Unity Machine Learning Agents

A Self-driving Car

Andrea Salamone

1 Introduzione

Lo scopo del seguente lavoro è quello di sperimentare l'utilizzo del Deep Reinforcement Learning nello sviluppo di Intelligenze Artificiali per videogiochi tramite la libreria ML-Agents di Unity. Verranno, in primis, introdotti i concetti chiave del Reinforcement Learning insieme ai Policy Gradient Methods, per poi descrivere l'algoritmo PPO (Proximal Policy Optimization), utilizzato dalla libreria ML-Agents. Ad accompagnare la descrizione di tale libreria segue una breve introduzione alle CNN (Convolutional Neural Network). Verranno, infine, descritti gli esperimenti effettuati.

2 Reinforcement Learning

Il Reinforcement Learning (RL), o Apprendimento per Rinforzo, è uno dei tre paradigmi basilari del Machine Learning, insieme a Supervised Learning ed Unsupervised Learning, il cui obiettivo è quello di apprendere una *Policy* secondo la quale specifici agenti, detti *Learning Agent*, eseguono *azioni* all'interno di un *ambiente* con lo scopo di massimizzare una certa *ricompensa* ad essi assegnata in base alle loro prestazioni, fino al raggiungimento di un comportamento ottimale.

Un Agente esegue un processo di *decision making*, ad ogni generico tempo t, per scegliere ed eseguire l'*Azione* A_t più opportuna, ricevendo *Osservazioni* dall'ambiente, che andranno a costituire lo *Stato* S_{t+1} percepito dell'ambiente, e, infine, una *Ricompensa* R_{t+1}.

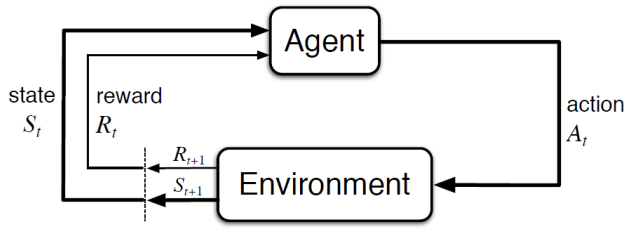


Fig. 1: Ciclo di interazione tra Agente ed Ambiente

Seguiranno adesso, le definizioni dei più comuni termini relativi al Reinforcement Learning.

A. Salamone, Metodi Statistici per l'Apprendimento, A/A 2019-2020, Università degli studi di Milano, Via Celoria 18, Milano, Italia
E-mail: andrea.salamone1@studenti.unimi.it

Definizione 1 (Ricompensa)**.** La Ricompensa *(Reward)* è un segnale numerico che indica ad un Agente l'ottimalità della situazione in cui si trova ad un istante di tempo t. Le Ricompense sono spesso proporzionali alla vicinanza dell'agente ad uno o più *goal*.

Una Ricompensa può essere *interna*, ovvero dettata dalla conoscenza che un agente possiede riguardo il suo *Stato*, o *esterna*, dettata dalla conoscenza che un agente possiede riguardo lo *Stato* dell'ambiente.

Come accennato precedentemente, l'obiettivo ultimo di un generico Agente è quello di massimizzare le ricompense ottenute durante l'interazione con l'ambiente. È bene notare che ad un agente non viene specificato quali azioni eseguire, ma è esso stesso ad esplorare l'ambiente provando un'azione dopo l'altra, verificando, quindi, quali di queste comportano una maggiore ricompensa. Inoltre, ogni azione compiuta potrebbe non influenzare unicamente la ricompensa ottenuta in un dato istante, ma anche lo stato successivo dell'ambiente, influenzando di conseguenza tutte le possibili ricompense successive.

Definizione 2 (Osservazioni ed Azioni)**.** Per *Osservazione* si intende una rappresentazione dell'ambiente che un agente riceve in un dato momento. Generalmente un'osservazione rappresenta un vettore di segnali numerici discreti o continui.

Un'*Azione* costituisce un segnale, emesso da un agente, verso l'ambiente, che può anche influenzare le future osservazioni.

Definizione 3 (Storia e Stati)**.** L'insieme dei cicli di interazione tra Agente ed Ambiente è detto *Storia*. Funzione della storia sono gli *Stati*, dell'Agente o dell'Ambiente, che costituiscono le loro rappresentazioni interne.

Gli *Stati dell'ambiente* raccolgono le informazioni che l'ambiente utilizza per generare Ricompense e Osservazioni da assegnare agli Agenti. Gli *Stati dell'agente* raccolgono le informazioni che un agente utilizza per determinare l'azione successiva e, eventualmente, per generare Ricompense interne.

Definizione 4 (Ambienti completamente o parzialmente osservabili)**.** Gli stati dell'agente e dell'ambiente possono essere differenti in quanto ciascuno di essi può possedere informazioni *private* e quindi non visibili dall'esterno. Questo comporta l'esistenza di *Ambienti*

completamente o parzialmente osservabili: un ambiente è *completamente osservabile* quando gli agenti hanno accesso (tramite osservazioni) a tutte le informazioni, rilevanti per l'apprendimento, riguardo lo Stato dell'ambiente; un ambiente è *parzialmente osservabile* nel caso contrario.

Per concludere questa sezione, è necessario un focus sul processo di *Decision Making* che caratterizza il problema del Reinforcement Learning, in particolare vengono distinti due approcci: *Model-Based* in cui l'Agente apprende un modello dell'Ambiente, utilizzato durante la fase di *Planning*, e *Model-Free* in cui l'Agente si occupa unicamente di valutare, ad ogni iterazione, l'Azione migliore in ogni Stato dell'Ambiente. Vanno, inoltre, definite due fasi del Decision Making:

- **Learning**: fase in cui avviene il ciclo di interazione tra Agente ed Ambiente, durante la quale l'Agente esplora l'Ambiente, inizialmente sconosciuto in modo tale da poter prima apprendere un comportamento ottimale e trovare poi una sequenza di Azioni correlate a Stati che mirano a massimizzare la Ricompensa, in questo caso si parla di approccio *Model-Free*, oppure, nel caso di un approccio *Model-Based*, l'Agente esplora prima l'Ambiente per apprendere un Modello di quest'ultimo, per poi prevederne i possibili Stati. Da notare, quindi, la presenza di due sotto-fasi durante il Learning (che sia Model-Based o Model-Free), queste sono rispettivamente chiamate fase di *Exploration* e fase di *Exploitation*.
- **Planning**: la fase di Planning è, in generale, esclusiva degli approcci Model-Based. Appreso un modello dell'Ambiente, l'Agente pianifica, a partire da esso, la politica da seguire ottimizzando ulteriormente il proprio comportamento.

2.1 Markov Decision Process

A questo punto della trattazione, verranno affrontati e descritti i *Processi Decisionali Markoviani* che descrivono formalmente uno scenario di Reinforcement Learning in presenza di *Ambienti completamente osservabili*. Un processo di questo tipo rappresenta situazioni in cui tutto ciò che avviene durante gli step futuri dipende soltanto dallo stato corrente, potendo quindi ignorare la Storia.

Definizione 5 (Markov Property)**.** La proprietà di Markov è definita come segue: "Il futuro è indipendente dal passato, dato il presente". Uno stato S_t è *Markoviano* se e soltanto se

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t] \tag{1}$$

Estendendo la Proprietà di Markov ad una sequenza di stati si ottiene un *Processo Markoviano*.

Definizione 6 (Markov Process)**.** Un Processo Markoviano (o *Markov Chain*) è una coppia $\langle \mathcal{S}, \mathcal{P} \rangle$ in cui:

- $\mathcal{S}$ è un insieme (finito) di stati
- $\mathcal{P}$ è una matrice probabilistica di transizione tra stati definita come:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s] \tag{2}$$

Aggiungendo ai Processi Markoviani i valori di Ricompense, otteniamo i *Markov Reward Process*, e aggiungendo ulteriormente Decisioni ed Azioni, possiamo parlare di *Markov Decision Process*.

Definizione 7 (Markov Decision Process)**.** Un Processo Decisionale Markoviano è definito da una tupla $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ in cui:

- $\mathcal{S}$ è un insieme finito di stati
- $\mathcal{A}$ è un insieme finito di azioni
- $\mathcal{P}$ è una matrice probabilistica di transizione tra stati definita come:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \tag{3}$$

- $\mathcal{R}$ è una funzione di ricompensa definita come:

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{4}$$

- γ è un fattore di sconto $\gamma \in [0, 1]$

Il nuovo termine γ, detto fattore di sconto, è utile per soppesare il valore delle ricompense future quando viene effettuato il calcolo del *Ritorno* ad ogni step t. Introduciamo, quindi, la definizione di *Ritorno*.

Definizione 8 (Return)**.** Il *Ritorno* G_t è la somma delle ricompense a partire dall'istante di tempo t, scontata mediante γ

$$G_t = R_{t+1} + R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5}$$

Possiamo, adesso, definire il concetto di *Policy*, strettamente legato alle Funzioni di Valore, definite successivamente ed utilizzate per stimare il valore di Ritorno a partire da uno Stato o da un'Azione.

Definizione 9 (Policy)**.** Una *Policy* π è una distribuzione sulle Azioni, condizionate dagli Stati:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{6}$$

La Policy è, in definitiva, ciò che definisce il comportamento di un Agente in dipendenza dallo Stato in cui esso si trova.

Definizione 10 (State Value Function e Action Value Function)**.** Le *State Value Function* $v_\pi(s)$ e le *Action Value Function* $q_\pi(s, a)$ rappresentano il Ritorno atteso a partire, rispettivamente, da uno Stato o da un'Azione, seguendo la Policy π

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{7}$$

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{8}$$

Come menzionato durante l'introduzione al Reinforcement Learning, il problema di questa tipologia di apprendimento consiste nel ricercare una *Policy Ottimale*, ovvero quella che che porta all'esecuzione delle

migliori Azioni da eseguire in dipendenza dagli Stati dell'Ambiente.

Definizione 11 (Optimal State Value Function e Optimal Action Value Function)**.** La *Optimal State Value Function* $v_*(s)$ e la *Optimal Action Value Function* $q_*(s,a)$ rappresentano, rispettivamente, la Massima State Value Function e la Massima Action Value Function tra tutte le Policy.

$$v_*(s) = \max_\pi v_\pi(s) \tag{9}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{10}$$

Ordinando tutte le Policy in base alle Value Function relative

$$\pi \geq \pi' \text{ se } v_\pi(s) \geq v_{\pi'}(s), \forall s$$

possiamo introdurre il seguente teorema sulle Policy ottimali:

Teorema 1 (Optimal Policy)**.** Per ogni Processo Decisionale Markoviano

- esiste una Policy Ottimale π_* che è maggiore o uguale a tutte le altre Policy, $\pi_* \geq \pi, \forall \pi$
- tutte le Policy ottimali ottimizzano le State Value Function, $v_{\pi_*}(s) = v_*(s)$
- tutte le Policy ottimali ottimizzano le Action Value Function, $q_{\pi_*}(s,a) = q_*(s,a)$

A questo punto è chiaro come una Policy Ottimale massimizzi il Ritorno atteso, per ottenere tale Policy, esistono tre principali metodi:

- metodi *Value-Based* per ricercare direttamente q_* o v_* come, per esempio, l'approccio ϵ*-greedy*;
- metodi *Policy-Based* che si basano sul parametrizzare direttamente la Policy per ricercare quella Ottimale tramite metodi di ascesa del gradiente (*Policy Gradient Methods*);
- metodi *Actor-Critic* che possono essere considerati come un misto dei due metodi precedenti.

2.2 Policy Gradient Methods

I metodi che andremo ad analizzare sono i *Policy Gradient Methods*, ai quali appartiene l'algoritmo di *Proximal Policy Optimization (PPO)* utilizzato dalla libreria Unity ML-Agents. Come accennato, i Policy Gradient Methods si basano sul parametrizzare direttamente la Policy:

$$\pi_\theta(s,a) = \mathbb{P}[a|s,\theta] \tag{11}$$

dove θ costituisce il vettore di parametri. Eseguito questo passaggio, è necessario comprendere quali siano i valori di θ che permettono di ottenere una Policy Ottimale, di conseguenza vengono utilizzate le *Policy Objective Functions* $J(\theta)$ per misurare l'effettiva qualità di una Policy. In presenza di ambienti episodici si utilizza lo *start value*

$$J_1(\theta) = V^{\pi\theta}(s_1) = \mathbb{E}_{\pi\theta}[v_1] \tag{12}$$

In presenza di ambienti continui (ambienti che persistono per un tempo indefinito, nei quali può anche non esistere uno stato iniziale) si utilizza l'*average value*

$$J_{avV}(\theta) = \sum_s d^{\pi\theta}(s) V^{\pi\theta}(s) \tag{13}$$

oppure l'*average reward per time-step*

$$J_{avR}(\theta) = \sum_s d^{\pi\theta}(s) \sum_a \pi_\theta(s,a) \mathcal{R}_s^a \tag{14}$$

in cui $d^{\pi\theta}(s)$ rappresenta una distribuzione stazionaria della Markov Chain per la Policy π_θ.

L'obiettivo è quello di trovare il valore di θ che massimizza $J(\theta)$, per raggiungere questo scopo, bisogna ottimizzare la funzione tramite *Ascesa del Gradiente*. Tale metodo consiste nel ricercare un punto di massimo locale in $J(\theta)$ risalendo il gradiente della policy con parametro θ:

$$\Delta\theta = \alpha \nabla_\theta J(\theta) \tag{15}$$

dove α rappresenta la dimensione del singolo passo di ascesa e $\nabla_\theta J(\theta)$ è il gradiente della policy

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix} \tag{16}$$

A questo punto, è bene notare come la funzione $J(\theta)$ dipenda sia dall'azione scelta, sia dalla distribuzione degli stati in cui le scelte avvengono, inoltre, entrambe le dipendenze sono influenzate dal parametro θ della policy. Per quanto riguarda le azioni scelte, conoscendo lo stato in un dato istante, è semplice calcolare gli effetti che i cambiamenti di θ hanno sulla scelta di tali azioni. Ma gli effetti che i cambiamenti della policy hanno sulla distribuzione degli stati variano in funzione dell'ambiente (in base alle osservazioni fatte), di conseguenza non è dato conoscere la sua evoluzione a meno che non si abbia un Modello (approcci Model-Based) di tale ambiente che, però, comporta una complicazione non banale dell'intero approccio. Il problema principale consiste quindi nell'utilizzare un metodo per stimare il gradiente della policy rispetto al parametro θ quando questo dipende dagli effetti che i cambiamenti della policy hanno sulla distribuzione degli stati e, fortunatamente, è possibile risolverlo, pur rimanendo in assenza di un modello dell'ambiente, con il *Policy Gradient Theorem*. Il teorema provvede un'espressione analitica del gradiente che non coinvolge la distribuzione degli stati.

Teorema 2 (Policy Gradient Theorem)**.** Data una Policy differenziabile $\pi_\theta(s,a)$ e data una qualsiasi delle *Policy Objective Function* J_1, J_{avV} o J_{avR}, il gradiente della policy è

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{17}$$

A questo punto è possibile introdurre il primo algoritmo di apprendimento basato sul Policy Gradient: il

Algorithm 1 REINFORCE

Inizializza arbitrariamente il vettore dei parametri θ.
forall *episodes* $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1$ *to* $T - 1$ **do**
    $\theta \leftarrow \theta + \alpha\gamma^t \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$
  end
end
return θ

Monte-Carlo Policy Gradient, considerato praticamente il medesimo dell'algoritmo REINFORCE.

Nell'algoritmo viene utilizzato il ritorno G_t per indicare la direzione del passo di aggiornamento, inoltre, notiamo che esso viene utilizzato al posto della value function, questo perchè viene considerato un campione "*unbiased*" della action value function $Q^{\pi_\theta}(s, a)$, poichè siamo in presenza di un valore atteso (quello della value function) all'interno di un altro valore atteso (quello del policy gradient theorem).

Essendo un algoritmo basato sull'approccio Monte-Carlo, ci occorre simulare l'intero episodio per assegnare un valore opportuno a G_t, questo può comportare un'alta varianza dei risultati e, di conseguenza, può portare ad un apprendimento molto lento. Per questo motivo possiamo generalizzare il Policy Gradient Theorem introducendo un *baseline* che viene sottratto alla action value function e quindi a G_t nell'algoritmo REINFORCE. Il baseline può assumere la forma di qualsiasi funzione o anche una variabile random, purchè non sia in funzione delle azioni, inoltre, viene generalmente considerato come una funzione $\hat{A}_t$ detta *Advantage Function*. Una buona advantage function può essere costituita dalla stima degli state value $\hat{v}(S_t, \omega)$, dove $\omega \in \mathbb{R}^d$ è un vettore di pesi che dovrà essere appreso come effettuato in precedenza con il vettore di parametri θ. Tale stima ci aiuta a discriminare il valore delle azioni in ogni stato poichè in presenza di stati il cui State Value è "buono" corrisponderà un'azione con un Action Value altrettanto "buono", il tutto vale anche nei casi "non buoni". Le

Algorithm 2 REINFORCE con Baseline

Inizializza arbitrariamente il vettore dei parametri θ.
Inizializza arbitrariamente il vettore dei pesi ω.
forall *episodes* $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1$ *to* $T - 1$ **do**
    $G_t \leftarrow$ ritorno a partire dallo step t
    $\delta \leftarrow G_t - \hat{v}(s_t, \omega)$
    $\omega \leftarrow \omega + \alpha_\omega \delta \gamma^t \nabla_\omega \hat{v}_\omega(s_t)$
    $\theta \leftarrow \theta + \alpha_\theta \delta \gamma^t \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$
  end
end
return θ

grandezze dei passi di ascesa del gradiente sono espresse da α_ω e α_θ, rispettivamente per ω e θ.

2.3 Proximal Policy Optimization

Definiti i principi e gli algoritmi base del settore, possiamo adesso definire il metodo *Proximal Policy Optimization* che si basa su ascesa stocastica del gradiente su più epoche per aggiornare i parametri θ della Policy π. Questo nasce dall'esigenza di avere un metodo che sia scalabile, efficiente sui dati e robusto, lasciando altri metodi quali *Q-Learning* o *Trust Region Policy Optimization (TRPO)* per casi più specifici. Il metodo PPO eredita stabilità ed affidabilità dai metodi TRPO, offrendo maggiore semplicità nell'implementazione e migliorandone le performance.

Di seguito viene illustrato come è possibile passare dall'algoritmo di REINFORCE a quello previsto dal metodo PPO. In primo luogo, il termine logaritmico del Policy Gradient Theorem viene sostituito con il rapporto $r_t(\theta)$:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{18}$$

ossia il rapporto tra la Policy attuale, con parametro θ, e la Policy con il vettore di parametri prima dello step di aggiornamento, il cui risultato sarà:

- $r_t(\theta) > 1$ quando l'azione prevista dalla Policy corrente è più probabile di quella prevista dalla vecchia Policy;
- $0 < r_t(\theta) < 1$ nel caso opposto.

Passiamo quindi all'ottimizzazione della funzione obiettivo:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t\right] \tag{19}$$

in cui troviamo la generica advantage function $\hat{A}_t$. A questo punto, però, è bene notare come i passi di ascesa del gradiente divengono fin troppo "lunghi" nel caso in cui l'azione prevista dalla policy corrente diviene sempre più probabile.

Il metodo TRPO prevede una serie di controlli volti a limitare il passo di ascesa del gradiente ma che, però, complicano il processo, è qui, infatti, che PPO giunge in aiuto utilizzando il *Clipped Surrogate Objective*:

$$J^{CLIP}(\theta) = \mathbb{E}_{\pi_\theta}[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{20}$$

la funzione consiste nel controllare effettivamente l'ampiezza del passo durante uno step ed, eventualmente, troncarlo tra $1-\epsilon$ ed $1+\epsilon$, dove un buon valore di ϵ è considerato 0.2. Come illustrato dai grafici della Figura 2, l'approccio consiste nel troncare il valore di r quando un aggiornamento porta a policy localmente ottimali, in maniera tale da evitare di compiere passi di ascesa del gradiente troppo grandi e, nel caso opposto, quando gli aggiornamenti portano a policy peggiori, vengono accettati passi di aggiornamento più grandi e che possano anche ricondurre ad una policy precedentemente ottima (almeno localmente).

A differenza dei Policy Gradient Methods basilari, e grazie alla *Clipped Surrogate Objective function*, PPO permette di effettuare l'ascesa del gradiente su epoche mul-

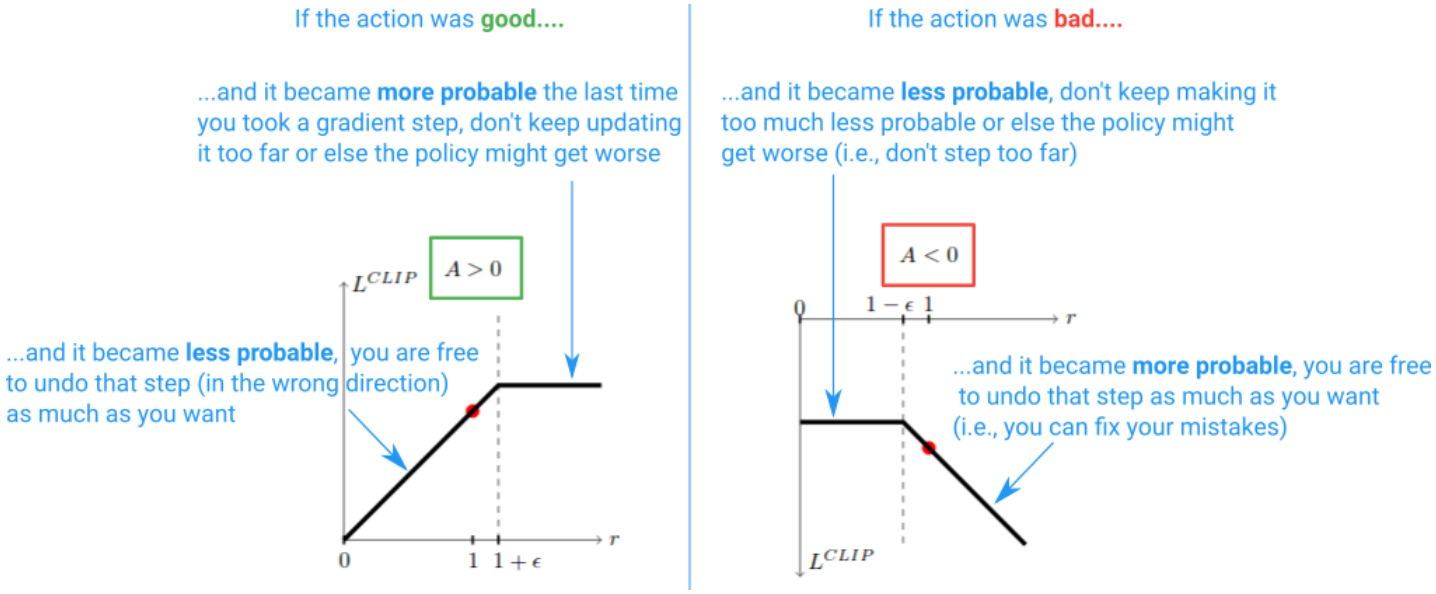


Fig. 2: Grafici della funzione di Clipping durante un singolo timestep in cui l'Advantage Function è positiva (a destra) e negativa (a sinistra). Il cerchio in rosso costituisce il punto dove comincia l'ottimizzazione.

tiple evitando di compiere passi di ascesa non costruttivi, questo perché l'algoritmo fa seguire la Policy ad N Attori in parallelo, ognuno dei quali raccoglie un numero T di esperienze, ottenendo NT differenti Advantage e Surrogate Objectives function. A questo punto, tali valori ottenuti vengono suddivisi in *mini-batch*, sui quali vengono eseguiti gli algoritmi di ascesa del gradiente per K epoche. Infine, otterremo che, per ogni iterazione, dopo aver campionato l'ambiente con la vecchia policy π_{old} e quando comincia ad essere eseguita l'ottimizzazione, la policy π risulterà uguale a π_{old}. Essendo le due policy uguali, nessun aggiornamento del passo di ascesa verrà troncato, di conseguenza otterremo degli aggiornamenti consistenti e si avrà la certezza che l'algoritmo permetta di apprendere qualcosa. Sappiamo, però, che la policy verrà aggiornata utilizzando più epoche, in questo modo l'obiettivo comincerà a raggiungere i limiti del Clipping ottenendo un gradiente che raggiungerà lo 0 per i campioni correnti, comportando un graduale arresto dell'apprendimento finchè non si passerà alla successiva iterazione collezionando nuovi campioni.

Algorithm 3 PPO, Actor-Critic Style

for *iterazione* = 1, 2, ... **do**
  for *attore* = 1, 2, ..., N **do**
    Esegui la Policy $\pi_{\theta_{old}}$ per T timesteps
    Calcola le Advantage function $\hat{A}_1, \ldots, \hat{A}_T$
  end
  Ottimizza J^{CLIP} rispetto θ, tramite discesa stocastica del gradiente con mini-batch di dimensione $M \leq NT$ e K epoche
  $\theta_{old} \leftarrow \theta$
end

Prima di passare alla descrizione della libreria ML-Agents di Unity, affrontiamo un'ultimo argomento: il Deep Reinforcement Learning.

2.4 Deep Reinforcement Learning

Si parla di Deep Reinforcement Learning quando si utilizza una Rete Neurale (di qualsiasi tipo) per approssimare una qualsiasi funzione propria del Reinforcement Learning. Nel caso di ML-Agents, la rete neurale viene utilizzata per approssimare la Policy.

Una Rete Neurale può essere addestrata per associare ad ogni Stato, una state value function oppure ad ogni coppia stato-azione, una action value function, potendo così trattare problemi di grandi dimensioni. In questo modo, alla rete neurale verranno date Osservazioni e Ricompense in input, ottenendo Azioni in output. Le osservazioni in input e le azioni in output sono costituite da vettori di valori discreti o continui.

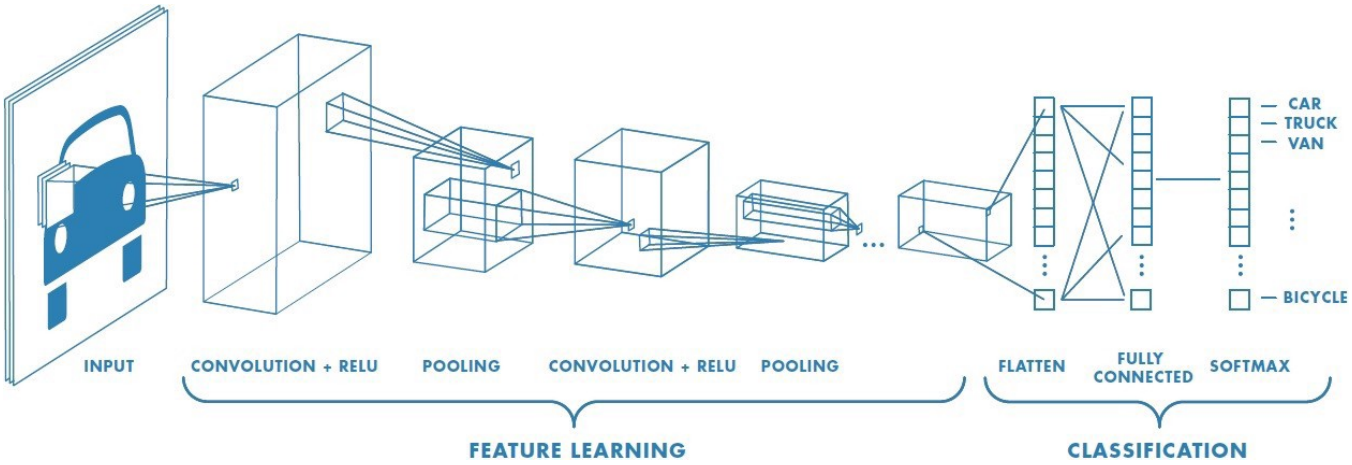
Nel nostro caso, all'interno della Rete Neurale, in combinazione con PPO viene utilizzato il classico algoritmo di *BackPropagation*.

2.5 Convolutional Neural Network (CNN)

La libreria ML-Agents, come vedremo successivamente, permette di dare agli Agenti delle *Vector Observations*, ossia vettori di valori discreti o continui, e/o *Visual Observations* che costituiscono immagini catturate costantemente da una *Camera*; gli esperimenti da me effettuati prevedono Agenti che possiedono osservazioni visive, in questo caso ML-Agents utilizza una CNN per l'apprendimento, di conseguenza è bene riservare loro una breve introduzione.

Una Rete Neurale Convoluzionale è un algoritmo che prende in input un'immagine, assegna dei gradi di importanza a vari aspetti dell'immagine e li differenzia

Fig. 3: Esempio di Convolutional Neural Network per la classificazione di veicoli.



tra loro, permettendone la classificazione. L'architettura di una CNN è ispirata alla conformazione della corteccia visiva del cervello umano: neuroni individuali rispondono a stimoli soltanto in una ristretta regione del campo visivo conosciuta come Campo Recettivo.

Possiamo immaginare come il compito di processare immagini sia computazionalmente intensivo, specie se queste raggiungono grandi dimensioni, di conseguenza, il ruolo di una CNN è quello di ridurre le immagini in una forma più semplice da processare e senza perdere le sue "features" che sono di fondamentale importanza per l'ottenimento di una buona predizione, di seguito verranno descritti i layer più importanti che costituiscono l'architettura di una CNN:

- **Input Layer**: contiene i valori dei pixel dell'immagine da processare, la sua dimensione è quella dell'immagine in input, includendo anche i tre canali di colore RGB se specificato.
- **Convolutional Layer**: è composto da una serie di neuroni connessi a regioni locali dell'Input Layer, ognuna di queste è chiamata Campo Recettivo del neurone e costituisce un filtro (o un kernel) con dimensioni stabilite dagli iperparametri del layer. Ogni neurone, ad ogni step della computazione, effettua una convoluzione tra l'immagine in input ed il filtro. L'output delle operazioni di convoluzione di ogni neurone viene aggiunto alla cosiddetta *Activation Map*, una matrice bidimensionale; è bene notare che, nel caso di immagini a colori, verranno generate tre Activation Map che vengono messe insieme per produrre l'output del layer. Generalmente, il primo Convolutional Layer è utile per estrarre feature di basso livello quali bordi, colori, orientamento del gradiente, ecc. Con l'aggiunta di ulteriori layer di questo tipo, è possibile estrarre, di volta in volta, feature di più alto livello, permettendo alla rete una maggiore comprensione dell'immagine.
- **Pooling Layer**: questo tipo di layer è generalmente applicato dopo ogni Convolutional Layer, il suo scopo è quello di ridurre la dimensione dell'output del layer precedente (effettuando un *downsampling*), utile per diminuire la potenza di calcolo necessaria per processare i dati, raggruppando le features trovate dalle convoluzioni del layer precedente. Anche questa volta vengono effettuate convoluzioni utilizzando un filtro che restituisce il valore massimo (*Max Pooling*) o il valore medio (*Average Pooling*) della porzione di immagine processata. Solitamente è utilizzato di più il Max Pooling poichè tenere i valori più alti delle porzioni di immagine ha come conseguenza diretta quella di ridurre eventuale rumore.
- **Fully-Connected Layer**: a questo punto l'immagine viene convertita in un vettore colonna e data in input ad una classica rete neurale feed-forward con connessioni complete tra ogni coppia di layer adiacenti alla quale, ad ogni iterazione dell'apprendimento, viene applicato l'algoritmo di backpropagation.

3 Unity ML-Agents

Unity ML-Agents è un plugin di Unity che permette la realizzazione di giochi o simulazioni che fungono da ambienti per addestrare Agenti intelligenti tramite diverse tecniche di machine learning (Reinforcement Learning nel nostro caso). Il plugin contiene tre componenti ad alto livello:

- **Learning Environment:** costituito da una qualsiasi *Scena* di Unity;
- **Python API:** è esterna a Unity e comunica con esso tramite un *External Communicator*, essa contiene tutti gli algoritmi utili per l'apprendimento;
- **External Communicator:** si trova all'interno del Learning Environment ed è utilizzato per connetterlo con la Python API.

Il **Learning Environment** contiene, a sua volta, tre componenti utili per organizzare una Scena su Unity: *Agent*, *Brain* e *Academy*.

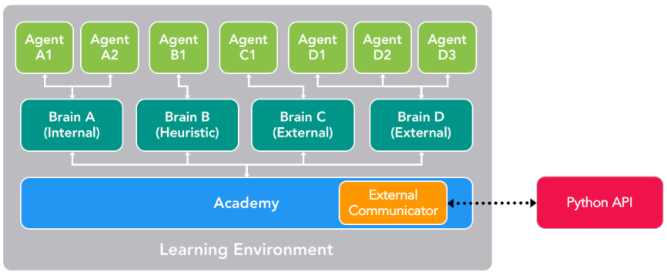


Fig. 4: Grafico d'esempio che mostra come comunicano le componenti di ML-Agents.

3.1 Agent

Un *Agent* è colui che interagisce con l'ambiente, in base alle osservazioni e alle ricompense ottenute, scegliendo le migliori azioni da eseguire. Per rendere tale un GameObject bisogna assegnare ad esso uno script che estende la classe *Agent*. Ad ogni agente può essere assegnato solamente un *Brain*: ad esso vengono passate le osservazioni dagli agenti e da esso gli agenti ricevono le azioni da compiere.

Le *Observations* collezionate dagli agenti sono di due tipi:

- **Vector Observation:** un vettore di informazioni costituito da un array di valori continui o discreti;
- **Visual Observation:** immagini catturate da uno o più oggetti di tipo *Camera* e/o da *Render Texture*.

Un agente ottiene osservazioni implementando il metodo `CollectObservations()` all'interno del quale viene invocato il metodo `AddVectorObs()` che specifica quali dati verranno utilizzati come osservazioni.

Le Azioni costituiscono vettori di informazioni continue o discrete e sono fornite ad un agente dal *Brain* ad esso assegnato; un'azione viene effettuata dagli agenti implementando il metodo `AgentAction()`.

3.2 Brain

La componente *Brain* incapsula il processo di decision making, fornendo agli agenti la Policy da seguire, ne esistono di tre tipi:

- **LearningBrain:** utilizzato per addestrare Reti Neurali o per utilizzare un modello già addestrato.
- **HeuristicBrain:** quando si implementa la logica dell'Agente via script.
- **PlayerBrain:** utilizzato per mappare l'input da tastiera del giocatore alle azioni dell'agente, utile per il testing.

Ogni brain contiene una serie di parametri utili per la definizione di *Vector Observation*, *Visual Observation* e *Vector Actions*, è bene notare che un Brain con una precisa configurazione di parametri va assegnato ad Agenti con una configurazione corrispondente.

3.3 Academy

Un'*Academy* gestisce tutti gli oggetti di tipo Agent e Brain presenti in una Scena di Unity, ogni scena può contenere soltanto un'Academy. Per creare un oggetto di tipo Academy bisogna assegnare ad esso uno script che eredita dalla classe `Academy` di ML-Agents; se non è necessario specificare cosa avviene quando viene chiamata una delle funzioni della classe Academy, è possibile lasciare vuoto lo script, ogni metodo che andremo ad osservare, verrà chiamato automaticamente sotto determinate condizioni.

Ogni Academy va inizializzata tramite il metodo `InitializeAcademy()` che imposta l'ambiente e tutti gli agenti al loro stato iniziale. Tramite il metodo `AcademyReset()` avviene il Reset dell'ambiente ad ogni Episodio dell'esecuzione, un Reset può avvenire automaticamente una volta che tutti gli agenti hanno portato a termine il loro loop di esecuzione (ovvero quando viene chiamato il relativo metodo `Done()`) o manualmente, chiamando il metodo `Reset()`; è bene notare che il Reset dell'ambiente non implica necessariamente che questo ritorni alle condizioni di inizializzazione. Infine, un'Academy permette di controllare direttamente l'ambiente tramite il metodo `AcademyStep()`, chiamato ad ogni step della simulazione, tramite il quale è possibile specificare i cambiamenti dell'ambiente tra uno Step e l'altro.

3.4 Il processo di apprendimento e simulazione

Apprendimento e simulazione vengono orchestrati dall'*Academy* che, come descritto in precedenza, gestisce gli Agenti presenti sulla scena per far procedere la simulazione. Durante l'apprendimento, il processo di apprendimento fornito dalle API Python esterne comunica con l'Academy, eseguendo una serie di episodi mentre colleziona dati volti ad ottimizzare il modello di rete neurale. L'Academy gestisce il loop di simulazione come segue:

1) Viene chiamato il metodo `AcademyReset()` della classe Academy.
2) Il metodo `AgentReset()` viene chiamato per ogni Agent presente sulla scena.
3) Il metodo `CollectObservations()` viene chiamato per ogni Agent presente sulla scena.
4) Utilizza il Brain di ogni Agent per decidere la prossima azione che questi andranno a compiere.
5) Viene chiamato il metodo `AcademyStep()` della classe Academy.
6) Per ogni agente presente sulla scena che non è settato a *Done*, viene chiamato il metodo `AgentAction()` a cui viene passata l'azione scelta dal Brain dell'Agent.
7) Nel caso in cui un Agent raggiunge il numero massimo di step, indicato dal parametro `MaxStep`, o viene settato a *Done*: se non viene selezionato il flag `ResetOnDone` di un Agent, viene chiamato il

metodo `AgentOnDone()`, altrimenti viene chiamato dall'Academy il metodo `AgentReset()`.

3.5 Python API

Il package Python fornito con ML-Agents contiene una API di basso livello, denominata `mlagents.envs`, fondamentale per controllare il loop di simulazione degli Agenti in un ambiente Unity. L'API viene utilizzata dagli algoritmi di apprendimento forniti da ML-Agents, ma permette anche di scrivere interi programmi Python.

Gli oggetti chiave della API Python sono:

- **UnityEnvironment**: l'interfaccia principale tra l'eseguibile di Unity e il codice implementato, è utilizzata per inizializzare e controllare una simulazione o una sessione di apprendimento.
- **BrainInfo**: contiene tutti i dati ottenuti dagli agenti durante una simulazione, come osservazioni e ricompense.
- **BrainParameters**: descrive i dati contenuti in un oggetto *BrainInfo*, per esempio la lunghezza degli array di osservazioni.

Una volta caricato l'ambiente di apprendimento denominato `env`, è possibile interagire con esso con i seguenti comandi:

- **Print**, comando `print(str(env))`: per stampare i parametri relativi all'ambiente attuale e ad i Brain assegnati agli agenti.
- **Reset**, comando `env.reset()`: resetta l'ambiente e restituisce un dizionario che mappa gli oggetti BrainInfo alle informazioni contenute nei Brain.
- **Step**, comando `env.step(action)`: invia un segnale all'ambiente, fornendo un insieme di azioni (il parametro `action`) agli agenti presenti nel dizionario attuale (ottenuto tramite il precedente Step o Reset); restituisce un nuovo dizionario che mappa gli oggetti BrainInfo alle informazioni contenute nei Brain.
- **Close**, comando `env.close()`: invia un segnale di spegnimento all'ambiente di apprendimento, chiudendo la socket di comunicazione.

4 Iperparametri

Per addestrare con successo un modello tramite Reinforcement Learning è spesso necessario calibrare una buon numero di iperparametri:

- **`batch_size`**: è il numero di esperienze da utilizzare quando si effettua un passo di Ascesa del Gradiente. Tale parametro dovrebbe tipicamente essere una frazione del `buffer_size`.
- **`beta`**: corrisponde alla forza dell'*entropy regularization*. L'entropia è aggiunta alla funzione obiettivo e viene pesata da β, modificando i valori del peso viene stabilito quanto l'algoritmo tende a massimizzare l'entropia. Tutto ciò viene utilizzato per evitare di rendere troppo probabile un'azione rispetto ad altre, cosa che potrebbe portare l'algoritmo di apprendimento a fermarsi in ottimi locali.
- **`buffer_size`**: corrisponde a quante esperienze (osservazioni, azioni e ricompense ottenute da un agente) devono essere raccolte prima di eseguire qualsiasi tipo di apprendimento. Tale parametro dovrebbe tipicamente essere un multiplo del `batch_size`.
- **`epsilon`**: indica il valore di soglia utilizzato nella *Clipped Surrogate Objective* dell'algoritmo PPO.
- **`hidden_units`**: utilizzato per stabilire quante unità sono presenti all'interno di ogni hidden layer della rete neurale.
- **`lambd`**: è il parametro λ utilizzato nel calcolo della *Generalized Advantage Function*.
- **`learning_rate`**: influenza la grandezza dei singoli step di ascesa del gradiente.
- **`max_steps`**: è il numero massimo di step che il processo di apprendimento può eseguire.
- **`normalize`**: va settato a `true` se è necessario normalizzare il vettore delle osservazioni in input.
- **`num_epoch`**: stabilisce quante volte, durante l'ascesa del gradiente, l'algoritmo considera il buffer di esperienze collezionate.
- **`num_layers`**: indica il numero di hidden layers presenti nella rete neurale.
- **`time_horizon`**: corrisponde a quanti step di esperienza vengono collezionati per ogni agente prima di aggiungerli al buffer delle esperienze. È utile quando viene calcolato il ritorno poichè serve a decidere quanto avanti nel tempo bisogna considerare le ricompense.

```yaml
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

Fig. 5: File .yaml che contiene gli iperparametri di default fornito dalla libreria ML-Agents.

Tutti gli iperparametri che non sono stati descritti sono utilizzati per tipologie di learning che si discostano dallo scopo del progetto. Alcuni parametri sono relativi all'utilizzo di reti neurali ricorrenti, altri per impostare i valori relativi agli *Intrinsic Reward*, ovvero ricompense definite anche al di fuori dell'ambiente unity.

5 Esercitazione: BalanceBall

BalanceBall consiste nella replica di un esempio fornito dalla documentazione di ML-Agents, l'esercizio è stato effettuato con lo scopo di apprendere le basi della libreria. L'ambiente è costituito da diverse copie di un agente che hanno come obiettivo quello di mantenere in equilibrio, il più a lungo possibile, una pallina posta sopra di essi, senza farla cadere, ruotando orizzontalmente o verticalmente.

Ogni agente riceve una ricompensa positiva per ogni step trascorso mantenendo la pallina in equilibrio, in particolare la ricompensa è di +0.1 per ogni step; nel caso in cui, invece, un agente fa cadere la pallina, questo viene resettato, riportando la pallina sopra di esso, e gli viene assegnata una ricompensa negativa pari a -1.0.

Tutti gli agenti dell'ambiente condividono lo stesso *Brain*, che riceve le seguenti osservazioni:

- la rotazione dell'agente cubo;
- la posizione della pallina;
- la velocità della pallina;

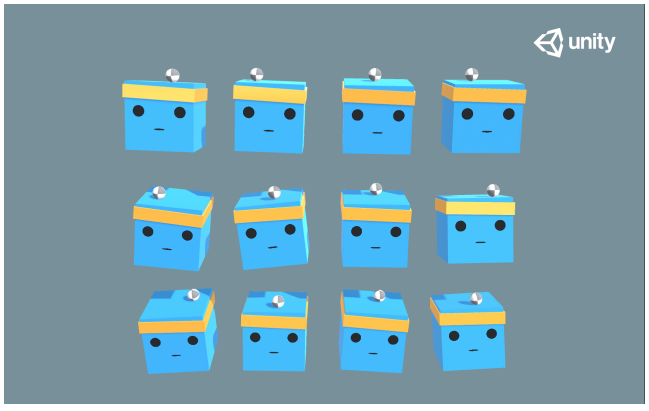


Fig. 6: Ambiente Unity di BalanceBall.

6 Esperimento: Self-Driving Car

L'esperimento consiste nell'ottenere un agente che riesca ad affrontare, correttamente e alla massima velocità possibile, un tracciato stradale.

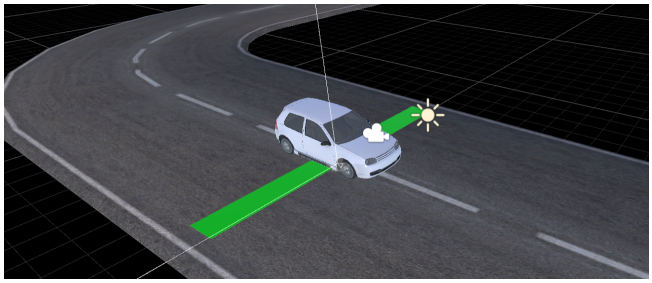


Fig. 7: L'agente automobile in posizione di partenza sul tracciato stradale.

6.1 Dettagli implementativi

L'interazione con l'ambiente, da parte dell'agente, avviene tramite due tipi di **Azioni**, entrambe costituite da valori continui:

- **Acceleratore/Freno**: assume valori che vanno da -1 a +1 e che, rispettivamente, rappresentano la massima potenza frenante e la massima accelerazione positiva. È bene notare come non sia contemplata una possibile marcia indietro del veicolo, in quanto ritenuta non necessaria per gli scopi della simulazione.
- **Sterzo**: assume valori che vanno da -1 a +1 e che rappresentano il massimo angolo di sterzo raggiungibile dalle ruote anteriori.

L'agente ottiene informazioni sull'ambiente tramite **Visual Observations**, pertanto quest'ultimo è costituito unicamente da un tracciato circondato da uno sfondo nero per semplificare il più possibile il riconoscimento del percorso. Nel dettaglio, ad ogni step, l'agente riceve osservazioni visive da una camera posta circa all'altezza del parabrezza anteriore ed inclinata per riprendere l'asfalto e l'orizzonte. Poiché, ai fini dell'apprendimento, è utile discernere unicamente i confini della strada, senza informazioni sul colore, le immagini vengono catturate in scala di grigi e di dimensione 32x24, in questo modo viene anche ridotto il carico computazionale del processo di apprendimento.

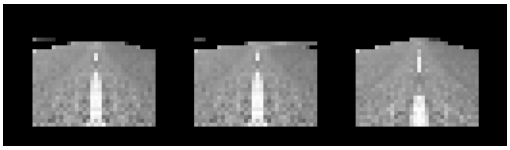


Fig. 8: Esempi di Visual Observations ricevute dall'agente.

In concomitanza con le Visual Observation, l'agente osserva, ad ogni step, la sua *Forward Velocity*, ossia la velocità misurata lungo il senso di marcia dell'automobile, in modo tale da poter decidere quanto effettivamente accelerare o frenare; trattasi quindi di una **Vector Observation** contenente valori continui.

Per raggiungere lo scopo prefissato, sono stati realizzati due ambienti di apprendimento, il primo contiene

quattro tracciati tutti uguali con quattro agenti, il secondo è costituito da tre differenti tracciati duplicati una volta, per un totale di sei tracciati e sei agenti. Tutti gli agenti in entrambi gli ambienti condividono lo stesso *Brain*, il fatto di avere più istanze di tracciati e agenti è utile per **parallelizzare** il processo di apprendimento del *Brain* che risulterà più rapido in quanto quest'ultimo, ad ogni step, collezionerà esperienze multiple.

Il motivo per cui sono stati realizzati due differenti ambienti è dato dal fatto che l'apprendimento eseguito soltanto su un singolo tracciato comporta l'ottenimento di un modello di rete neurale capace di percorrerlo quasi alla perfezione ma che, in presenza di tracciati differenti tra loro, può non risultare ottimale, pertanto, in assenza di un efficace generatore procedurale di percorsi, si è passato ad un ambiente contenente differenti tracciati, ottenendo, quindi, un modello migliore.

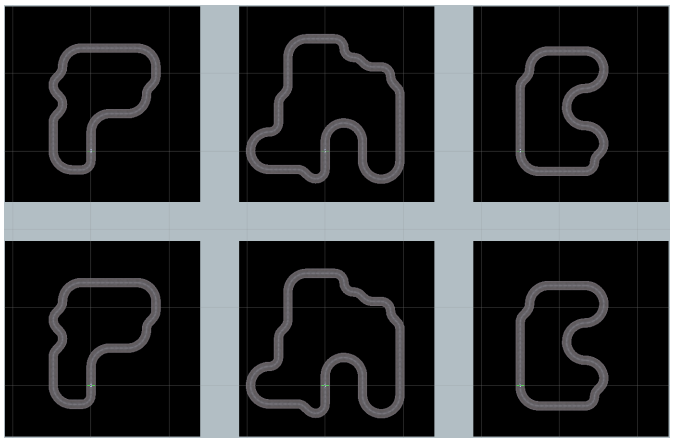


Fig. 9: Ambiente di apprendimento finale con tre diversi tracciati.

All'interno di entrambi gli ambienti, gli agenti vengono resettati automaticamente nel caso in cui questi escano fuori dal percorso, ricevendo una ricompensa negativa, o dopo un numero massimo di step (indicati dal parametro `MaxStep`) pari a 10.000. La presenza di un tetto massimo di step per ogni episodio permette di concludere gli episodi iniziali (altrimenti potenzialmente infiniti) in cui l'entropia è elevata e, di conseguenza, le azioni degli agenti costituiscono movimenti irrisori, inoltre, un elevato valore di `MaxStep` è utile per far permanere gli agenti sul tracciato per un tempo sufficiente a massimizzare la propria velocità.

6.2 Sistema di Ricompense

Dato l'obiettivo dell'agente, è necessario assucurarsi che questo acceleri per completare effettivamente il tracciato, pertanto, per incentivare ciò, è stata stabilita una ricompensa positiva calcolata come $V/5$, dove V rappresenta il valore assunto dalla velocità di marcia del veicolo (la *Forward Velocity* precedentemente menzionata). Di contro, per evitare frequenti frenate, è stata assegnata una piccola ricompensa calcolata come $A/10$, dove A rappresenta il valore dell'azione **Acceleratore/Freno** che, come precedentemente descritto, può assumere sia valori positivi che negativi, questo tipo di ricompensa viene assegnata solamente nel caso in cui i valori dell'azione siano negativi.

Ogni qualvolta un agente esce fuori dal tracciato, viene impostato un valore di ricompensa negativa pari a -1 e viene immediatamente concluso l'episodio corrente, comportando un reset dell'ambiente.

Infine, è stato scelto di assegnare un'ulteriore ricompensa negativa pari a -2 nel caso in cui l'automobile si ribalta a causa di una sterzata effettuata a velocità elevate (nel dettaglio quando la rotazione del veicolo sull'asse Z, ovvero l'asse corrispondente alla direzione del senso di marcia, supera una certa soglia). Questo tipo di ricompensa è stata introdotta durante le prime fasi dello sviluppo, quando non era stato ancora stabilito un appropriato sistema di simulazione della fisica del veicolo. Allo stato attuale, è stato scelto un sistema fisico che non permette al veicolo di ribaltarsi come precedentemente menzionato, risultando però in una simulazione meno realistica. Questo tipo di ricompensa è stata mantenuta per poter apportare modifiche a tale sistema ed effettuare comunque l'apprendimento in una situazione più realistica.

6.3 Iperparametri e Statistiche

I primi esperimenti di learning sono stati effettuati utilizzando gli ipermarametri di default (Figura 5) ma con il parametro `max_steps` impostato a 500.000 poichè il valore iniziale interrompeva l'apprendimento troppo presto. Con tali parametri, è stato ottenuto un modello soddisfacente che riesce a percorrere tutti e tre i tracciati senza un minimo errore. A questo punto, il lavoro sugli iperparametri ha avuto come scopo quello di ridurre i tempi di apprendimento. Sono stati prima modificati i valori di `batch_size` e `buffer_size`, per eseguire l'apprendimento più spesso, velocizzando, di conseguenza, il cambio di policy nei casi in cui un agente cade fuori dallo stesso punto del tracciato più volte consecutivamente. Trattandosi di un problema di apprendimento in cui l'ambiente è totalmente statico, poichè il percorso che ogni agente percorre non cambia mai e non vi sono elementi dinamici al suo interno, è stato deciso di dare una maggiore importanza alle esperienze collezionate dagli agenti, pertanto sono stati modificati i valori di `time_horizon` e `num_epoch` rispettivamente per conservare più esperienze nel buffer, prima di eseguire l'apprendimento, e per considerarle più frequentemente.

```yaml
default:
    trainer: ppo
    batch_size: 512
    beta: 1.0e-2
    buffer_size: 4096
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
    memory_size: 256
    normalize: false
    num_epoch: 6
    num_layers: 2
    time_horizon: 128
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

Fig. 10: Iperparametri utilizzati per Self-Driving Car.

I risultati ottenuti con questi parametri sono stati molto soddisfacenti in quanto la durata del processo di apprendimento è stata ridotta quasi della metà e ha portato ad un modello al pari di quello precedente, a riprova di ciò seguono due grafici in cui è possibile discernere le ricompense ottenute durante i due processi di apprendimento.

Utilizzando i parametri di default, l'apprendimento comincia a convergere una volta superati i 235.000 passi, cosa che avviene a circa 100.000 step prima con i nuovi parametri.

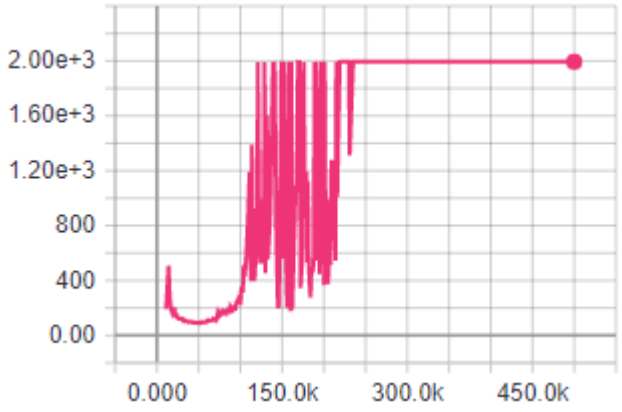


Fig. 11: Cumulative Reward ottenuto con i parametri di default.

È, inoltre, interessante notare come, con i parametri di default, vi sia un intervallo di step (lungo circa 100.000 step) in cui l'algoritmo non riesce a trovare una policy che riesca a far percorrere completamente il percorso a tutti gli agenti, cosa che non avviene con i nuovi parametri.

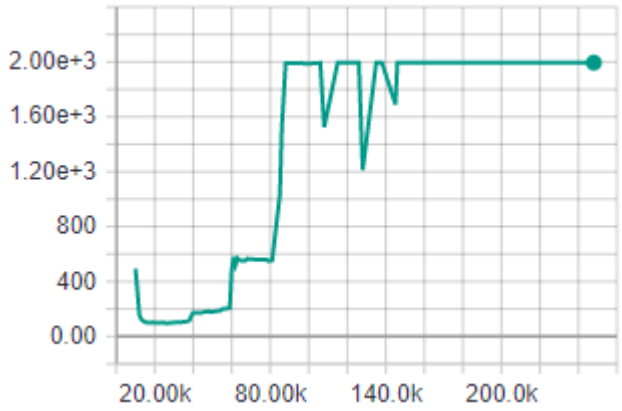


Fig. 12: Cumulative Reward ottenuto con i parametri modificati appositamente per l'esperimento.

Conservando più esperienze nel buffer, l'algoritmo passa a policy migliori non appena si verificano casi in cui gli agenti percorrono efficacemente tutto il loro percorso, per questo motivo si può osservare un grande aumento delle ricompense che rimane pressochè tale negli step successivi. Vengono, inoltre, messi a confronto i grafici di Learning Rate, a riprova dell'efficacia dei nuovi parametri:

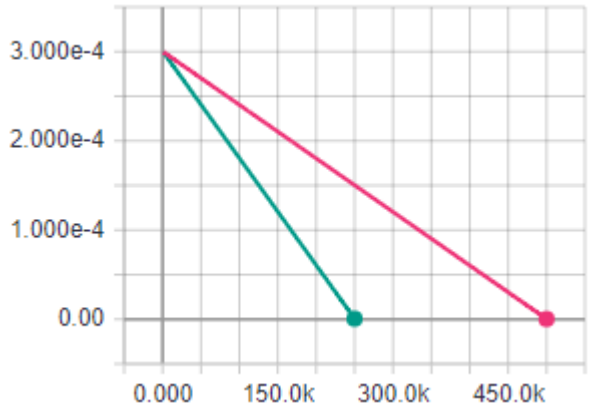


Fig. 13: Learning Rate con i parametri di default, in rosa, e con i nuovi parametri, in verde.

La stessa cosa viene fatta riguardo la stima del valore medio degli stati visitati dagli agenti, che aumenta in modo più efficace con i nuovi parametri rispetto a quelli di default:

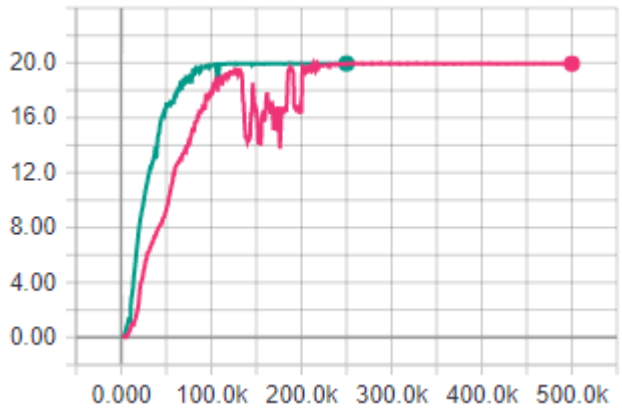


Fig. 14: Extrinsic Value Estimate con i parametri di default, in rosa, e con i nuovi parametri, in verde.

Di seguito verranno mostrate ulteriori statistiche relative all'esperimento effettuato con i parametri migliori:

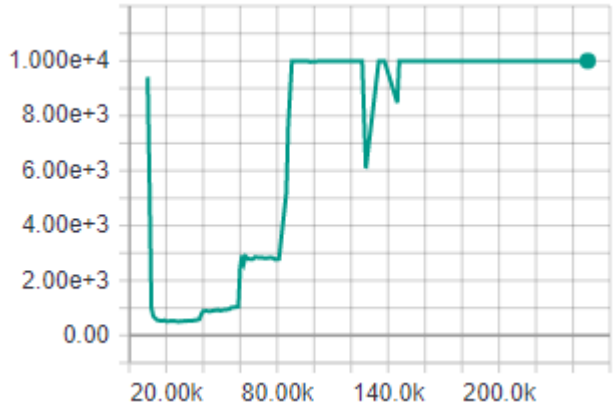


Fig. 15: Durata degli episodi.

La durata degli episodi è strettamente legata alle ricompense accumulate durante un episodio in quanto questo termina se un agente esce fuori dal percorso, di conseguenza, episodi di lunghezza massima comporteranno una ricompensa massima.

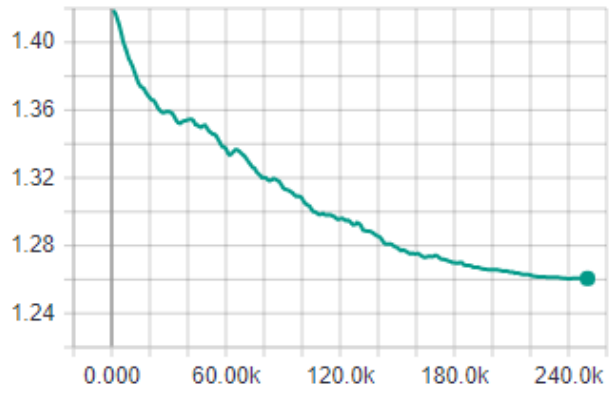


Fig. 16: Valori di Entropia della Policy.

L'Entropia della Policy decresce in modo piuttosto graduale, ciò dimostra come, altrettanto gradualmente, l'agente riesce a catturare un comportamento ottimo per ogni stato della simulazione. È stato scelto un alto valore del parametro β per permettere agli agenti di "superare" più rapidamente la fase iniziale in cui il veicolo rimane pressoché immobile.

6.4 Conclusioni e sviluppi futuri

Nonostante gli iperparametri possano essere modificati e testati per un periodo indefinito, con quelli attuali l'esperimento ha portato a risultati soddisfacenti, producendo degli agenti che riescono a destreggiarsi bene all'interno di una grande quantità di tracciati stradali. È stato anche realizzato un nuovo tracciato "sconosciuto" agli agenti per testare l'efficacia del modello prodotto, è possibile osservarlo nel **video in allegato** alla relazione.

Sarebbe stato molto utile avere a disposizione un generatore procedurale di tracciati per poter ottenere agenti ancora più performanti, non ne è stato realizzato uno ex-novo poichè questo avrebbe allungato considerevolmente i tempi di sviluppo dell'intero progetto.

Il lavoro svolto può essere facilmente ampliato introducendo scenari di apprendimento ancora più interessanti:

- è possibile introdurre oggetti all'interno dei percorsi generati, che verrebbero evitati o raccolti dagli agenti, questo sarebbe possibile aggiungendo nuove telecamere all'agente, con l'unico scopo di riconoscere tali oggetti;
- sarebbe interessante osservare il comportamento degli agenti in percorsi di maggiore "difficoltà", aggiungendo, ad esempio, parti di tracciato più strette o in salita/discesa e, possibilmente, differenti tipi di asfalto (con diversi valori di attrito) che l'agente andrebbe a preferire o evitare.
- l'introduzione di una nuova ricompensa al completamento di un giro del tracciato potrebbe portare a risultati differenti, il tutto, se unito alle aggiunte del punto precedente, comporterebbe agenti ancora più complessi e con un differente goal: completare i giri del tracciato nel minor tempo possibile. In questo caso gli agenti imparerebbero a preferire le parti di tracciato maggiormente percorribili, il tutto introducendo anche delle scorciatoie;
- aggiungere agenti multipli all'interno di uno stesso tracciato porterebbe a delle vere e proprie gare automobilistiche, questo sarebbe possibile introducendo un sistema di ricompense più approfondito che consideri, per esempio, le collisioni con gli altri veicoli e i sorpassi subiti/effettuati, insieme ad un nuovo sistema di telecamere per le Visual Observations che permetta il riconoscimento, oltre che del tracciato, degli altri veicoli, similmente a come è stato accennato nel primo punto.

References

[1] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents", 2018.

[2] R. S. Sutton and A. G. Barto, ”Reinforcement learning: An introduction. MIT press”, 2018.
[3] D. Silver, Reinforcement learning course: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms”, 2017.
[5] Topic about PPO: https://stackoverflow.com/questions/46422845/what-is-the-way-to-understand-proximal-policy-optimization-algorithm-in-rl
[6] Stanford’s Class about Convolutional Neural Networks: http://cs231n.github.io/convolutional-networks/