Creazione di un AI per la guida autonoma tramite la tecnica del Q-Learning

Francesco Romeo (N97000289), Andrea Pollastro (N97000282)

26 agosto 2019

Indice

Capitolo 1

Introduzione al problema

Il problema che si è affrontato in questo progetto riguarda la creazione di un intelligenza artificiale che riesca a guidare un auto in modo autonomo. Il problema in questione assume che l'automobile di riferimento sia una comune 4 ruote (con dimensioni normali) e che la careggiata su cui si guida sia una strada a senso unico delimitata da guardrails.

L'automobile è dotata di una serie di sensori anteriori che coprono uniformemente la visuale in avanti e di lato, i quali, sono ingrado di rilevare con precisione la distanza dell'automobile da eventuali ostacoli che essa incontra. Per muoversi nello spazio della carreggiata, l'automobile, è provvista di due tipi di spostamento: accelerazione e sterzata. Tramite questi due spostamenti essa può moversi in una qualsiasi direzione (modificando il grado di sterzata da sinistra verso destra) con una qualsiasi velocità (regolando l'accelerazione, o la decelerazione, in caso di frenata).

L'obiettivo che si perseguirà in generale, dunque, sarà quello di riuscire a creare un AI che riesca a guidare un automobile in un qualsiasi tracciato che rispetti le proprietà prima descritte.

Capitolo 2

Tecnologie e Tools per lo sviluppo

Prima di descrivere quali sono state le soluzioni adottate, si preferisce fare una breve introduzione sulle tecnologie adottate, al fine di chiarire meglio il contesto in cui si è sviluppato il progetto e le motivazioni che si celano dietro alcune scelte.

Partiamo subito con il dire che lo sviluppo del progetto è avvenuto totalmente in un ambiente simulato: ciò ha permesso di poter ignorare una serie di problemi non banali che si sarebbero affrontati in un caso reale, e di incentrare lo studio unicamente sulle tecniche di apprendimento dell'AI. Si mostra in questo capitolo l'insieme delle tecnologie utilizzate per sviluppare il progetto.

2.1 Unity

Dietro la parola "ambiente simulato" si nascondono una serie di aspetti, dati per scontati, che riguardano la gestione della realtà virtuale; tali aspetti, includono la gestione della grafica, passano per l'interfacciamento dei comandi con un agente (colui che impartisce i comandi di guida), fino ad arrivare ai complicati calcoli che servono per generare la fisica su cui si basa l'ambiente.

Tutti questi aspetti, ed altri, sono stati risolti tramite l'utilizzo di un motore grafico molto noto: Unity.

Come appena detto, dunque, Unity è un motore grafico 3D che mette a disposizione degli utenti una serie di tools che supportano lo sviluppo di un videogioco, e riesce a farlo in un modo molto semplice ed intuitivo. Ciò che segue, sarà una breve introduzione a Unity ed ai tools che si sono sfruttati.

2.1.1 Funzionamento di base

Si possono descrivere le dinamiche di progettazione in Unity facendo riferimento alle seguenti parole chiave:

- Scena
- GameObject
- Components

La scena non è altro che il "quadro" all'interno del quale si svolgerà il gioco, e che conterrà una serie di oggetti che faranno parte del gioco: i GameObject. Qualsiasi elemento si trovi all'interno di una scena è un GameObject per definizione.

Ogni GameObject nel suo "nucleo" più interno è uguale a tutti gli altri: tutti sono dei semplici punti nello spazio. Ciò che differenzia i GameObject tra di loro sono invece i Components, che sono appunto i comportamenti assunti dal GameObject; essi ne descrivono non solo la forma, ma anche le attività che tale oggetto deve compiere, e come esso deve reagire al verificarsi di eventi esterni.

E' dunque immediata l'idea che ciò su cui bisogna concentrarsi non è la creazione dei GameObject, ma bensì sui components ad esso associati.

2.1.2 I Components

I components che definiscono il comportamento dei GameObject sono sono altro che degli script che utilizzano come linguaggio di riferimento C#. Ogni qualvolta si definisce un nuovo script da "attaccare" ad un GameObject si sta, in pratica, definendo una nuova classe; tale classe, al fine di essere un component, ha il vincolo di dover estendere la classe **MonoBehaviour**, che è la superclasse di qualsiasi component.

Ogni component ha, inoltre, l'obbligo di dover implementare due metodi ereditati da tale superclasse:

- Start(), che è un metodo che viene richiamato un unica volta: quando il GameObject viene creato.
- Update(), che è il metodo che viene richiamato automaticamente ad ogni fotogramma della scena (in pratica si esegue decine di volte al secondo).

Il modo in cui andreamo a creare l'AI, obiettivo di questo progetto, sarà proprio lavaorando sul metodo Update(), che eseguirà tutte le operazioni necessarie per imparare a guidare.

2.1.3 Classi necessarie per lo sviluppo

L'ultima cosa che si vuole nominare circa il funzionamento di Unity sono le classi fondamentali per lo sviluppo e l'interazione dei GameObject nella scena. Tra le tante disponibili e utilizzare, quelle che si vogliono nominare in questa sezione sono solo:

- La classe Colliders
- La classe RigidBody

Per quanto riguarda la classe Colliders, essa serve principalmente per gestire, appunto, le collisioni tra oggetti e mette a disposizione una serie di metodi di supporto alla gestione tali avvenimenti. Questa classe è stata di fondamentale importanza per decretare quando l'AI fallisce nel suo scopo, ovvero, quando l'automobile si schianta in un ostacolo.

La classe RigidBody risulta invece necessaria per gestire lo spostamento dell'automobile: in generale questa classe permette ad un oggetto di interagire con tutta la fisica della scena. L'accelerazione, le sterzate, le frenate e tutte le altre azioni, sono tutte conseguenze dei calcoli che avvengono tramite l'utilizzo di questa classe.

2.2 Il tool ProBuilder

ProBuilder è un tool che consente di creare ed editare oggetti tridimensionali direttamente su Unity. Esso mette a disposizione una vasta gamma di elementi, come edifici, automobili, armi e tanto altro; ognuno di questi elementi può essere modificato e personalizzato, non solo nel suo aspetto, ma anche nel suo funzionamento.

L'utilizzo di questo tool ha permesso una rapida costruzione sia della mappa di gioco che dell'autovettura.

2.3 Anaconda

2.4 Altri tools

Capitolo 3

Il Reinforcement learning

Tra tutti i tipi di apprendimento esistenti ce n'è uno chiamato **Reinforcement Learning**, che è stato scelto per lo sviluppo di questo progetto. Il reinforcement learning è un tipo di addestramento che non necessita di esempi (come ad esempio il Supervisioned Learning), ma si basa unicamente sull'esperienza accumulata eseguendo delle azioni; questo perchè il reinforcement learning nasce proprio per affrontare situazioni nelle quali non è possibile definire cosa sia giusto o sbagliato tramite degli esempi.

Per capire meglio questo concetto, si pensi ad una partita a scacchi: il numero di possibili configurazioni dei pezzi sulla schacchiera è praticamente impossibile da enumerare, e ciò vuol dire che per quanti esempi possiamo dare, non riusciremo mai a definire cosa è (sempre) giusto oppure sbagliato.

Un esempio lampante è dato dalla classica mossa che consiste nel sacrificare un pezzo; solitamente perdere un pezzo non è una buona cosa, tuttavia, se tale perdita permette di conquistare il re avversario, allora può essere considerata una mossa eccellente.

3.1 Il sistema di ricompensa

Ciò che si vuole fare per riuscire ad addestrare comunque un AI per questo tipo di task, è di basare l'apprendimento sull'esperienza accumulata. In parole più semplici, si fa eseguire il task all'agente, e si da una ricompensa positiva se esso è stato eseguito bene, od una negativa se esso è stato eseguito male.

L'agente a questo punto capirà che deve eseguire il task in modo diverso (se la ricompensa è stata negativa), oppure che dovrà rieseguire il task come ha già fatto (se la ricompensa è stata positiva).

Il modello che abbiamo appena descritto in modo molto grossolano (azione-ricompensa) in realtà non è altro che una semplificazione di un modello già esistente in natura noto come **Sistema di ricompensa**. Il sistema di ricompensa è un gruppo di strutture neurali presenti negli animali (anche

nell'uomo), che ha come compito quello di associare un emozione (come ad esempio il piacere o il dolore) alle azioni che vengono compiute.

Per capire praticamente ciò che intendiamo, supponiamo di voler mangiare un panino con la mortadella dopo aver lavorato 6 ore senza alcun pasto: quando avremo dato l'ultimo morso al panino, avremo finalmente saziato il nostro senso di fame (avere fame provoca fastidi e dolori di stomaco, quindi è un emozione negativa), che lascierà spazio ad una piacevole senzazione di sazietà (avere lo stomaco pieno è una senzazione gradevole, quindi positiva). Supponiamo ora di mangiare lo stesso panino dopo un pasto abbondante, come ad esempio il cenone di capodanno: in questo caso a fine pasto il nostro senso di sazietà sarà già arrivato a buon punto (emozione positiva), tuttavia mangiare quel panino con la mortadella provocherà un senso di pienezza e gonfiore, dovuto all'aver mangiato eccessivamente (e quindi si passerà ad avere un emozione negativa).

Con questo esempio si è visto in modo molto semplice come il sistema della ricompensa riesca a valutare, in un modo diverso, una stessa azione a seconda di una serie di fattori esterni: questo è proprio il comportamento che vorremmo poter replicare.

Il sistema della ricompensa in realtà è estremamente più complesso e tiene conto di tantissimi fattori, tuttavia ciò che si prenderà in considerazione sarà una versione molto semplificata nella quale si possono identificare i seguenti elementi costitutivi:

- Un **agente**, che è colui che prende le decisioni (il giocatore di scacchi, o, nel nostro caso, il conducente dell'automobile).
- Un set di **azioni**, che può effettuare l'agente, come ad esempio sterzare il volante, od accellerare. Indicheremo tale set con il simbolo $\mathcal{A}$.
- Un set di **stati**, ognuno dei quali descrive lo stato dell'agente e dell'ambiente che lo circonda, come ad esempio il valore rilevato dai sensori dell'automobile. Indicheremo tale set con il simbolo $\mathcal{S}$.

In questo modo potremo identificare come **agente** colui che effettua l'**azione** di mangiare il panino, passando da uno **stato** di fame ad uno di sazietà (gli stati saranno codificati tramite variabili). Ancora, potremmo identificare come agente il pilota dell'automobile, che effettua l'azione di sterzare per passare da uno stato di pericolo imminente (collisione con il muro) ad uno di normalità (strada libera).

3.2 Formalizzazione del modello

Anche avendo abozzato un idea di quali sono gli elementi principali che entrano in gioco, è abbastanza evidente che ci mancano molti elementi per

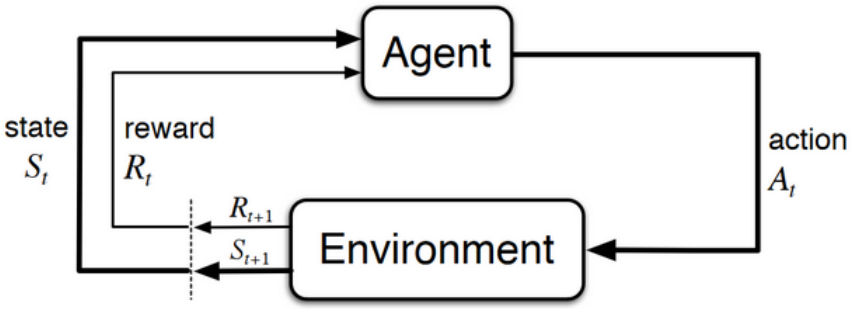


Figura 3.1: Esecuzione iterativa del task.

formalizzare un modello matematico adatto a sostenere le nostre necessità. Prima di cominciare ad elencare ciascuno di questi aspetti, è importante sottolineare che gli insiemi $\mathcal{A}$ e $\mathcal{S}$ sono finiti, e dunque abbiamo un numero limitato di possibili stati, ed azioni che possiamo effettuare.

A questo punto cominciamo ad introdurre qualche dettaglio in più, ed esponiamo quali sono gli elementi necessari ad un modello più completo.

3.2.1 Le reward (ricompense)

Quando si effettua una certa azione, $a \in A$, si è interessati a capire qual è il beneficio che si trae nell'aver effettuato tale azione, positivo o negativo che sia. Come già spiegato prima, tuttavia, non è possibile definire in modo assoluto la bontà di un azione senza legarla al contesto (stato) in cui essa viene effettuata.

L'idea allora è quella di introdurre un nuovo insieme, detto insieme delle **reward**, che contiene appunto delle ricompense. Questo insieme sarà indicato con il simbolo $\mathcal{R}$, e d'ora in poi si darà per scontato che anche tale insieme sia finito.

L'importantissimo ruolo di associare ad ogni coppia (s, a) (con $s \in \mathcal{S}$, $a \in \mathcal{A}$) una ricompensa può essere esplicitato tramite una funzione arbitraria (decisa a seconda dell'ambito applicativo) del tipo

$$f : \mathcal{S} \times \mathcal{A} \to \mathcal{R} \tag{3.1}$$

3.2.2 Definizione di trajectory

Una volta definite le nozioni di stato, azione e reward, si è pronti per analizzare il funzionamento generale del modello che si sta costruendo. La spiegazione di tale funzionamento diventa molto intuitiva se si immagina l'esecuzione del task come un processo iterativo, così come mostrato in Figura 3.1.

Il fatto stesso di dover analizzare il processo iterativo come un susseguirsi di rilevazioni di stato, azioni e ricompense, induce la necessità di utilizzare un fattore temporale al fine di distinguere gli step durante i quali si verifica ciascun avvenimento. Si introducono allora degli step temporali, che vengono codificati semplicemente tramite dei numeri naturali, e si introduce una nuova notazione anche per gli stati, azioni e ricompense in cui si aggiunge l'informazione temporale del loro avvenimento. In questo modo lo stato in cui si trova l'agente al tempo t verrà indicato con s_t, l'azione che si effettua verrà indicata con a_t e la ricompensa ottenuta la indicheremo con r_{t+1} ($t+1$ perchè la ricompensa viene assegnata in un momento successivo all'azione).

Prendendo in riferimento la Figura 3.1, si avrà dunque che al tempo t l'agente rileva lo stato s_t; in base a questa rilevazione esso sceglie un azione da effettuare, che sarà indicata con la scrittura a_t; l'azione effettuata provocherà un cambiamento nell'ambiente, che assumerà un nuovo stato s_{t+1}, e contemporaneamente, verrà assegnata la ricompensa r_{t+1} all'agente per aver effettuato l'azione a_t trovandosi nello stato s_t. A questo punto il ciclo ricomincia dall'inizio e segue le stesse dinamiche.

Il susseguirsi di questo ciclo viene indicato in modo molto coinciso tramite la **trajectory**, ovvero la sequenza che si viene a creare, e che viene indicata nel seguente modo:

$$S_0, A_0, R_1, S_1, A_1, R_2, ... \tag{3.2}$$

Ciò che finora non è mai stato menzionato è che sarà il vero scopo che perseguirà l'agente durante tutta la sua esistenza è la massimizzazione delle reward ottenute. L'agente d'ora in avanti dovrà cercare di scegliere, in ogni istante t, un'azione in modo tale da massimizzare le reward che otterrà in futuro; si noti che non a caso si sono menzionate **tutte** le reward future, e non solo quella immediatamente successiva: per questo motivo, d'ora in avanti diremo più propriamente che l'agente ha come scopo quello di massimizzare le **reward cumulate**.

3.2.3 Probabilità delle transizioni

Per massimizzare le reward ottenute dall'agente vi è la necessità di calcolare il valore che ci si aspetta di ottenere dalle reward future (come vedremo tra poco, ciò verrà fatto tramite il ritorno atteso); per fare questo calcolo vi è la necessità di conoscere qual è la probabilità che l'agente, scegliendo una determinata azione in un determinato stato, riceva una certa reward.

Ora, dato che le nostre premesse prevedevano che $\mathcal{S}$, $\mathcal{A}$ ed $\mathcal{R}$ erano insiemi finiti, allora ciascuna variabile aleatoria S_t ed R_t avrà una distribuzione di probabilità ben definita. Si noti, inoltre, che per quanto detto finora lo stato S_t e la reward R_t dipendono unicamente dallo stato precedente, ovvero S_{t-1} e dall'azione effettuata nel tempo precedente, ovvero A_{t-1}.

Ciò che è necessario definire al fine del calcolo del ritorno atteso delle reward future è di definire per ogni $s' \in \mathcal{S}$, $s \in \mathcal{S}$, $r \in \mathcal{R}$ e $a \in \mathcal{A}(s)$ (dove con $\mathcal{A}(s)$ indichiamo l'insieme di tutte le possibili azioni di $\mathcal{A}$ che possiamo intraprendere se ci troviamo nello stato s), la probabilità così definita

$$\mathbb{P}(s', r|s, a) = \mathbb{P}(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{3.3}$$

3.2.4 Le policy

Un altro aspetto fondamentale nella formalizzazione del modello riguarda la probabilità che l'agente, trovandosi in un certo stato, compia una determinata azione. Finora si è solamente detto che l'agente ha la possibilità di scegliere una possibile azione che rientri nell'insieme $\mathcal{A}(s)$, ovvero tra tutte le azioni disponibili in un determinato stato; ora si vuole invece formalizzare come avviene questa scelta, e lo si fa introducendo la nozione di policy.

Una policy non è altro che una funzione che associa a ciascuno stato $s \in \mathcal{S}$ una distribuzione di probabilità su $\mathcal{A}(s)$ (si ricordi sempre che tutti gli insiemi che utilizziamo sono finiti). In modo più formale si dice che l'agente **segue** una politica, la quale verrà indicata con il simbolo π.

Se l'agente segue la politica π al tempo t allora con $\pi(a|s)$ si indicherà la probabilità che l'agente scelga l'azione a se questo si trova nello stato s (in altre parole la probabilità che $A_t = a$ se $S_t = s$).

3.2.5 Ritorno atteso e fattore di sconto

Come detto finora l'agente ha come scopo quello di massimizzare le reward cumulate, per questo sarà utile introdurre un nuovo concetto che è quello di **ritorno atteso**. Il ritorno atteso non è altro che una funzione basata sulle reward future, che viene definita in base al tipo di funzionamento desiderato. Ad esempio potremmo definire il ritorno atteso (calcolato al tempo t) come

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... \tag{3.4}$$

Ovvero come la somma di tutte le reward future. Un calcolo di questo tipo, avrebbe tuttavia un problema legato al numero di reward future: se esse sono infinite allora non è possibile definire un risultato preciso di tale sommatoria. Ciò che si fa è allora di introdurre un fattore, che indichiamo con γ e che chiamiamo **fattore di sconto**. Questo fattore ha un ruolo importantissimo che è quello di limitare il contributo delle reward future rispetto a quelle prossime (in altre parole si presta più importanza alle reward immediatamente successive, mentre si presta meno attenzione a quelle in un futuro lontano).

Il fattore γ viene definito in un range che varia nell'intervallo $[0, 1]$, in modo tale che se γ si avvicina a 0 allora non si presta alcuna attenzione verso

le reward in futuro, mentre se si avvicina ad 1 allora si presta la stessa attenzione a tutte le reward, che siano prossime o future. Un modo di utilizzare il fattore di sconto potrebbe ad esempio essere il seguente:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... = \sum_{i=0}^{\infty} \gamma^k R_{t+1+k} \tag{3.5}$$

3.2.6 Value function

Le value function sono delle funzioni che ci permettono di stimare quanto sia positivo il fatto che un agente si trovi in un determinato stato, oppure, quanto sia positivo che esso effettui una certa azione quando si trova in un certo stato. Il concetto di positività, appena citato, viene espresso tramite il ritorno atteso; ora, dal momento che il ritorno atteso dipende dall'azione che prende l'agente, trovandosi in un determinato stato, dipende unicamente dalla policy adottata, si può dire senza problemi che la value function può essere espressa in termini della policy.

State-value function

La funzione state-value, per una certa policy π, indica quanto è buono, per un agente che segue la policy π, trovarsi in un certo stato. Questa funzione viene solitamente indicata con v_π, ed il valore per ogni stato s, viene definito come il ritorno atteso partendo dallo stato stesso s, al tempo t, seguendo poi la politica π

$$v_\pi = E_\pi[G_t|S_t = s] = E_\pi[\sum_{i=0}^{\infty} \gamma^k R_{t+1+k}|S_t = s] \tag{3.6}$$

Action-value function

La funzione action-value, per una certa policy π, indica quanto è buono, per l'agente, eseguire una certa azione se esso si trova in un certo stato. Dal momento che l'azione effettuata dipende unicamente dalla policy seguita, indicheremo la funzione action-value con il simbolo q_π, e definiremo $q_\pi(s, a)$ come

$$q_\pi = E[G_t|S_t = s, A_t = a] = E_\pi[\sum_{i=0}^{\infty} \gamma^k R_{t+1+k}|S_t = s, A_t = a] \tag{3.7}$$

La funzione q_π viene anche chiamata Q-function, mentre il valore che assume tale funzione per una certa coppia (s, a) viene detto Q-value.

3.2.7 Policy ottimali

Una volta introdotti tutti questi concetti, si può passare a spiegare quale sia in effetti lo scopo di un qualsiasi algoritmo di reinforcement learning. Tramite le state-value function e le action-value function si è riusciti a definire un modo di calcolare il ritorno atteso, partendo da un certo stato, e seguendo una certa policy; non è difficile capire, a questo punto, che ciò che fa veramente la differenza nella massimizzazione del ritorno atteso, sono prorpio le policy seguite.

L'obiettivo è dunque quello di trovare una policy, le cui scelte, permettano di ottenere un ritorno atteso più alto di qualsiasi altra policy. In modo più formale, si dirà che una policy π è migliore o uguale alla policy π' se il ritorno atteso della policy π è maggiore od uguale al ritorno atteso della policy π' per ogni stato s. In modo più formale

$$\pi = \pi' \iff \forall s \in S, v_\pi(s) \geq v_{\pi'}(s) \tag{3.8}$$

Una policy che sia migliore di qualsiasi altra policy, viene chiamata **policy ottimale**. Una policy ottimale, avrà associate delle proprie state-value function e action-value function, che verranno indicate rispettivamente con v_* e q_*. Queste due funzioni vengono formalmente definite in questo modo

$$v_*(s) = \max_\pi v_\pi(s) \tag{3.9}$$

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{3.10}$$

3.2.8 Equazione di Bellman

L'equazione di Bellman è un equazione ottenuta dalla funzione action-value ottima, e si può dimostrare che vale la seguente:

$$q_*(s, a) = E[R_{t+1} + \gamma \max_{a'} q_*(s', a')] \tag{3.11}$$

Quest'equazione dice che per ogni coppia (s, a), al tempo t, il ritorno atteso ottenuto partendo dallo stato s, eseguendo l'azione a, e poi seguendo la policy ottima, è uguale al valore atteso della somma della reward ottenuta effettuando l'azione a nello stato s (che sarebbe R_{t+1}) e del massimo ritorno atteso (scontato da γ) che può essere raggiunto tra tutte le possibili azioni a' che si possono effettuare nel prossimo stato s'.

Si noti che se conoscessimo quale sia la funzione q_* allora potremmo risalire alla policy ottima, in quanto, ad ogni passo t, sapremmo calcolare il massimo valore di $q_*(s', a')$ per ciascuna azione a': calcolare il valore di questa funzione ed approssimarla al meglio è dovere degli algoritmi di reinforcement learning.

Capitolo 4

Q-Learning

Il Q-Learning è uno tra i più noti algoritmi di reinforcement learning, nonchè la prima tecnica usata per risolvere il problema su cui si concentra questo progetto. Come detto nel capitolo precedente, l'obiettivo di un algoritmo di reinforcement learning è quello di trovare la funzione ottima q_*, dunque, mostriamo come ciò avviene nel Q-Learning.

4.1 Iterazione per valore

Piuttosto che trovare la funzione q_*, il Q-Leaning cerca di apprendere, per ogni coppia (s, a), il Q-value ottimo. Per riuscire in questo scopo, l'idea è quella di effettuare continuamente delle iterazioni, durante ognuna delle quali si aggiornano tutti i Q-value appresi fino a quel momento, e lo si fa utilizzando l'equazione di Bellman: il tutto si ripete finchè i valori appresi non convergono verso quelli ottimali, che sono quelli della funzione q_*. Questo approccio viene chiamato iterazione per valore.

Per capire meglio ciò che intendiamo, proviamo ad applicare questo concetto al nostro problema; avremo dunque l'agente, che sta guidando un auto con dei sensori, e che può accelerare e sterzare. L'obiettivo del nostro agente è quello di riuscire a guidare l'auto senza che essa vada a sbattere contro il muro, per questo faremo in modo che:

- Quando la macchina va a sbattere contro un muro, l'agente prende una reward di -20.
- Quando l'agente non va a sbattere contro il muro (e riesce dunque a rimanere in pista) prende una reward di 0.05.

All'inizio dell'algoritmo, ovviamente, l'agente non sa quali sono le azioni che gli permettono di avere una reward positiva (o meglio, di evitare quella negativa), e questo viene rispecchiato pienamente impostando a 0 il Q-value

di ciascuna coppia (s, a). Ora, al fine di memorizzare in modo più intuitivo i Q-value che andremo a calcolare, utilizzeremo una tabella, chiamata **Q-table**, in cui le righe rappresentano gli stati e le colonne le azioni: ciascuna cella di questa tabella rappresenterà il Q-value associato alla rispettiva coppia.

**********INSERIRE IMMAGINE TABELLA**********

4.2 Exploration ed Exploitation

Ora, come abbiamo appena detto, la tabella contiene inizialmente tutti valori nulli; durante l'esecuzione, tuttavia ciascuna cella si assesterà su un certo valore, ed una volta riusciti in questo, potremo utilizzare questi valori per scegliere ad ogni passo l'azione che massimizza la reward ottenuta. Rimane comunque il problema di decidere quali siano, all'inizio dell'algoritmo, le azioni che devono essere eseguite; per rispondere efficacemente a questa domanda vi è la necessita di introdurre i concetti di **exploration** e di **exploitation**.

Exploration L'exploration consiste nell'esplorare l'ambiente che circonda l'agente, al fine di raccogliere informazioni su quest'ultimo, e permettere all'agente di aggiornare la Q-table in modo opportuno.

Exploitation L'exploitation consiste nello sfruttare le conoscenze che l'agente ha acquisito fino a quel momento per massimizzare le reward ottenute.

Per riuscire ad ottenere una Q-table abbastanza efficace, l'idea è quella di imporre all'agente di alternare exploration ed exploitation, in modo da visitare l'ambiente, e capire quale sia l'azione che permette di ottenere una reward massima. Il modo di alternare exploration ed exploitation, avviene in modo piuttosto semplice: modellando delle probabilità. All'inizio dell'algoritmo le probabilità di scegliere un exploration sono più alte rispetto a quelle di scegliere un exploitation; all'avanzare degli step temporali, diventa sempre meno probabile l'exploration, ma più probabile l'exploitation.

4.3 Aggiornamento della Q-table

Spiegato il funzionamento generale del Q-Learning non ci rimane altro che spiegare il modo in cui viene aggiornata la Q-table ad ogni iterazione. Per spiegare tale funzionamento sarà necessario introdurre il concetto di **learning rate**. Il learning rate è un numero compreso tra 0 ed 1 che servirà ad indicare che pesò avrà il valore già presente nella Q-table rispetto ad un valore appena calcolato (detto in altre parole, quanto velocemente viene alterato il valore della Q-table), e d'ora in poi sarà indicato con α.

E' chiaro che per valori bassi di α avremo che i valori appena calcolati influiranno poco sull'aggiornamento, al contrario, valori alti di alpha faranno in modo di dare molta rilevanza ai valori appena calcolati. Allora l'equazione per aggiornare la nostra Q-table può essere ottenuta in modo molto semplice:

$$q^{new}(s,a) = (1-\alpha)q(s,a) + \alpha(R_{t+1} + \gamma \max_{a'} q_*(s',a')) \tag{4.1}$$

Con questo si conclude la parte teorica riguardo il Q-learning e si passa alla parte pratica, nella quale si mostrerà l'applicazione pratica del modello al problema, e i risultati che sono stati ottenuti.

4.4 Episodi

Come già detto in precedenza i Q-values vengono appresi iterando continuamente per un certo numero di step. Il numero di step per la quale continuare il task, dipende strettamente dal task: potrebbe variare da poche unità fino a diverse migliaia.

Esisono tuttavia dei task in cui vi è una naturale suddivisione tra i diversi step, come ad esempio in una partita a ping pong; quando si gioca una partita, infatti, vengono effettuati numerosi scambi di pallina, ma vi è un momento in cui si "resetta" l'ambiente e si ricomincia da zero: quando uno dei due giocatori guadagna un punto.

Si distingueranno allora i vari cicli che definiscono l'aver guadagnato un punto, e chiameremo tali cicli **episodi**. Ogni task potrà avere un numero di episodi variabile, che verrà definito specificamente all'occorrenza.

4.5 Applicazione del modello al problema

In questa sezione vedremo com'è stato impostato il modello del reinforcement learning al nostro problema, e come esso è stato risolto tramite l'algoritmo del Q-learning.

Per quanto riguarda gli stati, essi sono definiti in base alla discretizzazione dei valori assunti dai sensori dell'automobile. Il sensore dell'automobile, ricordiamo, riesce a rilevare la distanza dall'ostacolo tramite un numero reale a patto che questo si trovi in un range di valore che il sensore è ingrado di rilevare; per questo motivo si è deciso di discretizzare il valore di ciascun sensore in 4 possibile fasce:

- Fascia rossa, quando il sensore rileva una distanza minore del 25% della sua portata massima.
- Fascia arancione, quando il sensore rileva una distanza minore del 50% della sua portata massima.

- Fascia gialla, quando il sensore rileva una distanza minore del 75% della sua portata massima.
- Fascia verde, quando il sensore rileva una distanza superiore al 75% della sua portata massima.

Le fasce sono state espresse in termini di percentuale perchè ogni sensore ha una portata massima diversa. Dal momento che sono presenti 5 sensori, avremo un totale di 20 possibili stati, determinati dalle varie combinazioni.

Per le azioni invece si è scelto di optare per 3 possibili azioni: accelerazione con direzione in avanti, accelerazione con direzione a sinistra ed accelerazione con direzione a destra.

Le reward invece sono solamente due: -20 e 0.05. La prima viene assegnata quando l'automobile si scontra con il muro, mentre la seconda quando questo non avviene.

I parametri principali impostati sono quelli che riguardano il learning rate ed il fattore di sconto, che abbiamo posto rispettivamente a $\alpha = 0.1$ e $\gamma = 0.99$.

Per quanto riguarda il numero di episodi adottati, ci si limita a 500 episodi, ognuno dei quali termina quando la macchina non riesce a completare il suo task, ovvero quando si scontra con il muro.

4.6 Risultati

I risultati ottenuti sono stati piuttosto soddisfacenti: il tutto è andato secondo le aspettative. Il grafico mostrato in figura 4.1 indica come si sono modificate le reward cumulate all'avanzare degli episodi.

Come si evince dal grafico, nelle prime epoche le reward sono piuttosto basse e seguono un andamento irregolare: questo rispecchia perfettamente il fatto che l'agente sta facendo continue operazioni di esploration (azioni casuali per esplorare l'ambiente). Quando l'agente comincia a fare operazioni di exploitation, esso cerca continuamente di massimizzare le sue reward, e come possiamo vedere ci riesce egregiamente. Ovviamente vi sono episodi in cui le reward sono basse e altri in cui le reward sono alte (questo anche dipendentemente dal fatto che esiste ancora una lieve probabilità che venga effettuata un esplorazione), ma la cosa che conta maggiormente è che mediamente le reward si sono alzate di molto.

La linea rossa indicata sul grafico rappresenta una regressione cubica dei dati, per mostrare meglio l'andamento dell'apprendimento.

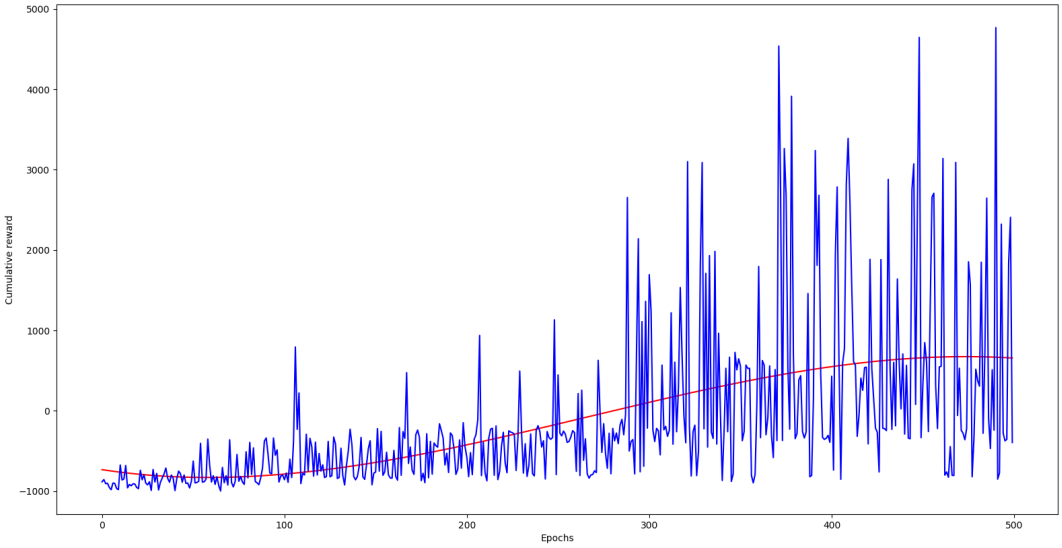


Figura 4.1: Grafico delle reward cumulate rispetto agli epidosi.

Capitolo 5

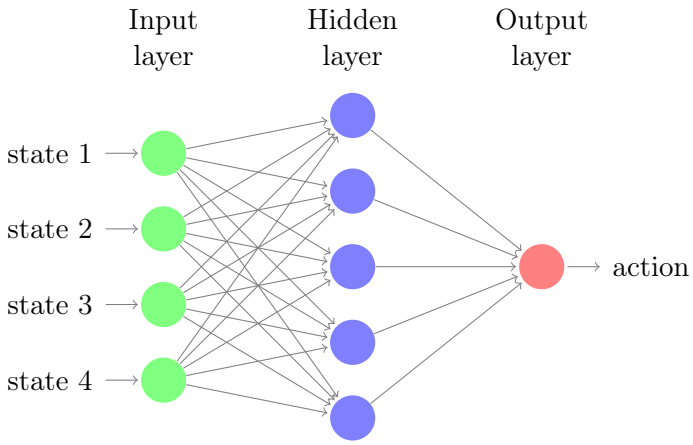
Deep Q-Learning

Il Q-Learning, introdotto nel capitolo precedente, soffre di alcune limitazioni che riguardano il numero di stati ed azioni che si possono gestire: se il numero di questi risulta essere elevato le performance calano drasticamente. Ciò è dovuto al fatto che per ottenere un apprendimento efficace (e quindi per trovare una Q-function ottima), sia gli step che gli episodi, dovrebbero diventare incredibilmente alti, e ciò è computazionalmente ingestibile.

Per risolvere i task in cui sia il numero di stati che di azioni sono molto alti viene allora introdotta una nuova tecnica che è quella del Deep Q-Learning. In questa tecnica ciò che si fa è di ricorrere all'utilizzo di reti deep con lo scopo di approssimare la Q-Function; vediamo allora come viene organizzata la rete deep.

5.1 La rete deep

L'idea è quella di avere come input della rete tutti gli stati e come output le azioni da prendere; il vantaggio di avere questa strutturazione è evidente: ogni stato può assumere un valore nel continuo (e dunque possiamo avere una combinazione altissima di stati), così come l'output stesso assume valori continui e dunque possiamo avere un insieme di possibili azioni altrettanto alto.



5.1.1 Apprendimento

Per permettere l'apprendimento di una rete neurale è necessario introdurre una funzione di errore, su cui si baserà la back propagation per la correzione dei parametri. Nel caso in esame, si è utilizzata una tecnica standard per l'apprendimento delle reti neurali che è la **PPO** (Proximal Policy Optimization).

Per spiegare in cosa consiste la PPO è opportuno proseguire in modo graduale, introducendo prima la nozione di policy gradient.

Policy gradient

In generale nei metodi basati sulle policy si cerca di apprendere direttamente la policy (che mappa gli stati in azioni), piuttosto che apprendere una value function che ci fornisce il ritorno atteso delle reward cumulate dato uno stato ed un'azione.

Utilizzeremo i parametri della value function per ottimizzare i parametri della policy che vogliamo apprendere. Avendo allora la funzione di policy π, avremo che essa viene espressa tramite i parametri assunti dalla rete che la rappresenta, che indicheremo con π_θ.

Una volta che partiamo da una certa configurazione della rete, come facciamo a sapere se i parametri θ della policy sono effettivamente buoni? Ricordiamo che la ricerca della policy migliore può essere vista come un problema di ottimizzazione in cui si vuole massimizzare la funzione obiettivo G (ritorno atteso dal tempo zero).

A questo punto bisognerà seguire due semplici step:

1. Misurare la bontà della policy π tramite una certa funzione obiettivo (nel nostro caso G).

2. Usare la policy gradient ascendente per migliorare i parametri θ.

Il motivo per cui si utilizza la policy gradient ascendente piuttosto che la discendente è molto semplice: nel nostro caso vogliamo massimizzare una reward, e non minimizzare un errore. Il gradiente sarà rappresentato da $\nabla_\theta G(\theta)$, e l'aggiornamento di θ sarà dunque $\theta \leftarrow \theta + \alpha \nabla_\theta G(\theta)$.

Andiamo ora a vedere qual è il gradiente che bisogna sommare ad ogni episodio.

Sfruttando la definizione di valore atteso, possiamo considerare $G(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} R_{t+1}|\pi_\theta] = \sum_{t=0}^{T-1} \mathbb{P}(s_t, a_t|\tau) R_{t+1}$ dove i è un punto arbitrario di inizio traiettoria e τ è la traiettoria.

Differenziando entrambi i termini rispetto a θ si ottiene che

$$\begin{aligned}
\nabla_\theta G(\theta) &= \sum_{t=i}^{T-1} \nabla_\theta \mathbb{P}(s_t, a_t|\tau) R_{t+1} \\
&= \sum_{t=i}^{T-1} \mathbb{P}(s_t, a_t|\tau) \frac{\nabla_\theta \mathbb{P}(s_t, a_t|\tau)}{\mathbb{P}(s_t, a_t|\tau)} R_{t+1} \\
&= \sum_{t=i}^{T-1} \mathbb{P}(s_t, a_t|\tau) \nabla_\theta \log \mathbb{P}(s_t, a_t|\tau) R_{t+1} \\
&= \mathbb{E}[\sum_{t=i}^{T-1} \nabla_\theta \log \mathbb{P}(s_t, a_t|\tau) R_{t+1}] \qquad (5.1)
\end{aligned}$$

Ora, dal momento che le scelte che prendiamo possono essere casuali non si considera il valore atteso ma un approssimazione:

$$\nabla_\theta G(\theta) \sim \sum_{t=i}^{T-1} \nabla_\theta \log \mathbb{P}(s_t, a_t|\tau) R_{t+1} \qquad (5.2)$$

Si vede inoltre che applicando le proprietà dei logaritmi ed effettuando alcuni passaggi algebrici, si ottiene

$$\nabla_\theta \log \mathbb{P}(s_t, a_t|\tau) = \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \qquad (5.3)$$

Si ottiene quindi:

$$\nabla_\theta G(\theta) = \sum_{t=0}^{T-1} R_{t+1} \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) (\sum_{t'=t+1}^{T} R_{t'}) \qquad (5.4)$$

Considerando l'introduzione di γ si ha $\sum_{t'=t+1}^{T} \gamma^{t'-t-1} R_{t'} = G_t$, avendo quindi che

$$\nabla_\theta G(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) G_t \qquad (5.5)$$

PPO

Possiamo ora introdurre la PPO. La PPO non è altro che una policy gradient che funziona in modo tale da evitare di fare aggiornamenti dei parametri θ che abbiano intervalli troppo larghi o troppo piccoli (in pratica modifica l'azione di poco). A questo scopo introduciamo delle nuove funzioni che chiameremo **Clipped Surrogate Objective Function**.

Queste funzioni, piuttosto che utilizzare il logaritmo della policy, per tener traccia dell'impatto delle azioni, utilizza il rapporto tra la probabilità dell'azione sotto la policy attuale e la probabilità dell'azione sotto la policy precedente.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{5.6}$$

Ora, se la ratio $r_t(\theta) > 1$ allora l'azione ha una maggiore probabilità di essere contenuta nella vecchia policy, viceversa se $0 < r_t(\theta) < 1$.

Dunque la funzione obiettivo diventa:

$$L^{CPI}(\theta) = \mathbb{E}_t[r_t(\theta)G_t] \tag{5.7}$$

Tuttavia, si osserva che se un azione ha maggiore probabilità di essere contenuta nella policy attuale, si otterrebbero degli step molto grandi e quindi un aggiornamento errato. Dobbiamo quindi aggiungere un vincolo alla funzione obiettivo, così da portare la ratio ad assumere valori in un intorno di 1 non eccessivamente grande.

Questo può essere realizzato tramite Clipped surrogate objective function, precisamente:

$$L^{CLIP}(\theta) = \mathbb{E}_t[min(r_t(\theta)G_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)G_t)] \tag{5.8}$$

5.2 Experience replay

Dal momento che l'idea di fondo per fare learning prevede di utilizzare le immagini come input vi sono delle considerazioni importanti da fare. Supponendo di porre in ogni singolo input un singolo frame nasce un problema rilevante: non si è in grado di contestualizzare lo stato (ad esempio non si può distinguere la direzione di un movimento).

Si introduce quindi una tecnica nota come **Experience replay** in cui ad ogni step t si memorizza l'esperienza dell'agente, $e_t = (s_t, a_t, R_t, s_{t+1})$, in un dataset $\mathcal{D} = e_1, ..., e_\mathcal{N}$ che chiameremo **Replay memory**.

5.3 Algoritmo: Deep Q-Learning con experience replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figura 5.1: Algoritmo.

5.4 Versione senza cnn

5.5 Risultati ottenuti