UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Magistrale in Informatica

Approfondimento per il corso di Machine Learning

Offline Reinforcement Learning

Luca Dorigo
130185

Diego Borsoi
129804

18 febbraio 2021

Sommario

In questa relazione andremo a presentare il problema dell'offline reinforcement learning. Dopo una descrizione del problema, andremo ad esporre quali sono le principali tecniche che vengono usate per risolverlo e vedremo quali tool le implementano. Infine, presenteremo la nostra implementazione dell'algoritmo CQL e i risultati ottenuti.

Indice

1 Problema

1.1 Reinforcement Learning

Supponiamo di avere un agente immerso in un ambiente. L'agente potrà interagire con l'ambiente osservandolo, eseguendo delle azioni e ottenendo una ricompensa (reward) per le azioni compiute. L'obiettivo del Reinforcement Learning è di imparare una sequenza di azioni che massimizza le ricompense date dall'ambiente.

Nel resto della sezione vedremo in modo più preciso questi concetti

Stati, azioni e traiettorie

Uno stato s_t è una descrizione completa dell'ambiente in cui si trova il nostro agente. Nel caso l'ambiente non sia completamente conoscibile all'agente, chiamiamo osservazione la descrizione parziale dello stato disponibile all'agente. Un'azione a_t è una risposta del nostro agente allo stato, o alla sua osservazione.

Possiamo quindi descrivere il comportamento dell'agente come un vettore $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ che ciamiamo traiettoria.

Ricompense

Un elemento immportante per il RL è il concetto di ricompensa. Ad ogni azione dell'agente corrisponde una risposta, positiva o negativa, dell'ambiente, che può essere immediata o ritardata nel tempo.

Data la reward hypotesis, che ci dice che ogni obiettivo può esere espresso dalla massimizzazione del valore atteso della ricompensa totale, l'obiettivo di RL è massimizzare

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$$

dove γ è un fattore di attenuazione.

Markov Decision Process (MDP)

L'intero sistema può essere rappresentato come un Markov decison process $\mathcal{M}$ definito da

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, d_0, r, \gamma)$$

dove

- $\mathcal{S}$ è un insieme di stati (continuo o discreto);
- $\mathcal{A}$ è un insieme di azioni (anche questo, continuo o discreto);
- T definisce una probabilità condizionata $T(s_{t+1}|s_t, a_t)$ che descrive il risultato delle azioni sul sistema;
- d_0 definisce la distribuzione iniziale degli stati;
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ è la funzione reward;
- γ è il fattore di attenuazione $\in [0, 1]$.

Partyally Observed MDP

Nel caso in cui lo stato non sia fully observable, definiamo un partially observed MDP $\mathcal{M}$ come

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, d_0, E, r, \gamma)$$

dove

- gli elementi omonimi sono definiti come sopra;
- $\mathcal{O}$ è un insieme di osservazioni;
- E è la funzione di emissione, che definisce la distribuzione $E(o_t|s_t)$.

Policy

Una policy π è una mappa tra stati o osservazioni a azioni. Può essere deterministica ($\pi(s) = a$) oppure stocastica ($\pi(a|s) = P[a_t = a, s_t = s]$).

Inoltre, possiamo definire policies parametriche π_θ, cioè dipendenti da dei parametri che possono essrere ottimizzati.

Funzioni valore

Può essere utile definire delle funzioni che ci restituiscano il valore di uno stato:

- la funzione state–value $V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s]$ restituisce il valore atteso della traiettoria che parte da s e segue la policy π.

- la funzione action–value $Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$ restituisce il valore atteso della traiettoria che parte da s_0, esegue a e poi segue la policy π.

Nel caso parametrico, le funzioni diventano $V^\pi_\theta(s)$ e $Q^\pi_\theta(s,a)$.

1.2 Offline Reinforcement Learning

Il caso dell'offline RL si può vedere come come una formulazione data–driven del RL classico. Infatti, non abbiamo più l'interazione con l'ambiente, ma ci viene fornito un dataset $\mathcal{D} = \left\{(s^i_t, a^i_t, s^i_{t+1}, r^i_t)\right\}$ da cui dobbiamo essere in grado di apprendere una policy che ci permetta di interagire con successo con l'ambiente reale. L'insieme $\mathcal{D}$ può essere visto come il training set degli algoritmi classici di Machine Learning supervisionato.

Vantaggi e svantaggi

In alcuni casi, interagire con l'ambiente potrebbe non essere un'opzione, per via di costi o rischi proibitivi (si pensi al caso medico, in cui l'agente prende decisioni che influiscono sulla salute dei pazienti). Allora, essere in grado di apprendere una policy offline basandosi magari su delle azioni eseguite da umani (es. il medico) potrebbe essere una strada molto interessante. Inoltre, essere capaci di apprendere da esperienze passate può ridurre il numero di interazioni con l'ambiente necessario ad ottenere buoni risultati, applicando le conoscenze già apprese per altri obiettivi (cioè prendendo in $\mathcal{D}$ tutti i sample usati per il training di altre policies).

Tuttavia, spostarsi fuori dall'ambiente crea diversi problemi. Il più ovvio è che il dataset deve coprire bene lo spazio degli stati; infatti, non essendo in grado di esplorare nuove regioni, se non ci sono dati riguardo a zone con reward molto alta non saremo mai in grado di scoprirle. Un altro problema è che, per essere in grado di fare meglio di quanto si vede nel dataset, dobbiamo comportarci in modo un po' diverso. Per fare questo, dovremmo essere capaci di sapere come si comporterebbe il sistema nei casi che non sono nel dataset. Se tuttavia assumiamo che i dati siano indipendenti e identicamente distribuiti (iid) ci esponiamo al distributional shift: la nostra policy funzionerà bene su dati distribuiti come quelli del dataset, mentre non sarà altrettanto se la distribuzione è diversa.

2 Panoramica

In questa sezione, presenteremo brevemente alcune tecniche per affrontare il problema dell'offline RL, presentando anche alcuni tool che le implementano.

2.1 Metodi basati su Importance Sampling

In questa sezione, parleremo dei metodi che cercano di stimare direttamente la policy. Poichè conosciamo solo la distribuzione degli stati nel dataset $\mathcal{D}$, usiamo diverse forme di importance sampling per stimare $R(\tau)$ con traiettorie prese da $\mathcal{D}$ (off–policy evaluation), oppure cercheremo di stimare il gradiente di $R(\tau)$, che poi useremo per costruire la policy (off–policy gradient). Alternativamente, si può stimare lo state–marginal inportance ratio, usando formulazioni in avanti o all'indietro dell'operatore di Bellman.

Tuttavia, questi metodi sono stati sviluppati per RL off–policy, che comunque ha la possibilità di ottenere nuovi sample online, riusando quelli vecchi per efficienza, ma non trovano applicazione nel caso offline. Infatti, tutti questi metodi presentano una grande varianza, che aumenta esponenzialmente con il differire della distribuzione reale da quella di $\mathcal{D}$. Nel caso online, questa differenza è minima perchè nuovi dati vengono sempre raccolti da distribuzioni simili, mentre nel caso offline può diventare molto significativa.

2.2 Metodi basati sulla Programmazione Dinamica

In linea di principio, tutti i metodi basati sulla programmazione dinamica (come Q–Learning o metodi actor–critic) possono essere usati nel caso offline. Queste tecniche sono state usate anche nel caso di approssimatori fatti da reti neurali, sia per funzioni lineari (es. LSTD–Q e LSPI) che non.

Tuttavia, nel caso in cui si elimini completamente la raccolta di informazioni online, questi metodi soffrono di diversi roblemi, tra cui il principale è il distributional shift (sugli stati e sulle azioni). Si viene infatti a creare un *unlearning effect*, tale per cui la curva di apprendimento tende a salire all'inizio e scendere in seguito. Questo problema è simile all'overfitting, ma rimane anche aumentando la dimensione di $\mathcal{D}$.

Per ovviare a questi problemi, sono state proposte strategie che possono essere raggrupate in due classi: metodi basati su policy constraint e metodi uncertainty–based.

Policy Constraint

L'idea di questi metodi è che la policy appresa non sia *troppo diversa* da quella sottostante a $\mathcal{D}$. Quindi, le azioni su cui Q sarà valutata non saranno mai diverse da quelle su cui è stata addestrata, eliminando una delle fonti di distributional shift.

In base a come esprimiamo questo concetto di vicinanza e a come richiediamo i vincoli, parliamo di:

- explicit f–divergence constraint, che vincolano l'actor update ad avere una policy vicina a quella del dataset usando una f–divergenza (misura di similarità tra distribuzioni) come distanza.
- implicit f–divergence constraint, in cui la similarità è data per costruzione dalla funzione actor update (AWR, AWAC, ABM).
- integral probability metric constraints, che esprimono vincoli con propiretà teoriche migliori per l'offline RL (es. BEAR).

Inoltre, i vincoli possono essere espressi come veri e propri vincoli o come penalità che modificano la reward.

Tuttavia, questi metodi portano ad apprendere una policy che è molto simile a quella che definisce il dataset, limitando in tal modo l'apprendimento. Infatti, nei casi d'uso reali questa distribuzione è spesso complessa e quindi difficile da stimare con precisione. I metodi che usano vincoli impliciti sono però promettenti per alleviare questo problema.

Inoltre, in questo come nei prossimi casi, si presenta il problema dell'accumulo di errori successivi (scalano quadraticamente rispetto all'orizzonte temporale). Per cercare di evitare questo, i vincoli usati devo essere molto stretti, limitando quindi ulteriormente l'apprendimento.

Uncertainty–based methods

L'idea alla base di questa classe di metodi è che se siamo in grado di valutare l'incertezza epistemica (cioè dovuta alla non completa conoscenza della distribuzione) della funzione Q, questa sarà maggiore su stati al di fuori della nostra distribuzione, e possiamo quindi usare questa stima per evitarli, stimando Q in modo conservativo.

Nella pratica però è difficile ottenere stime di incertezza ben calibrate, soprattutto usando stimatori di funzione complessi (come le reti neurali profonde).

2.3 Metodi Model–Based

Questi metodi cercano di stimare $T(s_{t+1}|s_t, a_t)$ usando un modello parametrizzato $T_\psi(s_{t+1}|s_t, a_t)$. Di conseguenza, possono beneficiare di tutte le tecniche di apprendimento supervisionato per il fitting del modello. Tuttavia, non sono immuni al problema del distributional shift. In primo luogo, il modello verrà usato per predire i risultati di un'azione secondo la policy che sta stimando, che è in generale diversa da quella del dataset. Inoltre, il tentativo di massimizzare $T_\psi(s_{t+1}|s_t, a_t)$ potrebbe portare a predizioni erroneamente alte per stati fuori dalla distribuzione (model exploitation).

Alcuni metodi che rientrano in questa categoria sono:

Metodi di RL standard alcuni metodi model–based, sviluppati per il caso online, sono stati usati con discreto successo nel caso offline, anche su sistemi piuttosto complessi;

Metodi off–policy + modelli dove il modello è usato per migliorare le performance di uno stimatore (es. importance sampling);

Vincoli su distribuzione e *safe–region* dove il modello è usato per apprendere dei vincoli sulla policy e limitarla in regioni sicure dello spazio degli stati (dove la possibilità di sbagliare è bassa), ad esempio i Deep Imitative Models (DIMs);

Modelli conservativi dove si cerca, con scelte conservative, di limitare analiticamente l'errore. Il metodo è simile a quello di CQL, dove la penalità viene messa sulla policy per evitare la visita di stati dove è probabile che il modello non sia corretto. Due esempi di questo metodo sono MoREL e MOPO, per cui è possibile dimostrare che la policy appresa si comporta bene anche nell'ambiente reale, se si conosce un buono stimatore per l'errore del modello.

Un ulteriore problema per questi metodi è che i modelli possono essere molto difficili da modellare, e si sta cercando di risolverli usando approcci ibridi (cioè che sono in parte model–based e in parte model–free).

3 Conservative Q–Learning (CQL)

3.1 Descrizione

Invece di imporre vincoli sulla policy, cerchiamo di regolarizzare la funzione Q per evitare di sovrastimare il suo valore in stati fuori dalla distribuzione. In questo caso la stima di Q sarà conservativa, ma senza bisogno di modellare esplicitamente l'incertezza.

In pratica, scegliamo di modificare la funzione obiettivo per il fitting di Q introducendo una penalità, e in base alla scelta di questo termine otterremo diversi algoritmi. Si può ad esempio scegliere una penalità che minimizzi i valori di tutti gli stati fuori dalla distribuzione (scegliendo in modo adversarial la distribuzione su cui la calcoliamo). In questo caso, se scegliamo bene il peso di questo termine di penalità, otteniamo una funzione Q che rappresenta un Lower Bound della vera funzione Q, e per questo parliamo di Q–Learning *Conservativo*.

Un problema di questa scelta è che è troppo conservativa e tende a sottostimare troppo la vera funzione Q. Una possibile soluzione consiste nell'aggiungere un altro termine alla funzione obiettivo che massimizzi il valore nel caso di tuple stato–azione nel dataset $\mathcal{D}$. Così facendo, perdiamo la proprietà di essere un Lower Bound in ogni punto, ma conserviamo quella di essere un Lower Bound in valore atteso, riducendo inoltre in maniera significativa la tendenza a sottostimare troppo Q.

3.2 Implementazioni

Questo algoritmo in pratica consiste nella sola modifica della funzione di aggiornamento di Q, come si può vedere nel pseudo-codice 1, e questo ci ha permesso di implementarlo al di sopra di altri algoritmi di RL che fanno uso di tale funzione. In particolare le due opzioni che siamo andati ad utilizzare sono:

- SAC (Soft Actor Critic) : per ambienti con action-space continuo (es. Ant-Bullet)
- QR-DQN (Quantile Regression - Deep Q-Learning) : per ambienti con action-space discreto (es. ATARI)

Algorithm 1 Conservative Q-Learning

Initialize Q-function, Q_θ, and optionally a policy, π_ϕ
for step t in {1,...,N} **do**
    Train the Q-function using G_Q gradient steps on objective
    $\theta_t \coloneqq \theta_{t-1} - \eta_Q \nabla_\theta CQL(\mathcal{H})(\theta)$
    (Use $\mathcal{B}^*$ for Q-learning, $\mathcal{B}^{\pi_{\phi_t}}$ for actor-critic)
    (**only for actor-critic**) Improve policy π_ϕ via G_π gradient steps on ϕ with SAC-style entropy regularization:
    $\phi_t \coloneqq \phi_{t-1} + \eta_\pi \mathbb{E}_{s\sim D, a\sim\pi_\phi(\cdot|s)}[Q_\theta(s,a) - \log \pi_\phi(a|s)]$
end for

Nelle nostre implementazioni abbiamo deciso di usare la variante chiamata $CQL(\mathcal{H})$ definita come segue:

$$\min_Q \alpha \mathbb{E}_{s\sim D}\left[\log \sum_a \exp(Q(s,a)) - \mathbb{E}_{a\sim\hat{\pi}_\beta(a|s)}[Q(s,a)]\right] + \frac{1}{2}\mathbb{E}_{s,a,s'\sim D}\left[\left(Q - \hat{\mathcal{B}}^{\pi_k}\hat{Q}^k\right)^2\right]$$

dove $\hat{\pi}_\beta$ è la polici che stimiamo dalle osservazioni del dataset, $\hat{\mathcal{B}}^{\pi_k}$ è l'operatore di Bellman empirico, anche questo stimato partendo dal dataset e $\hat{Q}^k$ rappresenta i valori calcolati dalla rete target per la Q–function.

CQL su SAC

In questo algoritmo andiamo a modellare sia una policy (actor) che una funzione Q (critic): il primo è implementato come una distribuzione normale dove media e varianza sono fornite da una rete neurale, il secondo invece consiste semplicemente in una rete neurale in cui l'ouput corrisponde al valore della funzione Q. In entrambe le reti sono presenti due layer fully-connected da 256 nodi con funzioni di attivazione ReLU.

L'implementazione di SAC usata utilizza due reti per modellare la funzione Q e prende sempre il minimo fra i due valori ottenuti, questo permette di evitare una sovrastima che peggiorerebbe la prestazione della policy. Oltre a questo il coefficente del termine dell'entropia presente nella policy non viene preso costante, ma viene minimizzato in modo da modificare nel corso del training la sua importanza.

Per quanto riguarda invece la formula introdotta da CQL sono state eseguite delle modifiche:

- la costante alpha del termine conservativo viene resa dinamica, massimizzandone il valore rispetto ad una costante di threshold

$$\min_{Q} \max_{\alpha \geq 0} \alpha \left(\mathbb{E}_{s \sim D} \left[\log \sum_{a} \exp(Q(s,a)) - \mathbb{E}_{a \sim \hat{\pi}_{\beta}(a|s)}[Q(s,a)] \right] - \tau \right) + \dots$$

- il calcolo del log-sum-exp viene sostituito con un importance sampling, essendo non possibile in uno spazio delle azioni continuo; perciò vengono estratti N valori di Q usando azioni eseguite dalla policy stimata e altrettante usando azioni scelte a random, per poi eseguire il calcolo del log-sum-exp con questi valori

CQL su QR-DQN

In questo algoritmo andiamo a modellare la funzione Q usando una rete neurale profonda. In particolare si tratta di una rete convoluzionale (figura 1) definita appositamente per un ambiente che simula un gioco dell'ATARI, quindi lo stato corrisponde all'immagine della schermata di gioco.

L'input della rete consiste nella concatenazione di 4 frame successivi della schermata, in modo da introdurre nello stato una dimensione temporale (es. capire quali oggetti si stanno muovendo). Prima di questo passaggio le immagini vengono preprocessate: partendo dall'immagine RGB viene calcolata la luminanza, riscalata da 210x160 a 84x84, e infine combinata con il frame precedente calcolando il massimo in ogni pixel (questo perché il video dei giochi ATARI era interlacciato a causa delle poche risorse a disposizione).

La rete è strutturata nel seguente modo:

- layer convoluzionale di 32 filtri 8x8 con stride 4
- funzione di attivazione ReLU
- layer convoluzionale di 64 filtri 4x4 con stride 2
- funzione di attivazione ReLU
- layer convoluzionale di 64 filtri 3x3 con stride 1
- funzione di attivazione ReLU
- layer fully-connected di 512 nodi

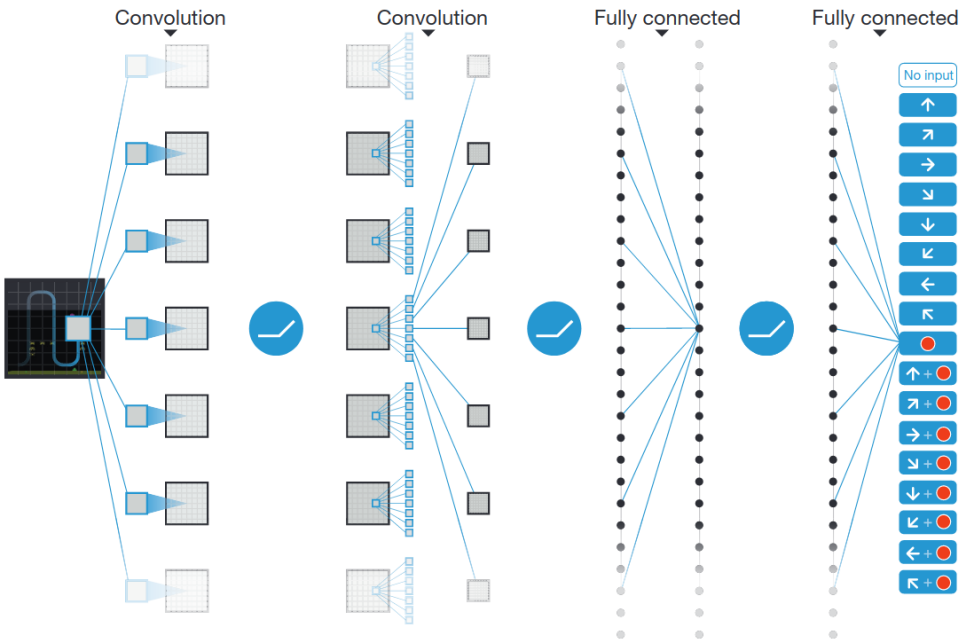

Figura 1: Struttura generale della rete convoluzionale deep usata in CQL su QR–DQN

- funzione di attivazione ReLU
- layer fully-connected con un numero di nodi pari alle azioni possibili

3.3 Risultati ottenuti

Abbiamo testato le nostre implementazioni su due dataset del pacchetto d4rl. Il training è stato eseguito parzialmente su colab e poi, a causa dei lunghi tempi richiesti e dei timeout di colab, su Azure ML, che offriva una velocità minore ma nessuna disconnessione per timeout.

Il salvataggio dei dati su disco è stato eseguito usando la libreria TensorBoard.

Caso discreto

Per testare l'algoritmo basato su DR–DQN abbiamo scelto di usare uno dei giochi dell'ATARI su cui non era stato testato, e la scelta è caduta su MsPacman. Il dataset è stato recuperato dal pacchetto d4rl–atari, e consiste in 1.000.000 di sample tratti dalle ultime iterazioni del training (opzione *expert*).

Gli iperparametri che abbiamo utilizzato sono:

- numero di frame: 10.000.000
- seed: 1
- batch_size: 32
- N: 200
- γ: 0.99
- τ: 1 (l'update della rete target non è soft ma vengono usati solo i nuovi pesi)
- learning rate: $5e^{-5}$
- ε_{adam}: 0.01/32
- α: 1
- aggiornamento della target: 2000 passi

Durante il training, ogni 100.000 frame abbiamo valutato la rete, per avere un'idea di come stesse procedendo il lavoro; questi risultati sono presenti in figura 2.

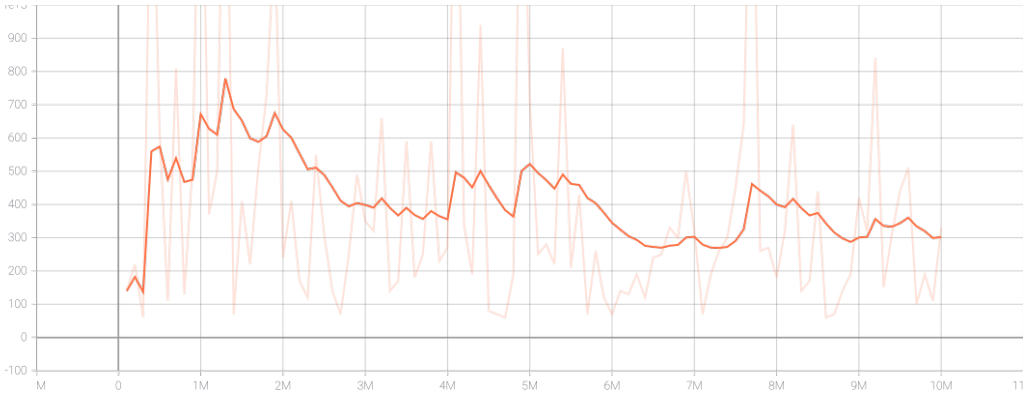

Figura 2: Valore della reward di un singolo episodio durente il training.

Il valore della loss per l'aggiornamento della funzione Q è mostrato in figura 3

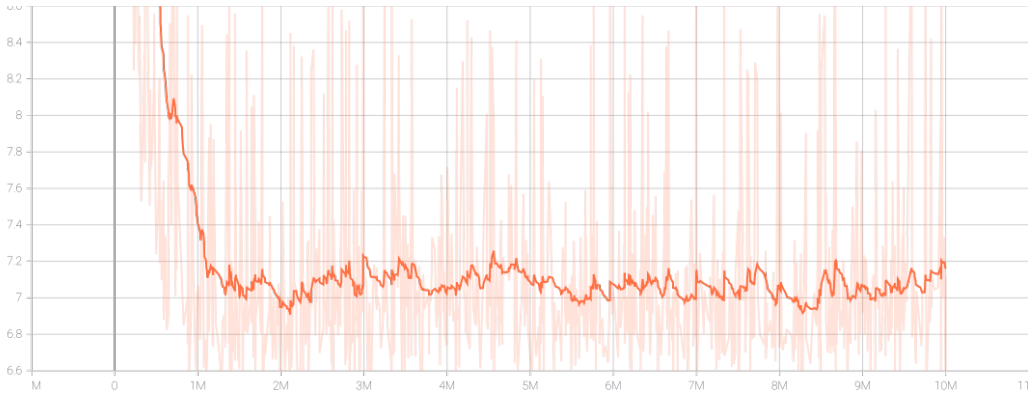

Figura 3: Valore della loss per l'aggiornamento di Q.

In questo caso, i risultati non sono ottimi. Questo è dovuto al fatto che i mezzi a nostra disposizione non ci hanno permesso di efettuare un training abbastanza approfondito (sull'ordine di qualche centinaio di milioni di frame). Tuttavia, dal grafico possiamo vedere come l'andamento sia comunque buono, con la loss che tende a diminuire nel tempo.

Caso continuo

Per testare l'implementazione basata su SAC abbiamo usato l'unico dataset di d4rl che non fosse già stato provato, cioè Ant. Per il training abbiamo usato il dataset fornito da d4rl–pybullet, mentre per la valutazione abbiamo usato bullet come motore fisico, poichè non disponevamo di una licenza per MuJoCo per usare d4rl.

Gli iperparametri che abbiamo utilizzato sono:

- numero di frame: 5.000.000
- seed: 1
- batch_size: 256
- γ: 0.99
- τ: $5e^{-3}$
- learning rate per la policy: $3e^{-5}$
- learning rate per la Q: $3e^{-4}$
- learning rate per α: $3e^{-4}$
- threshold: 10
- Numero di azioni per l'importance sampling (N): 10
- aggiornamento della target: 1 (ad ogni passo)

Il training di questo agente è stato più problematico: avendo esaurito il tempo a disposizione su Azure, abbiamo dovuto eseguirlo a pezzi su colab. Ogni volta che viene eseguita la simulazione per valutare il risultato, veniva anche salvato un checkpoint con lo stato delle reti e degli ottimizzatori, che poi poteva essere ripristinato da disco per continuare il training una volta che il timeout di colab lo interrompeva.

Purtroppo, a causa di queste continue interruzioni, i dati salvati mediante tensorboard sono andati persi, quindi non abbiamo grafici per stimare l'andamento. Possiamo però vedere il risultato della rete addestrata, che mediamente ottiene un punteggio di circa 500. Questo è in linea con i dati del dataset, in cui la reward media è di 570; però, anche in questo caso, il numero di frame su cui siamo stati in grado di addestrare l'agente è troppo basso, perciò i risultati non sono troppo buoni.