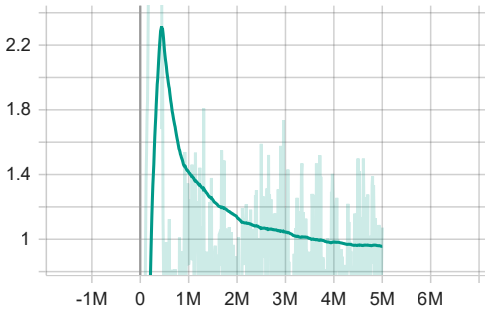
(b) Policy loss



(c) Value loss

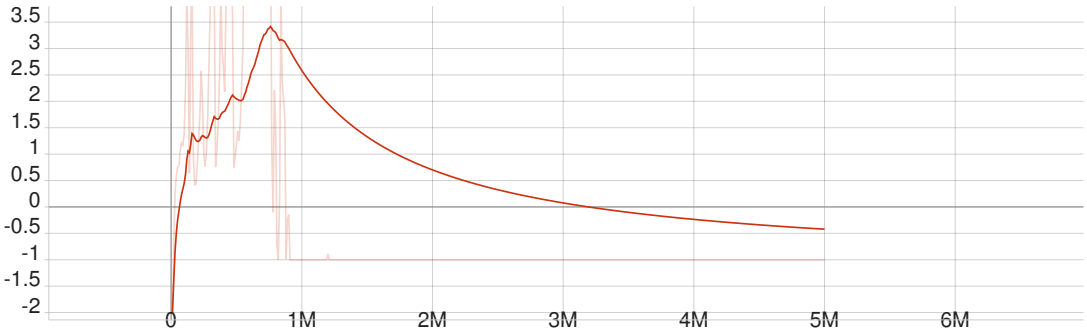


(d) Entropia

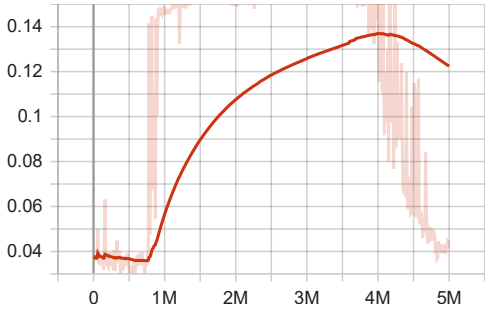


(e) Extrinsic value estimate

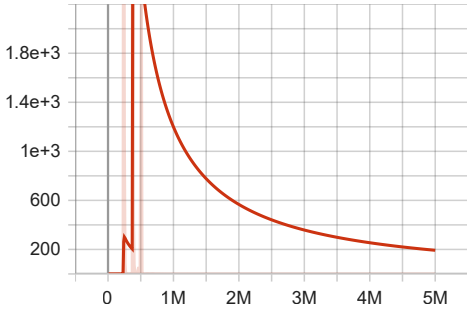
Figura 4.22.: Grafici test 11; numero passi: 5 milioni



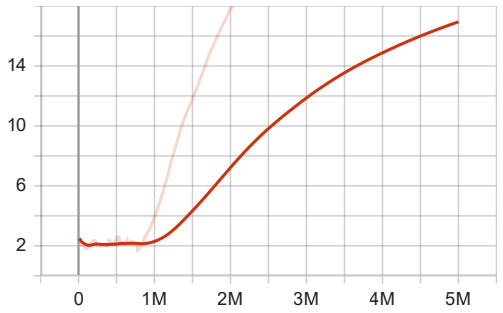
(a) Reward cumulativo



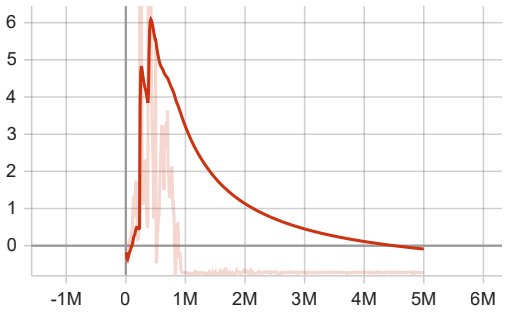
(b) Policy loss



(c) Value loss

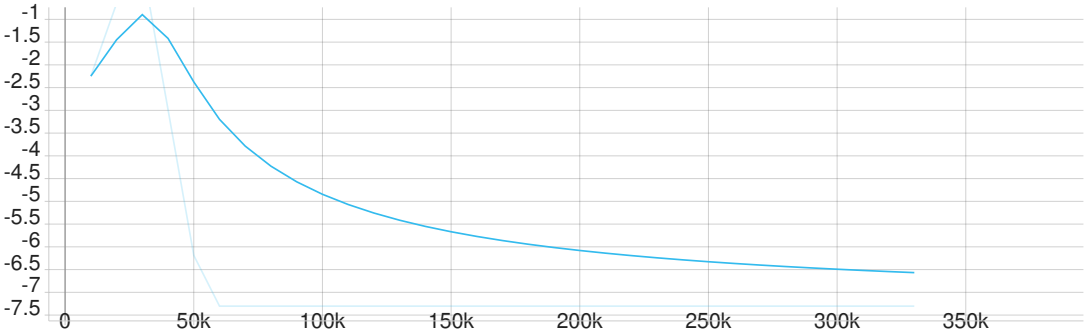


(d) Entropia

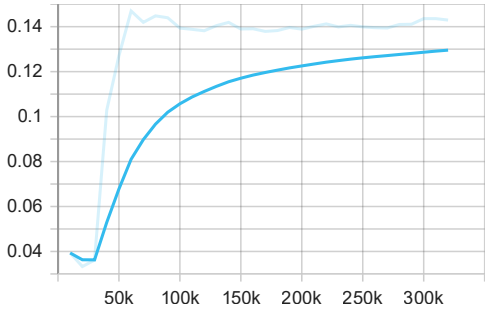


(e) Extrinsic value estimate

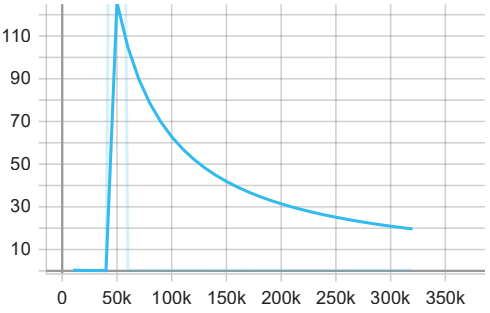
Figura 4.23.: Grafici test 12; numero passi: 5 milioni



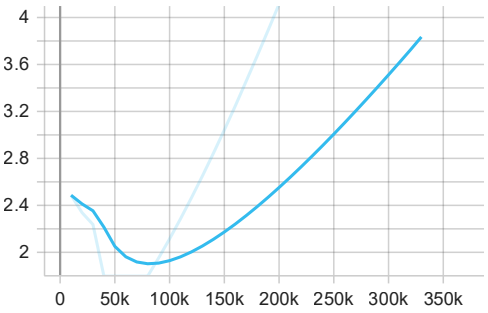
(a) Reward cumulativo



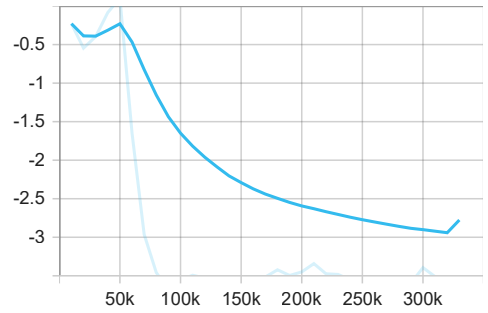
(b) Policy loss



(c) Value loss

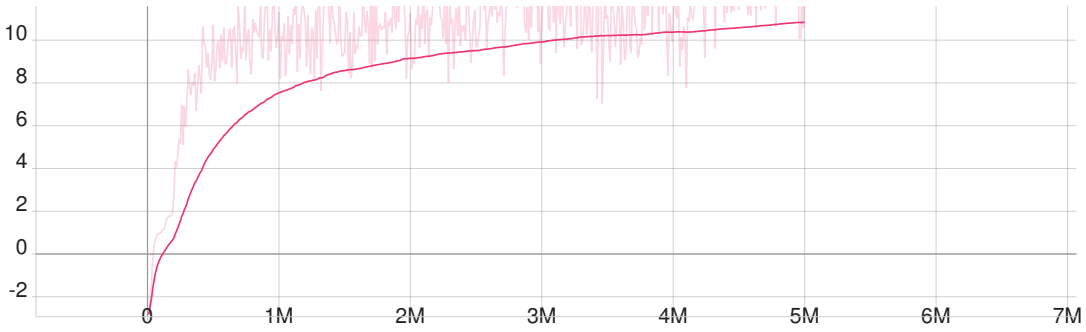


(d) Entropia

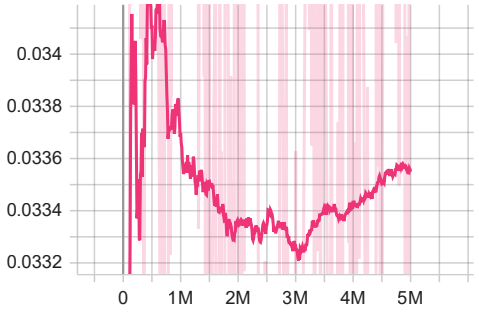


(e) Extrinsic value estimate

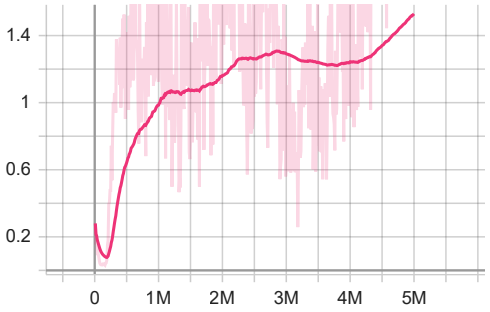
Figura 4.24.: Grafici test 13; numero passi: 330k



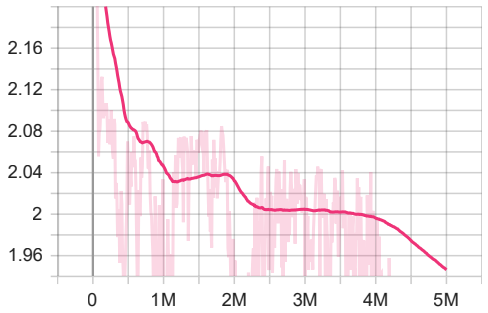
(a) Reward cumulativo



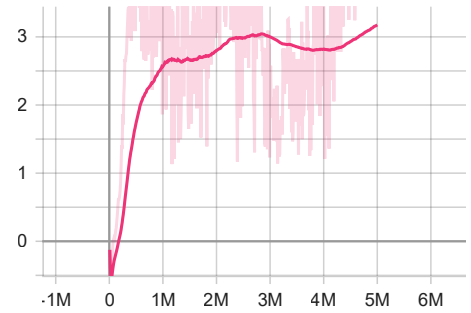
(b) Policy loss



(c) Value loss



(d) Entropia



(e) Extrinsic value estimate

Figura 4.25.: Grafici test 14; numero passi: 5 milioni

4.3. Confronto risultati

Vengono di seguito discussi i risultati ottenuti e mostrati in 4.2, evidenziando quali siano i migliori e le motivazioni del loro successo.

4.3.1. Super Sparty Bros.

Come evidenziato nella sezione precedente, il risultato migliore è stato quello ottenuto nel test 11 Fig.4.11. Nessuno degli altri test ha delle metriche alla pari.

Viene riportato, per ogni esperimento, il comportamento dell'agente in inferenza ed una valutazione degli iperparametri utilizzati (Tab.4.1, Tab.4.2):

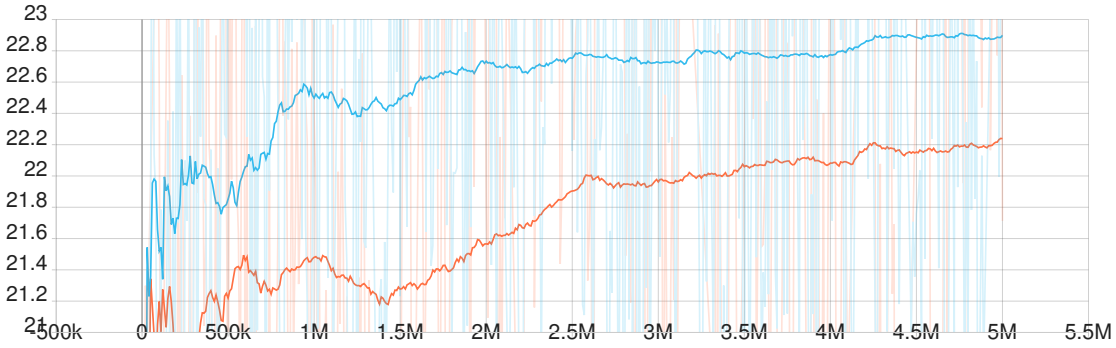
- Test 1 Fig.4.1: non funzionante; interazione troppo bassa per comprendere la scena. Batch size e buffer size troppo bassi (non permettono di raggiungere i reward), beta troppo alto (l'agente esplora lo spazio troppo superficialmente), epsilon troppo basso (rallenta il training), numero layer troppo basso (non permette un apprendimento profondo da parte dell'agente)
- Test 2 Fig.4.2: non riesce a salire sulla prima piattaforma. Beta troppo alto, epsilon troppo basso, numero layer troppo basso (stessi problemi ai parametri del test 1)
- Test 3 Fig.4.3: non funzionante. Batch size e buffer size troppo bassi, beta troppo alto (rende le azioni troppo casuali)
- Test 4 Fig.4.4: agente prende le prime monete poi si muove sul posto. Batch size e buffer size troppo bassi (non permettono di raggiungere tutti i reward), beta troppo basso (esplorazione troppo lenta)
- Test 5 Fig.4.5, test 6 Fig.4.6, test 7 Fig.4.7,test 8 Fig.4.8: agente prende le prime monete poi si scontra volontariamente con nemico; possibilità di rete caduta in minimo locale. Per una limitazione di ML-Agents non è stato possibile cambiare il learning rate da uno scheduling lineare, perciò si è reso necessario apportare delle modifiche al posizionamento dei reward
- Test 9 Fig.4.9: agente salta sul posto senza muoversi. Batch size e buffer size troppo alti (passa troppo tempo tra un aggiornamento del modello ed il successivo), learning rate troppo alto (rende il training instabile), epsilon troppo alto (induce cambiamenti di policy troppo drastici, portando ad un training instabile), numero di epoche e numero di layer troppo alto (rallentano il training)
- Test 10 Fig.4.10: non funzionante. Batch size e buffer size troppo alti, learning rate troppo alto (anche se più basso del test 9), epsilon troppo alto, numero di epoche e numero di layer troppo alto (come test 9)
- Test 11 Fig.4.11: rete funzionante. Parametri corretti

Il gioco presenta anche un secondo livello, su cui sono stati effettuati dei test (sia utilizzando una rete pre-trainata sul primo livello sia partendo da zero); purtroppo, data la differente struttura che questo livello presenta rispetto al primo, in particolare per quanto riguarda il posizionamento degli oggetti (monete, nemici, piattaforme ecc.) ed i conseguenti reward associati, i risultati ottenuti non sono stati soddisfacenti.

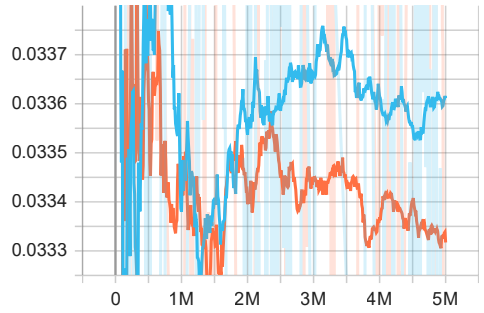
4.3.2. Red Runner

In Fig.4.26 è riportato un confronto diretto tra l'allenamento delle due migliori reti (9 e 10); come si può notare, i reward cumulativi sono alti e molto simili (22.9 per test 9, 22.24 per test 10), favorendo leggermente la rete del test 9; anche la policy loss è estremamente simile, attestandosi su 0.03361 per il test 9 e 0.03332 per il

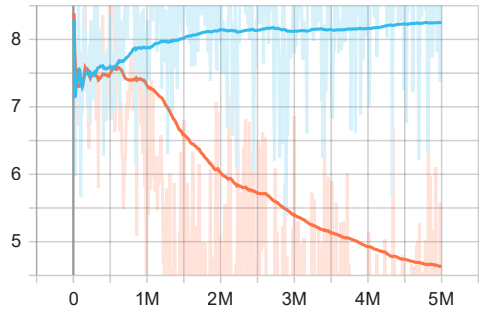
test 10, entrambi valori piuttosto bassi e nella media dei test effettuati. La value loss è più o meno stabile ma abbastanza elevata (valore finale 8.25) per il test 9, mentre decresce ed è circa la metà (valore finale 4.632) nel test 10, favorendo, in tal modo, quest'ultimo. L'entropia finale per il test 9 è di 2.119, con una tendenza al ribasso, mentre è di 2.123 per il test 10, con una leggera tendenza al rialzo (risultano comunque essere entrambi sotto la media dei valori rilevati negli esperimenti); migliore in questo caso, quindi, è il test 9. Infine, l'extrinsic value estimate finale risulta essere di 9.854 per il test 9, in leggera crescenza, e di 6.102 per il test 10, decrescente. In questo caso viene quindi favorito il test 9, che risulta avere un margine maggiore di apprendimento rispetto al test 10.



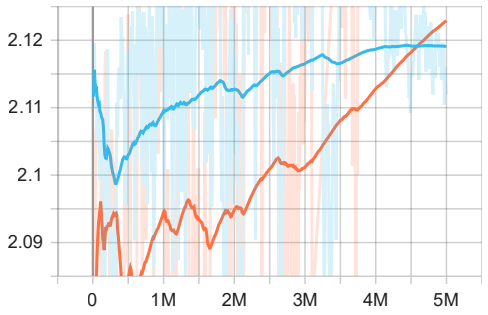
(a) Reward cumulativo



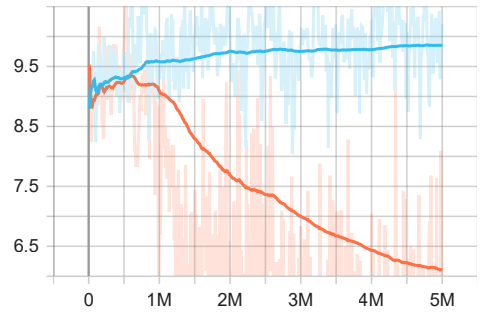
(b) Policy loss



(c) Value loss



(d) Entropia



(e) Extrinsic value estimate

Test 9 (training 2) Test 10 (training 3)

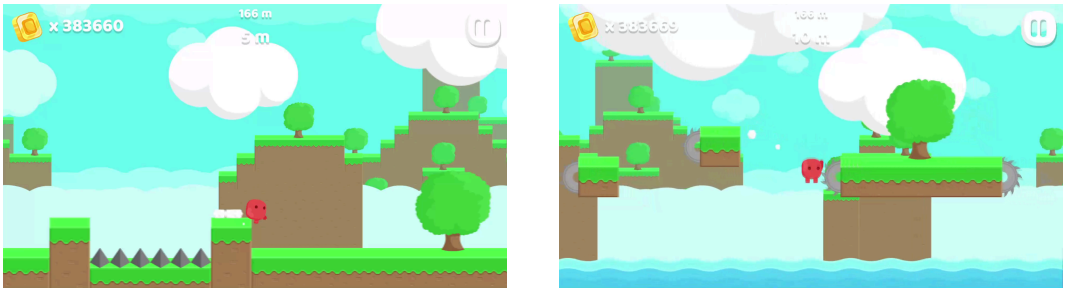
Figura 4.26.: Confronto test 9 (training 2) e test 10 (training 3); numero passi: 5 milioni

Come si può dedurre dal confronto appena effettuato, entrambe le reti presentano dei punti di forza e di debolezza nel loro andamento rispetto l'un l'altra. Sono comunque molto simili ed anche il loro comportamento in inferenza risulta essere pressoché il medesimo, riuscendo a superare svariati ostacoli, alcuni anche in maniera differente.

Viene riportata una dimostrazione video[2] che documenta il comportamento della rete generata dal test 10. Come si può notare dal video l'agente riesce a superare facilmente diversi ostacoli Fig.4.27a, mentre fa più fatica in alcuni specifici scenari

[2] `https://www.youtube.com/watch?v=2e8R_so0Cc4`

Fig.4.27b. È plausibile che con un training estensivo si possano migliorare ancora le performance, già più che buone, data la randomicità dell'ambiente.



(a) Superamento di un ostacolo (b) Non superamento di un ostacolo

Figura 4.27.: Esempi di interazione con ostacoli in Red Runner

Viene riportato, per ogni esperimento, il comportamento dell'agente in inferenza ed una valutazione degli iperparametri utilizzati (Tab.4.3, Tab.4.4, Tab.4.5):

- Test 1 Fig.4.12: agente salta troppo spesso e quindi non prende velocità. Beta troppo basso (lo spazio degli stati non viene esplorato al meglio), epsilon troppo basso (rallenta il training), numero di layer troppo basso (il problema è complesso, serve una rete con più hidden layer)
- Test 2 Fig.4.13: agente salta e si muove sul posto, senza prendere nessun checkpoint. Learning rate troppo basso (non apprende correttamente), beta troppo basso, epsilon troppo basso, numero di layer troppo basso (stesse motivazioni test 1)
- Test 3 Fig.4.14: agente salta subito nell'acqua. Beta troppo basso, epsilon troppo basso, numero di layer troppo basso (stesse motivazioni test 1)
- Test 4 Fig.4.15: agente salta subito nell'acqua. Beta troppo basso, epsilon troppo basso, numero di layer troppo basso (stesse motivazioni test 1)
- Test 5 Fig.4.16: agente salta subito nell'acqua. Epsilon troppo basso, numero di layer troppo basso (stesse motivazioni test 1)
- Test 6 Fig.4.17: agente salta subito nell'acqua. Numero di layer troppo basso
- Test 7 Fig.4.18: movimento discreto ma l'agente salta troppo spesso, perdendo velocità; dopo questo test è stata introdotta una penalità per il salto.
- Test 8 Fig.4.19: buon andamento ma a volte sbaglia all'inizio; a causa delle limitazioni di ML-Agents non è stato possibile modificare lo scheduling del learning rate da lineare
- Test 9 Fig.4.20: ottimi risultati. Utilizzando 4 layer e 256 hidden units sono bastate 3 epoche per avere ottime performance
- Test 10 Fig.4.21: ottimi risultati. Rispetto al test 9 sono stati aumentati i layer a 5, il risultato è ancora migliore (seppur simile) del precedente

- Test 11 Fig.4.22: agente riluttante ad oltrepassare gli ostacoli, saltando sul posto quando ne incontra uno che non pensa di poter oltrepassare. Passare il numero di epoche da 3 a 5 ha solo peggiorato le performance della rete, rendendola più instabile
- Test 12 Fig.4.23: in questo e nei successivi due test si è provato ad impostare *Stacked Vectors* e *Stacked Raycasts* a 4, con risultati non soddisfacenti. In particolare, in questo test l'agente cade in un minimo locale, cercando di concludere il prima possibile l'episodio perdendo subito. Il learning rate è risultato essere troppo alto (rendendo il training instabile), numero di layer troppo alto
- Test 13 Fig.4.24: stesso comportamento dell'agente del test precedente. Numero di layer troppo alto (rallentamento eccessivo del training)
- Test 14 Fig.4.25: l'agente esita ad oltrepassare gli ostacoli fino alla fine dell'episodio. Numero di layer troppo alto

Sono state anche effettuate delle prove in Red Runner utilizzando la rete migliore allenata in Super Sparty Bros.. Come era plausibile aspettarsi, le performance della rete sono state deludenti, in quanto i due ambienti sono estremamente dissimili, l'obiettivo è diverso ed anche gli ostacoli sono differenti. Inoltre Red Runner necessità di una capacità di generalizzazione, data dall'ambiente proceduralmente generato, che in Super Sparty Bros. non è richiesta.

Capitolo 5.

Conclusione e sviluppi futuri

5.1. Riflessioni

In questo lavoro di tesi è stato dimostrato che, con semplici modifiche, esiste la concreta possibilità di adattare al Deep Reinforcement Learning, con il fine di allenare agenti intelligenti, i videogiochi platform 2D già sviluppati. In particolare, è stato mostrato come sia nel caso di Super Sparty Bros. (platform con struttura a livelli) sia nel caso di Red Runner (platform endless runner) sia possibile allenare delle reti efficaci con risultati soddisfacenti. È stato anche provato come i due videogiochi analizzati producano reti incompatibili l'un l'altro, data la diversità di ambiente, di obiettivo e la necessità di generalizzazione maggiore nel caso di Red Runner, a causa della randomicità dell'ambiente. Inoltre, è risultata evidente la differenza nei tempi di apprendimento delle reti sulla base del videogioco selezionato: mentre in Super Sparty Bros. la rete riesce dopo relativamente poco tempo a raggiungere l'obiettivo, in Red Runner, dove l'ambiente è generato proceduralmente, la rete necessita di più tempo per apprendere e generalizzare al meglio.

5.2. Sviluppi futuri

La maggiore sfida presente in questo ambiente è sicuramente la creazione di una rete capace di generalizzare il più possibile; alcune idee per migliorare l'apprendimento sotto questo punto di vista potrebbero essere:

- Sostituire le osservazioni raycast con delle osservazioni visuali, che andrebbero a migliorare la visione globale che l'agente ha dell'enviornment; l'utilizzo di osservazioni visuali comporterebbe sicuramente, però, un aumento del costo computazionale e del tempo di allenamento
- Solamente per Red Runner, si potrebbero inserire le informazioni di posizioni dei checkpoint come osservazioni vettoriali anziché raycast, in modo che i raggi non vengano "bloccati" dai checkpoint, permettendo all'agente di avere una maggiore visibilità dell'ambiente, individuando al meglio gli ostacoli che sono davanti a lui

Infine, per quanto concerne la velocizzazione dell'allenamento, una possibilità è quella di ristrutturare il codice in modo da renderlo idoneo al training multi-agente.

Appendice A.

Codice creato e modificato in Super Sparty Bros.

A.1. Classe CharacterAgent

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;

public class CharacterAgent : Agent
{
    private CharacterController2D character;
    private Victory rose;
    private Vector3 localPos;
    private Vector3 rosePos;

    private int jumpNumber = 0;
    private int stepCount = 0;

    private void Awake()
    {
        character = GetComponent<CharacterController2D>();
        rose =
        ↪  GameObject.FindWithTag("Rose").GetComponent<Victory>();
        rosePos = rose.transform.localPosition;
        Academy.Instance.AutomaticSteppingEnabled = false;
    }
```

```
private void Start()
{
    CharacterController2D.getInstance().OnDied +=
    ↪ Character_OnDied;
}

private void FixedUpdate()
{
    stepCount++;
    if (stepCount >= MaxStep)
    {
        StartCoroutine(character.KillPlayer());
        EndEpisode();
    }
}

void Update()
{
    Academy.Instance.EnvironmentStep();
}

private void Character_OnDied(object sender, System.EventArgs e)
{
    Debug.Log("END EPISODE");
    EndEpisode();
}

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(character.transform.localPosition);
    if (character._isGrounded)
    {
        jumpNumber = 2;
    } else if (character._canDouvbleJump)
    {
        jumpNumber = 1;
    }
    else
    {
        jumpNumber = 0;
    }
    sensor.AddObservation(jumpNumber);
```

```
        sensor.AddObservation(rosePos);

    }
    public override void OnActionReceived(ActionBuffers
    ↪  actionBuffers)
    {
        var movement = actionBuffers.DiscreteActions[0];
        var jump = actionBuffers.DiscreteActions[1];

        switch (movement)
        {
            case 0:
                character.direction = 0; //character fermo
                break;
            case 1:
                character.direction = 1; //movimento a destra
                break;
            case 2:
                character.direction = -1; //movimento a sinistra
                break;
        }

        switch (jump)
        {
            case 0:
                character.jumping = 0; //non sta saltando
                break;
            case 1:
                character.jumping = 1; //jump button down
                break;
            case 2:
                character.jumping = 2; //jump button up
                break;
        }
        AddReward(-1f / MaxStep);
    }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var discreteActionsOut = actionsOut.DiscreteActions;
        if (Input.GetAxisRaw("Horizontal") == 1f)
            discreteActionsOut[0] = 1;
```

```
        if (Input.GetAxisRaw("Horizontal") == -1f)
            discreteActionsOut[0] = 2;
        if (Input.GetButtonDown("Jump"))
            discreteActionsOut[1] = 1;
        if (Input.GetButtonUp("Jump"))
            discreteActionsOut[1] = 2;
    }

    private void OnCollisionEnter2D(Collision2D c)
    {
        if (c.gameObject.CompareTag("DeathZone"))
        {
            AddReward(-1f);
            EndEpisode();
        }
        if (c.gameObject.CompareTag("Teleport"))
            AddReward(4f);
    }

    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("coin"))
            AddReward(.1f);
        if (collision.gameObject.CompareTag("Rose"))
        {
            AddReward(5f);
            Debug.Log("END EPISODE WIN");
            EndEpisode();
        }
        if (collision.gameObject.CompareTag("Enemy"))
        {
            AddReward(-1f);
            EndEpisode();
        }
    }
}
```

A.2. Classe CharacterController2D

Modificato metodo `Update()` per adattare movimento e salto al DRL; fixato controllo collisione con terreno; rimosso passaggio dal basso attraverso piattaforma.

```
public void Update()
{
        if (!playerCanMove || (Time.timeScale == 0f))
                return;

        _vx = direction;

        if (_vx != 0)
        {
                _isRunning = true;
        } else {
                _isRunning = false;
        }

        _animator.SetBool("Running", _isRunning);

        _vy = _rigidbody.velocity.y;

    //fixato groundCheck
        _isGrounded = Physics2D.Linecast(new
        ↪  Vector2(groundCheck.position.x, groundCheck.position.y +
        ↪  0.1f), groundCheck.position, whatIsGround);

        if(_isGrounded)
        {
                _canDouvbleJump = true;
        }

        _animator.SetBool("Grounded", _isGrounded);

        if(_isGrounded && jumping == 1) // If grounded AND jump
        ↪  button pressed, then allow the player to jump
        {
                DoJump();
        } else if (_canDouvbleJump && jumping == 1)
        {
                DoJump();
                _canDouvbleJump = false;
        }

        if(jumping == 2 && _vy>0f)
```

```
        {
                _vy = 0f;
        }

        _rigidbody.velocity = new Vector2(_vx * moveSpeed, _vy);

    //rimosso passaggio attraverso piattaforma
        //Physics2D.IgnoreLayerCollision(_playerLayer,
        ↪  _platformLayer, (_vy > 0.0f));
}
```

Appendice B.

Codice creato e modificato in Red Runner

B.1. Classe CheckpointSingle

```
using System;
using System.Collections;
using System.Collections.Generic;
using RedRunner.Characters;
using UnityEngine;

public class CheckpointSingle : MonoBehaviour
{
    private TrackCheckpoints trackCheckpoints;
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.TryGetComponent<RedCharacter>(out RedCharacter
        ↪  redCharacter))
        {
            trackCheckpoints.PlayerThroughCheckpoint(this);
            redCharacter.setTrackCheckpointsRed(trackCheckpoints);
        }
    }

    public void SetTrackCheckpoint(TrackCheckpoints
    ↪  trackCheckpoints)
    {
        this.trackCheckpoints = trackCheckpoints;
    }

    public string getBlockName()
    {
        return transform.parent.parent.name;
    }
```

```
    public TrackCheckpoints getBlockTrackCheckpoints()
    {
        return
        ↪  transform.parent.parent.GetComponent<TrackCheckpoints>();
    }
}
```

B.2. Classe TrackCheckpoints

```
using System;
using System.Collections;
using System.Collections.Generic;
using RedRunner.Characters;
using UnityEngine;
using UnityEngine.Events;

public class TrackCheckpoints : MonoBehaviour
{
    public event EventHandler OnPlayerCorrectCheckpoint;
    public event EventHandler OnPlayerWrongCheckpoint;

    private List<CheckpointSingle> checkpointSingleList;
    private int nextCheckpointSingleIndex;

    private String checkpointCorrect;
    private void Awake()
    {
        Transform checkpointTransform =
        ↪  transform.Find("Checkpoints");

        checkpointSingleList = new List<CheckpointSingle>();
        foreach (Transform checkpointSingleTransform in
        ↪  checkpointTransform)
        {
            CheckpointSingle checkpointSingle =
            ↪  checkpointSingleTransform.GetComponent
            <CheckpointSingle>();
            checkpointSingle.SetTrackCheckpoint(this);

            checkpointSingleList.Add(checkpointSingle);
        }
```

```
        nextCheckpointSingleIndex = 0;

        checkpointCorrect = "not set";
    }

    public void PlayerThroughCheckpoint(CheckpointSingle
    ↪ checkpointSingle)
    {
        if (checkpointSingleList.IndexOf(checkpointSingle) ==
        ↪ nextCheckpointSingleIndex)
        {
            checkpointCorrect = "correct";
            OnPlayerCorrectCheckpoint?.Invoke(this,
            ↪ EventArgs.Empty);
            Debug.Log("correct");
            nextCheckpointSingleIndex++;
        }
        else
        {
            checkpointCorrect = "wrong";
            OnPlayerWrongCheckpoint?.Invoke(this, EventArgs.Empty);
            Debug.Log("wrong");
        }
    }

    public int getCheckpointsNumber()
    {
        return checkpointSingleList.Count;
    }

    public void setCheckpointState(String checkpointCorrect)
    {
        this.checkpointCorrect = checkpointCorrect;
    }

    public String getCheckpointState()
    {
        return checkpointCorrect;
    }
}
```

B.3. Classe RedAgent

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization.Json;
using RedRunner;
using RedRunner.Characters;
using RedRunner.TerrainGeneration;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
using UnityStandardAssets.CrossPlatformInput;

public class RedAgent : Agent
{
    private RedCharacter redrunner;
    private TrackCheckpoints trackCheckpoints;
    private int currentBlockCheckpointsNumber;
    private TrackCheckpoints currentTrackCheckpoints;
    private bool firstCheckpointPassed;
    private bool agentDead;
    private int stepsSinceLastCheckpoint;
    Rigidbody2D redrunnerRigidbody;

    [SerializeField] private int maxEnvironmentStep;

    public override void OnEpisodeBegin()
    {
        currentBlockCheckpointsNumber = 0;
        firstCheckpointPassed = false;
        StartCoroutine(FirstTrackCheckpoint());
        agentDead = false;
        stepsSinceLastCheckpoint = 0;
        redrunnerRigidbody.constraints =
        ↪  RigidbodyConstraints2D.FreezeAll;
    }

    public void setAgentDead(bool agentDead)
    {
```

```
        this.agentDead = agentDead;
    }

    IEnumerator FirstTrackCheckpoint()
    {
        yield return new WaitForSeconds (1f * Time.timeScale);
        trackCheckpoints =
        ↪  TerrainGenerator.Singleton.GetCharacterBlock().
        GetComponent<TrackCheckpoints>();
        currentTrackCheckpoints = trackCheckpoints;
        Subscribe(trackCheckpoints);
        redrunnerRigidbody.constraints &=
        ↪  ~RigidbodyConstraints2D.FreezePosition;
    }

    private void Subscribe(TrackCheckpoints tc)
    {
        Debug.Log("SUBSCRIBED TO " + tc);
        tc.OnPlayerCorrectCheckpoint +=
        ↪  TrackCheckpoints_OnPlayerCorrectCheckpoint;
        tc.OnPlayerWrongCheckpoint +=
        ↪  TrackCheckpoints_OnPlayerWrongCheckpoint;
    }

    private void TrackCheckpoints_OnPlayerCorrectCheckpoint(object
    ↪  sender, System.EventArgs e)
    {
        stepsSinceLastCheckpoint = 0;
        firstCheckpointPassed = true;
        Debug.Log("reward added");
        AddReward(1f);
        currentBlockCheckpointsNumber++;
        if (currentBlockCheckpointsNumber ==
        ↪  currentTrackCheckpoints.getCheckpointsNumber())
            currentBlockCheckpointsNumber = 0;
    }

    private void TrackCheckpoints_OnPlayerWrongCheckpoint(object
    ↪  sender, System.EventArgs e)
    {
        stepsSinceLastCheckpoint = 0;
        Debug.Log("penalty added");
```

```
        AddReward(-1f);
        Unsubscribe(trackCheckpoints);
        redrunner.Die();
        EndEpisode();
    }

    private void Awake()
    {
        redrunner = GetComponent<RedCharacter>();
        Academy.Instance.AutomaticSteppingEnabled = false;
        redrunnerRigidbody = GetComponent<Rigidbody2D>();
    }

    void Update()
    {
        Academy.Instance.EnvironmentStep();

        stepsSinceLastCheckpoint++;
        if (stepsSinceLastCheckpoint >= maxEnvironmentStep)
        {
            redrunner.Die(false);
            EndEpisode();
        }
    }

    private void FixedUpdate()
    {
        if (firstCheckpointPassed){
            trackCheckpoints = redrunner.getTrackCheckpointsRed();
            if (trackCheckpoints != currentTrackCheckpoints)
            {
                currentTrackCheckpoints = trackCheckpoints;
                Subscribe(trackCheckpoints);
                Debug.Log("rew fixedupdate");
                Debug.Log("reward added");
                AddReward(1f);
            }
        }

        if (agentDead)
        {
            AddReward(-1f);
```

```
            Debug.Log("character dead");
            Unsubscribe(trackCheckpoints);
            EndEpisode();
            agentDead = false;
        }
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(redrunner.transform.localPosition);
        sensor.AddObservation(redrunner.GetComponent<Rigidbody2D>()
        .velocity.x);
    }

    public override void OnActionReceived(ActionBuffers
    ↪  actionBuffers)
    {
        redrunner.directionFloat =
        ↪  actionBuffers.ContinuousActions[0];
        var jump = actionBuffers.DiscreteActions[0];

        switch (jump)
        {
            case 0:
                redrunner.jumping = 0; // not jumping
                break;
            case 1:
                redrunner.jumping = 1; // jump button pressed
                AddReward(-0.1f);
                break;
        }
        AddReward(-0.1f / maxEnvironmentStep);
    }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        Time.timeScale = 1f;
        var discreteActionsOut = actionsOut.DiscreteActions;
        var continuousActionsOut = actionsOut.ContinuousActions;
        continuousActionsOut[0] =
        ↪  CrossPlatformInputManager.GetAxis("Horizontal");
```

```
        if (CrossPlatformInputManager.GetButtonDown ("Jump"))
        {
            discreteActionsOut[0] = 1;
        }
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Instakill"))
        {
            redrunner.Die();
            AddReward(-1f);
            EndEpisode();
        }
    }

    private void Unsubscribe(TrackCheckpoints tc)
    {
        Debug.Log("UNSUBSCRIBED TO " + tc);
        tc.OnPlayerCorrectCheckpoint -=
        ↪  TrackCheckpoints_OnPlayerCorrectCheckpoint;
        tc.OnPlayerWrongCheckpoint -=
        ↪  TrackCheckpoints_OnPlayerWrongCheckpoint;
    }
}
```

B.4. Classe RedCharacter

Modificato metodo `Update()` per implementare DRL.

```
void Update ()
{
        if ( !GameManager.Singleton.gameStarted ||
        ↪  !GameManager.Singleton.gameRunning )
        {
                return;
        }

        if ( transform.position.y < 0f )
        {
                Die ();
        }
```

```
// Speed
m_Speed = new Vector2 ( Mathf.Abs ( m_Rigidbody2D.velocity.x
↪  ), Mathf.Abs ( m_Rigidbody2D.velocity.y ) );

// Speed Calculations
m_CurrentRunSpeed = m_RunSpeed;
if ( m_Speed.x >= m_RunSpeed )
{
        m_CurrentRunSpeed = Mathf.SmoothDamp ( m_Speed.x,
        ↪  m_MaxRunSpeed, ref m_CurrentSmoothVelocity,
        ↪  m_RunSmoothTime );
}

// Input Processing
Move (directionFloat);
if (jumping == 1)
{
        Jump ();
}
if ( IsDead.Value && !m_ClosingEye )
{
        StartCoroutine ( CloseEye () );
}
if ( CrossPlatformInputManager.GetButtonDown ( "Guard" ) )
{
        m_Guard = !m_Guard;
}
if ( m_Guard )
{
        if ( CrossPlatformInputManager.GetButtonDown (
        ↪  "Fire" ) )
        {
                m_Animator.SetTrigger ( m_Actions [
                ↪  m_CurrentActionIndex ] );
                if ( m_CurrentActionIndex < m_Actions.Length
                ↪  - 1 )
                {
                        m_CurrentActionIndex++;
                }
                else
                {
```

```
                                m_CurrentActionIndex = 0;
                        }
                }
        }

        if ( Input.GetButtonDown ( "Roll" ) )
        {
                Vector2 force = new Vector2 ( 0f, 0f );
                if ( transform.localScale.z > 0f )
                {
                        force.x = m_RollForce;
                }
                else if ( transform.localScale.z < 0f )
                {
                        force.x = -m_RollForce;
                }
                m_Rigidbody2D.AddForce ( force );
        }
}
```

Bibliography

[1] R. Dechter. "Learning while searching in constraint-satisfaction problems". In: (1986), pp. 178–183.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[3] K. Shao et al. "A Survey of Deep Reinforcement Learning in Video Games". In: (2019), p. 2. arXiv: 1912.10944 [cs.MA].

[4] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), p. 529. DOI: 10.1038/nature14236.

[5] M. Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). URL: https://ojs.aaai.org/index.php/AAAI/article/view/11796.

[6] J. A. Arjona-Medina et al. "RUDDER: Return Decomposition for Delayed Rewards". In: (2019). arXiv: 1806.07857 [cs.LG].

[7] T. Pohlen et al. "Observe and Look Further: Achieving Consistent Performance on Atari". In: (2018). arXiv: 1805.11593 [cs.LG].

[8] J. Schulman, X. Chen, and P. Abbeel. "Equivalence Between Policy Gradients and Soft Q-Learning". In: (2018). arXiv: 1704.06440 [cs.LG].

[9] D. Ha and J. Schmidhuber. "Recurrent World Models Facilitate Policy Evolution". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[10] N. Nardelli et al. *Value Propagation Networks*. 2019. arXiv: 1805.11199 [cs.AI].

[11] J. Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4.

[12] R. J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696.

[13] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1928–1937.

[14] J. Schulman et al. "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1889–1897. URL: https://proceedings.mlr.press/v37/schulman15.html.

[15] J. Schulman et al. "Proximal Policy Optimization Algorithms". In: (2017). arXiv: 1707.06347 [cs.LG].

[16] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912.

[17] G. Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[18] C. Beattie et al. *DeepMind Lab*. 2016. arXiv: 1612.03801 [cs.AI].

[19] A. Juliani et al. *Unity: A General Platform for Intelligent Agents*. 2020. arXiv: 1809.02627 [cs.LG].

[20] E. Espié et al. "TORCS, The Open Racing Car Simulator". In: 2005.

[21] M. Kempka et al. "ViZDoom: A Doom-based AI research platform for visual reinforcement learning". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. 2016, pp. 1–8. DOI: 10.1109/CIG.2016.7860433.

[22] G. Synnaeve et al. "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games". In: (2016). arXiv: 1611.00625 [cs.LG].

[23] O. Vinyals et al. "StarCraft II: A New Challenge for Reinforcement Learning". In: (2017). arXiv: 1708.04782 [cs.LG].

[24] K. Kurach et al. "Google Research Football: A Novel Reinforcement Learning Environment". In: (2020). arXiv: 1907.11180 [cs.LG].

[25] J. Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: (2018). arXiv: 1506.02438 [cs.LG].