Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Relazione progetto computational learning

REINFORCEMENT LEARNING SUL GIOCO DI BLACKJACK

FRANCESCO BELLEZZA

Anno Accademico 2023-2024

INDICE

ELENCO DELLE FIGURE

1

INTRODUZIONE

1.1 SCOPO DEL PROGETTO

In questo elaborato esploreremo il gioco del blackjack, in particolar modo creeremo degli agenti all'interno di un contesto di reinforcement learning per cercare di fare giocare a blackjack il nostro computer. Tutto il codice verra' scritto in Python. Iniziamo pero' facendo un passo indietro, descrivendo le regole di questo gioco di carte.

1.2 CHE COS'E' IL BLACKJACK?

Il blackjack e' un gioco di carte dove e' presente un giocatore e il banco (che possiede il mazzo). Al principio della partita il mazziere ha una carta coperta ed una carta scoperta. La carta rivelata rappresenta il suo punteggio iniziale. L'altro sfidante invece possiede due carte scoperte e deve decidere se chiedere al mazziere una carta aggiuntiva oppure di tenere le carte che gia' ha. Lo scopo consiste nel cercare di arrivare piu' vicini possibili alla somma 21: chi piu' si avvicina a tale somma vince la mano. Il giocatore puo' chiedere quante carte vuole al mazziere, solamente che se per caso raggiunge come punteggio un numero maggiore di 21, allora perde. Supponiamo che il giocatore riesca ad ottenere una somma vicina a 21 e decida di fermarsi (ad esempio 20). Come gioca il mazziere? Il mazziere deve obbligatoriamente svelare la carta che ha coperta e deve continuare a pescare carte dal mazzo fino a raggiungere una somma maggiore o uguale a 17. Se il mazziere sfora, allora il giocatore ha vinto, indipendentemente dalla somma che possedeva. Altrimenti, se nessuno sfora il punteggio di 21, bisogna vedere chi si avvicina maggiormente a 21. Se il punteggio e' pari, abbiamo una situazione di parita'. Le figure hanno questi valori:

- l'asso puo' valere 1 o 11 a seconda della scelta del giocatore

- le figure (fante, donna e re) valgono ciascuna 10 punti
- le altre carte valgono un numero di punti pari al valore della carta.

Per il nostro gioco supporremmo che il mazzo abbia infinite carte.

1.3 REINFORCEMENT LEARNING E AGENTI

Considerando che affronteremo il tema del Reinforcement Learning, verra' presentata una introduzione generale all'argomento. Questo tipo di apprendimento si distingue da altri tipi per l'assenza di un supervisore (come gia' visto ad esempio nel clustering, quindi il Reinforcement Learning non e' l'unico metodo di apprendimento senza supervisore). Le componenti in gioco in questo mondo sono tre: un ambiente, un agente e l'osservazione. L'agente e' colui che sceglie quali azioni compiere in base ad una ricompensa (reward). La ricompensa viene fornita dall'ambiente, ed e' grazie a quest'ultima che vengono prese delle decisioni in una direzione piuttosto che in un'altra. In termini matematici diremmo che l'agente deve massimizzare la "discounted cumulative reward", ovvero la somma di tutte le ricompense dall'istante attuale in poi. Ci sono alcuni casi in cui l'agente non osserva completamente l'ambiente, ma non rientriamo in questa casistica, in quanto nel blackjack l'agente osserva chiaramente le carte scoperte che sono fornite al giocatore e al mazziere (siamo quindi in un Fully Observable Environment).

1.3.1 *Componenti Reinforcement Learning*

Le tre componenti principali nel Reinforcement Learning sono:

1. Policy: dato uno stato S_t seleziona quale azione intraprendere. Ne esistono di due tipologie: deterministica e stocastica. Nel caso stocastico la policy e' descritta come una probabilita', che rientra perfettamente nel nostro scenario [1]: $\pi(a|s) = P(A_t = a|S_t = s)$
2. Value Function: funzione che valuta la "bonta'" di uno stato in termini di reward. Puo' valutare anche il valore della coppia stato-azione, dipende dal contesto in cui la usiamo. Noi ci troviamo nel secondo scenario [1]: $q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$ Questo e' il valore atteso ottenuto seguendo la policy π partendo dallo stato s.

3. Model: dato un determinato stato S_t predice la ricompensa R_t e il prossimo stato S_{t+1}

Per essere piu' precisi, il modello nel nostro caso descrive un Markov Decision Process, il quale e' caratterizzato da:

- un insieme di stati S
- un insieme di azioni A
- una funzione di probabilita' P, che ci indica la probabilita' di trovarci in un certo stato successivo, dato che mi trovo nello stato corrente ed ho svolto una determinata azione, in simboli [1]: $P(S_{t+1} = s' | S_t = s, A_t = a)$
- una reward. La reward successiva viene calcolata come valore atteso, in simboli [1]: $\mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- un valore gamma utilizzato per la discounted cumulative reward. In formule[1]: $G_t = \sum_{k=1}^{\infty} \gamma^k R_t$. Questo parametro gamma prende il nome di discount factor per il Markov Decision Process.

Nel caso del blackjack, siamo nella situazione in cui in realta' il modello sarebbe possibile descriverlo, ma alquanto poco pratico. Infatti il numero possibile di stati da descrivere e' 704 (32 x 11 x 2) dove 32 sono i possibili punteggi del player, 11 i possibili punteggi iniziali del mazziere, e 2 rappresenta la possibilita' del player di usare un jolly o meno. (Verra' esplicitata la struttura dello spazio delle osservazioni piu' avanti). Quindi modellare le probabilita' di uno spazio cosi' ampio e' alquanto costoso e non consigliabile. Abbiamo parlato brevemente degli elementi che contraddistinguono il reinforcement learning. Adesso dobbiamo pero' unirli insieme. In concreto, l'agente di reinforcement learning effettua le seguenti azioni:

1. Compie una decisione, basandosi sulla propria policy (che inizialmente e' per forza casuale, in quanto l'agente non ha appreso ancora niente)
2. Calcola il valore della value function, che non e' altro che un valore atteso delle reward future (quindi si valuta le qualita' delle proprie decisioni)
3. Aggiorniamo la nostra policy prendendo quella che massimizzi la value function (quindi quella che ci fornisce un migliore risultato), e

si riparte dall'inizio. Per compiere una decisione pero', non sempre conviene scegliere la scelta piu' greedy. Infatti, comportandosi in questo modo, potremmo privarci della possibilita' di trovare un percorso che inizialmente e' meno redditizio dal punto di vista della reward, ma che infine ci premia rispetto alla strategia greedy. Qui entrano in gioco le politiche epsilon-greedy, che, con probabilita' epsilon, puo' succedere che il nostro agente prenda una decisione differente rispetto alla migliore (locale). Ricordiamo che esistono casi in cui la decisione greedy coincide con la soluzione ottimale.

1.4 ERRORI DELLA PRECEDENTE CONSEGNA

In questa revisione del progetto, sono stati corretti gli errori che erano presenti nella precedente consegna. Anticipando brevemente l'entita' di questi errori, si basavano sulla correttezza dei due algoritmi e anche sull'errore di valutazione riguardo l'agente che si comporta in maniera casuale (dovuta ad un errore della funzione di testing e dovuta al fatto che, erroneamente, si definiva policy casuale quella ottenuta da un agente con epsilon = 1, senza epsilon decay. In realta', questo tipo di policy, performera' meglio rispetto alla policy completamente causale).

2

ENVIRONMENT: GYMNASIUM

Come ambiente abbiamo utilizzato quello fornito da gymnasium sul blackjack. Lo spazio delle osservazioni e' definito da una terna di valori:

- il valore della somma delle due carte del giocatore
- il valore della carta scoperta del mazziere
- una variabile booleana che rappresenta la possibilita' o meno da parte del giocatore di cambiare valore all'asso. (Che ricordiamo puo' valere sia uno che undici).

Inizialmente l'ambiente considera l'asso come 1, poi assegna il valore 1 alla variabile booleana se e solo se, aggiungendo 10 al punteggio del player, non si va sopra i 21. Per inizializzare l'ambiente e' sufficiente scrivere la seguente riga di codice:

```
env = gym.make('Blackjack-v1', natural=False, sab=False)
```

Inoltre, se vogliamo visualizzare la schermata di gioco durante il training e' sufficiente passare come argomento al metodo make il parametro render_mode e passargli come valore "human". In tal caso, otterremo una schermata di questo genere:

Figura 1: Il mazziere mostra il re, con valore di 10 punti. Il punteggio delle due carte del player e' uguale a 15.

Il parametro natural indica se attribuire al blackjack naturale (ovvero il player ha un asso e una figura) una reward di 1.5 invece di 1. Nel nostro caso abbiamo dato alla reward sempre 1 in caso di vincita.

3

AGENTI

Le ricompense che vengono fornite agli agenti sono le seguenti:

- +1 se il giocatore vince
- +0 se il giocatore pareggia
- -1 se il giocatore perde

Ricordiamo che essendo nella situazione senza modello, noi non calcoliamo la qualita' di uno stato, ma calcoliamo la qualita' della coppia stato azione (che prende il nome di Q-function).

3.1 AGENTE MONTECARLO

L'agente Montecarlo e' un agente che puo' calcolare la q function solo esclusivamente quando viene calcolata la "discounted cumulative reward", quindi solamente alla fine del gioco di blackjack che coincide con la fine dell'episodio. In realta' in questo caso, la discounted cumulative reward e' uguale alla reward dell'istante finale (in quanto le ricompense vengono attribuite solo a fine gioco). Nonostante cio', e' stata presa la decisione di calcolare la reward cumulata come previsto dall'algoritmo senza introdurre semplificazioni per uniformita' di notazione. Inoltre e' stato scelto di implementare un approccio incrementale, in modo tale da semplificare il calcolo dei valori della q-function. Abbiamo usato questa formula [1]: $Q[S, A] = Q[S, A] + \alpha \cdot (G - (Q[S, A])$, dove G e' la reward cumulata dell'episodio. L'algoritmo implementato e' il first-visit MonteCarlo. Forniamo l'algoritmo in pseudo codice dell'approccio non incrementale [2]: Nel nostro caso, al posto di V abbiamo una q-function (perche' ci ritroviamo in un contesto model-free). Quindi vengono valutate coppie (stato, azione) e non i singoli stati. La policy π utilizzata e' epsilon greedy (che utilizza valori della q-function che vengono aggiornati durante l'algoritmo: quando generiamo un nuovo episodio, i valori della q-function

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figura 2: Pseudocodice MonteCarlo non incrementale

si sono modificati). La principale differenza da questo approccio rispetto a quello da noi implementato consiste nell'update della q-function. Infatti, nel caso non incrementale dobbiamo salvare i vari valori di G per ogni coppia (stato, azione) e poi ogni volta farne la media di quelli raccolti per quella coppia, mentre nel caso incrementale utilizziamo la formula definita precedentemente per risparmiare spazio e tempo di esecuzione del programma. Spazio in quanto non salviamo il vettore returns per ogni (stato, azione), tempo perche' per fare la media bisogna sommare ogni volta tutti gli elementi interni a quella determinata lista. In questo caso introduciamo un nuovo parametro α, che e' il learning rate dell'algoritmo.

3.2 AGENTE TEMPORAL-DIFFERENCE

Gamma rappresenta il discount factor del Markov Decision Process. In questo caso per aggiornare la q-function utilizziamo la seguente formula [1]: $Q[S, A] = Q[S, A] + \alpha * (R + \gamma(Q[S', A']) - Q[S, A])$
Dove S' e A' sono rispettivamente lo stato e l'azione temporalmente successivi allo stato S ed A. L'azione A' viene scelta tramite l'approccio epsilon-greedy. Qui sotto riportiamo lo pseudocodice come da riferimento [2]: Oltre ad una differenza nell'update della q-function rispetto all'agente MonteCarlo, in questo caso l'episodio non viene generato a priori, ma viene generato passo passo, mentre vengono aggiornati anche i valori della q-function.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 3: Pseudocodice dell'agente Temporal Difference

3.3 SCELTA DEL LEARNING RATE E DELL'EPSILON

Entrambi gli agenti adottano la strategia epsilon-greedy. Ovvero, non sempre viene scelta come azione successiva l'azione piu' redditizia nel breve termine, ma, con probabilita' uguale ad epsilon viene selezionata una decisione completamente casuale dallo spazio delle possibili azioni disponibili. Sono stati implementati in questa versione del progetto due fattori di decadimento: uno lineare e uno esponenziale. Il fattore lineare e' identico a quello implementato da gymnasium [3]. Il fattore esponenziale, invece, lo abbiamo definito in questo modo [4]: $N(t) = N_0 e^{-\mathrm{decay} \cdot t}$, dove N_0 e' il primo valore di epsilon, decay e' il fattore di decadimento, infine t e' il numero dell'episodio. Entrambi fanno decadere l'epsilon fino a quando non si raggiunge una certa soglia, da noi impostata.

4

GUIDA

4.1 STRUTTURA DEL PROGETTO

Per avere una maggiore chiarezza mentale, il codice e' stato riscritto da zero, con una suddivisione in un numero minore di classi, consentendomi in questo modo, di concentrarmi maggiormente sugli aspetti concettuali e revisionando il mio precedente elaborato.
La versione di python che e' stata utilizzata e' la 3.11.5. Controllare quindi di avere la corretta versione di python nel sistema. Il progetto e' composto da diversi file. Il file di esecuzione di interesse e' "main.py". Oltre a questo file abbiamo aggiunto un secondo file dove abbiamo inserito alcune funzionalita' riguardo i grafici. Entrando piu' nel dettaglio:

- Main: contiene tutte le implementazioni degli agenti e i vari training e testing effettuati.
- Plotting: contiene dei metodi di utilita' per mostrare vari grafici relativi ai vari aspetti del training e per vedere a fine allenamento i valori delle policy e dei q-value.

4.2 COME USARE IL CODICE

Inanzittuto e' necessario avere i pacchetti python installati. E' presente all'interno della cartella un file chiamato "requirements.txt". Utilizzando pip, si possono installare i pacchetti sfruttando il comando "pip -r install requirements.txt". All'interno del file main.py sono presenti due metodi: uno dei due trova la migliore configurazione di alcuni parametri da noi selezionati, l'altro invece valuta una unica impostazione. All'inizio dell'esecuzione del programma, verra' chiesto se si vuole trovare la migliore configurazione o se si vuole provare un unico scenario. Se si vuole provare piu' configurazioni, bisogna premere 1, qualunque altro

tasto per provarne una soltanto. Alcuni parametri di interesse che possono essere trovati nel codice sono:

- seed: serve per rendere i vari esperimenti riproducibili. Viene settato il random state dell'environment
- n_episodes: riguarda il numero di episodi utilizzati durante il training dei due agenti
- n_tests: numero di prove per poter testare le policy di entrambi gli agenti
- ci sono diversi parametri riguardanti epsilon: questo e' dovuto al fatto che sono stati implementati degli epsilon decay, la cui funzionalita' consiste nel rendere l'agente piu' dinamico durante l'apprendimento (ovvero all'inizio del training cerchera' di esplorare maggioramente lo spazio delle osservazioni per imparare, successivamente invece cerchera' piano piano di prediligere l'approccio greedy)
- alpha: consiste nel learning rate dell'algoritmo. Piu' e' alto il learning rate e piu' l'algoritmo impara da ogni singolo episodio.
- gamma: e' il parametro che ci indica quanto vengono considerate le reward precedenti. 1 se si tiene il conto di tutte le reward passate, 0 se invece si tiene conto solo della reward attuale. E' collegato alla discounted cumulative reward.

Per rendere i confronti onesti tra i vari agenti, per ogni training e per ogni test di ogni singolo agente viene utilizzato un seed diverso. Il metodo mc_td_evaluation e' il metodo che consente di provare un particolare tipo di configurazione. L'altro metodo, get_best_config e' utilizzato per provare alcuni settaggi di parametri da noi prestabiliti.

5

PROVE SVOLTE E CONSIDERAZIONI FINALI

In questo capitolo verranno visualizzati alcuni grafici di interesse e alcuni risultati ottenuti dai due agenti con configurazioni diverse. Nella prossima sezione discuteremo dei problemi della precedente consegna e di come sono stati risolti.

5.1 PROBLEMI RISCONTRATI DURANTE IL PROGETTO

Durante varie prove e' stato riscontrato un problema con il seed: nonostante sia stato impostato il seed dell'environment, il seed relativo all'action space segue una impostazione a parte che va configurata manualmente (cosa che successivamente e' stata svolta).
Precedentemente, capitava che con temporal difference si registrassero dei picchi anomali nei valori della q-function: abbiamo scoperto che dietro a questo comportamento in realta' si celava un errore derivato dall'errata implementazione dell'algoritmo. Per quanto riguarda la correttezza di temporal difference infatti, avevamo un problema principale: non avevamo considerato il caso in cui il prossimo stato potesse essere uno stato terminale. In quel caso, il valore della q-function e' 0. Questo puo' essere risolto moltiplicando il q-value della prossima azione e del prossimo stato per la variabile terminated negata. Invece per quanto riguarda l'agente MonteCarlo, era presente un problema di indici, infatti si usava lo stesso indice per definire il numero di episodio corrente e per l'esponente di gamma. Il problema e' che l'indice i veniva aumentato due volte, e c'era il rischio di sovrascrivere l'indice dentro al for con l'indice usato da gamma. L'agente MonteCarlo e' stato riscritto da zero, usando l'implementazione fornita dal libro [2], per l'algoritmo first-visit (con approccio incrementale, come specificato precedentemente). Ottenevamo i risultati sbagliati in fase di testing perche' era presente un errore nella funzione che era adibita alla prova dei risultati ottenuti. Inoltre, per ottenere risultati migliori in questa consegna, abbiamo aumentato il numero

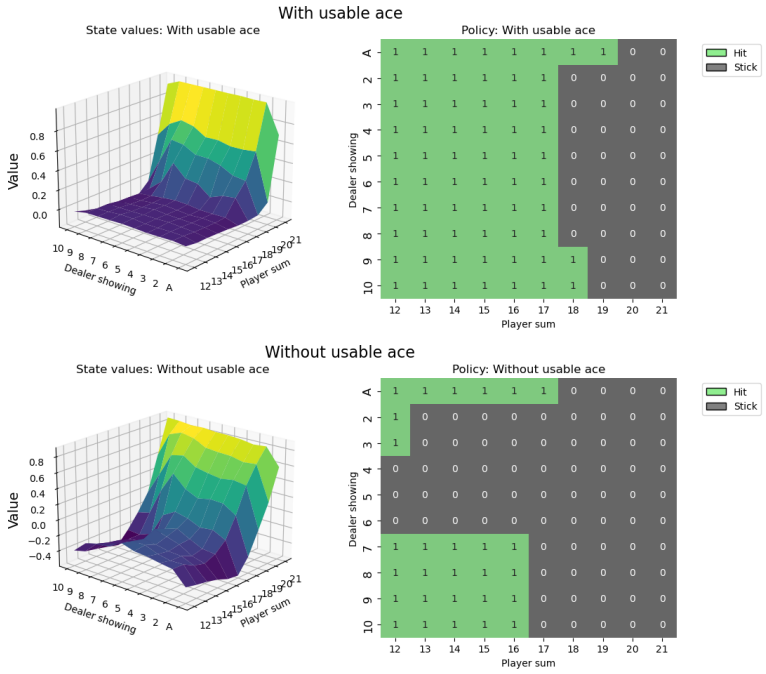

Figura 4: Q-learning con jolly e senza jolly di gymnasium

di episodi in fase di training, come vedremo piu' avanti.

5.2 CONFRONTO CON AGENTE IMPLEMENTATO IN GYMNASIUM

Abbiamo confrontato alcuni risultati dei nostri agenti con un agente di q-learning implementato dalla pagina gymnasium [3]. Questo confronto ci e' risultato utile in fase di progetto per capire se effettivamente i risultati ottenuti ci portavano verso la giusta direzione. I grafici che abbiamo visualizzato in questo progetto sono stati realizzati sfruttando le funzioni presenti in [3], in modo da avere un confronto diretto con l'agente di q-learning. E' stato leggermente modificato il file realizzato da gymnasium, in particolare abbiamo modificato il parametro rolling_length. Tale parametro serve per cercare di ottenere dei grafici piu' lisci (smooth), sfruttando una convoluzione. Rolling_length definisce la finestra della convoluzione, e per un numero di episodi uguale a un milione, abbiamo deciso di ampliarla per avere una maggiore leggibilita' (da 500 siamo passati a 1000). I parametri utilizzati da loro sono: learning rate = 0.01, epsilon iniziale = 1 con epislon decay lineare e numero di episodi = 100000. Adesso cercheremo di ottenere risultati simili effettuando diverse configurazioni di iperparametri.

5.3 RISULTATI OTTENUTI

Maneggiando i vari iperparametri utilizzati, siamo riusciti a raggiungere un winrate massimo del 43% circa, rispetto al 28% circa per la policy completamente casuale. In questa sezione riporteremo alcuni grafici relativi ad alcune prove che abbiamo effettuato. Come si potra' notare, la policy e i valori della q-function non sono poi cosi' dissimili da quelli forniti da gymnasium. Abbiamo impostato i seguenti vettori di parametri per la model selection:

- $\alpha = [0.001, 0.0055, 0.01]$
- $\gamma = [0, 0.5, 1]$
- $\epsilon = [start\_epsilon = 1, decay = start\_epsilon/$ $/(nepisodes/2), end\_epsilon = 0.1]$
- Decay = lineare oppure esponenziale

Per un totale di 18 differenti configurazioni. Ogni configurazione e' stata allenata su un numero di partite uguale a 1 milione, e sono state testate le varie policy su 100000 partite. Abbiamo deciso di favorire il numero di episodi in training per ottenere delle policy migliori, mentre il numero di episodi di test non serve che sia necessariamente grande come gli episodi di training per capire se una policy e' buona oppure no. Aumentare la grandezza del test serve per ottenere stime piu' attendibili. I risultati dei due agenti migliori verranno visualizzati nei grafici sottostanti (con seed = 1234): La policy in realta' risulta essere molto simile a quella dell'agente q-learning fornita da gymnasium. Dal grafico della durata degli episodi durante le fasi di training, possiamo supporre che negli episodi finali, siccome il fattore di epsilon e' decaduto, e' probabile che l'agente abbia imparato una buona azione che faccia chiudere velocemente la partita. Siccome volevamo essere sicuri che non si celasse un errore dietro a questo comportamento della durata degli episodi, per rimuovere ogni possibile dubbio, abbiamo inserito manualmente un milione di episodi nel training dell'agente di gymnasium, e abbiamo notato un andamento simile delle reward e della lunghezza degli episodi: quindi siamo maggiormente propensi a escludere l'ipotesi di un errore nascosto nel codice. Inoltre, globalmente, per entrambi gli agenti, le rewards hanno un andamento crescente: questo significa che l'agente sta imparando una strategia per vincere sempre di piu'. Oltre che agli episodi finali, anche gli episodi iniziali sono di durata ridotta.

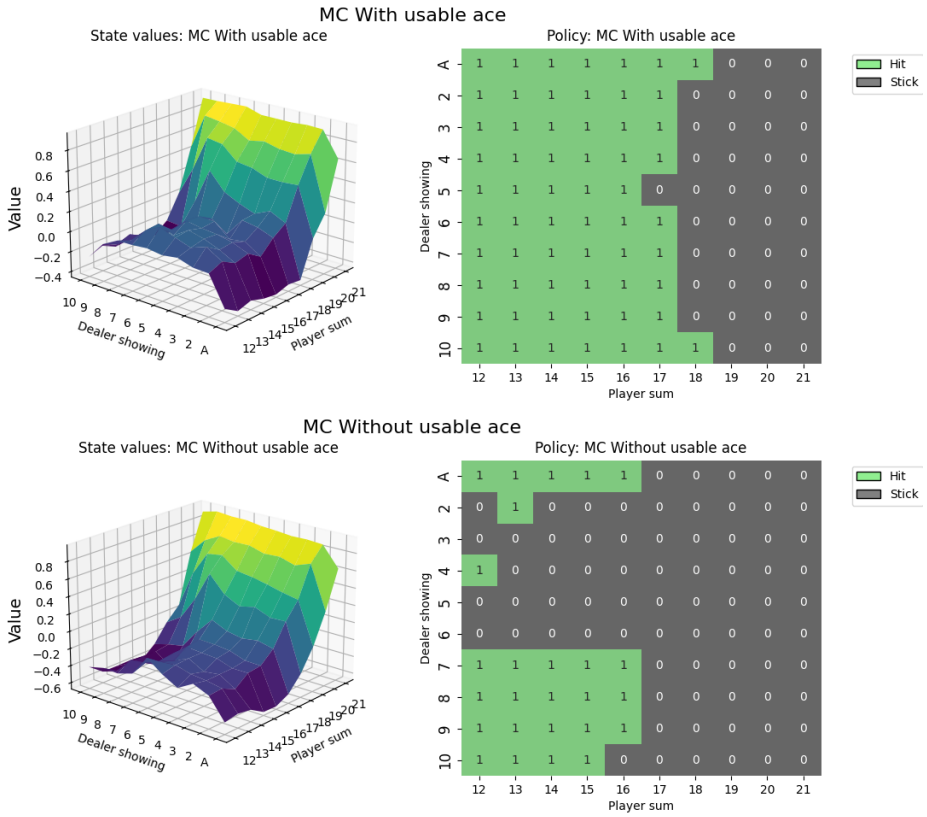

Figura 5: MonteCarlo valori della q-function e policy con o senza jolly.

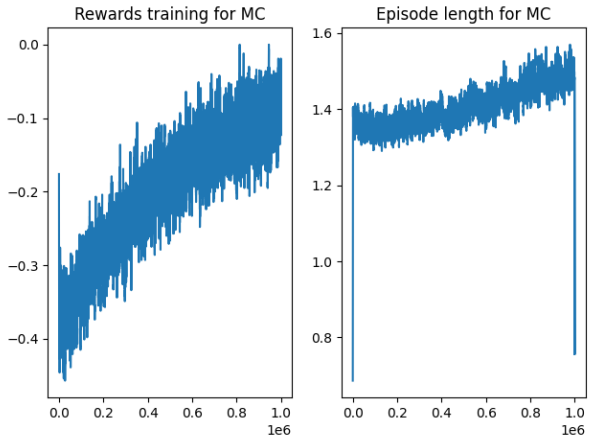

Figura 6: Rewards e durata degli episodi durante il training per l'agente Monte-Carlo

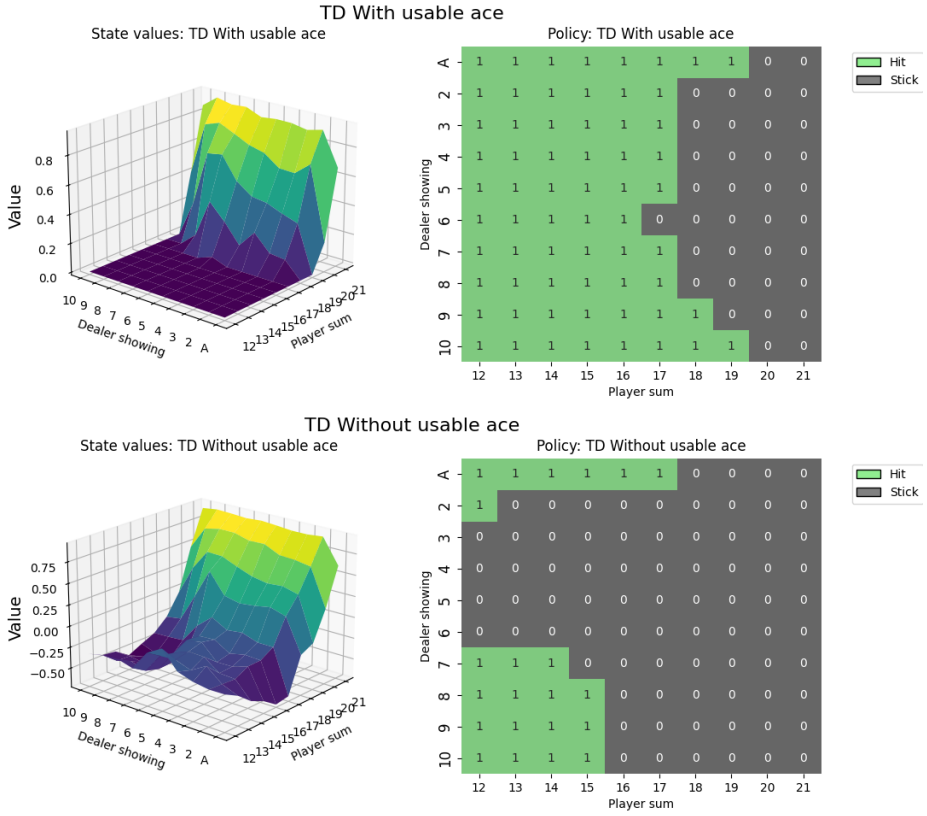

Figura 7: Temporal difference: valori q-function e policy con e senza ace

Potrebbe essere lecito pensare che dipenda dal fatto che durante i primi episodi abbiamo un valore di epsilon molto alto. Abbiamo un winrate in testing per la policy trovata dall'agente MonteCarlo del 43.18 % circa su 100000 episodi. I parametri che sono stati selezionati dal modello sono i seguenti: $\alpha = 0.0055, \gamma = 1..., \texttt{fattore\_di\_decay} = exp$. Per quanto riguarda l'agente temporal difference invece, abbiamo ottenuto risultati molto simili (43.3 % circa di winrate) con i seguenti parametri: $\alpha = 0.0055, \gamma = 0, \texttt{fattore\_di\_decay} = exp$.
In nessuno dei due casi e' stato scelto il fattore di decay lineare. La model selection non prende in considerazione un grandissimo numero di parametri a causa delle mancate risorse computazionali in mio possesso, infatti, con questo setting, il mio personal computer ha impiegato all'incirca quattro ore di tempo di esecuzione. Per rewards in entrambi i casi intendiamo la somma delle rewards all'interno di un episodio.

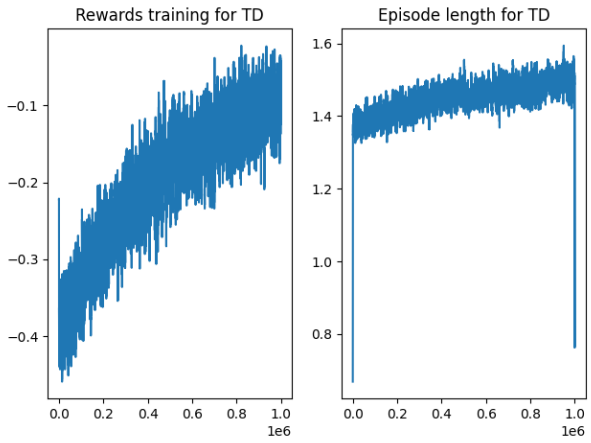

Figura 8: Temporal difference: rewards e lunghezza dell'episodio

5.4 RIFLESSIONI SULLA CONSEGNA PRECEDENTE

Considerando gli errori della precedente consegna, e' stata presa la decisione di confrontare i vari agenti con un agente casuale, per vedere se effettivamente e' presente una differenza. Alla fine, si riscontra che si hanno maggiori vincite da un qualsiasi agente di reinforcement learning rispetto a quello casuale. Perche', logicamente, non ha alcun senso lasciare la mano quando abbiamo come somma 10 o meno. Ci porterebbe ad una situazione di sconfitta in ogni caso, non importa che il mazzo considerato sia infinito. Precedentemente siamo caduti nella fallacia logica di pensare che se non ci potessimo avvantaggiare contando le carte, non sarebbero esistiti altri modi per avvantaggiarsi ma questo, in virtu' di quello che e' stato detto, e' falso. Inoltre abbiamo compreso che vi e' una sostanziale differenza tra una policy completamente casuale e una policy ottenuta scegliendo le azioni in base ai valori massimi della q-function ottenuti con un agente con epsilon-greedy = 1. Sono due casistiche diverse e, tra le due, quella che performa significativamente peggio e' quella della policy completamente casuale.

5.5 RISULTATI A CONFRONTO CON POLICY CASUALE

In questo paragrafo confronteremo le policy ottenute da due agenti (uno MonteCarlo e uno Temporal Difference) con una policy completamente casuale. Abbiamo deciso di allenare entrambi gli agenti su un milione di episodi e di testarli su centomila partite. Gli altri parametri utilizzati

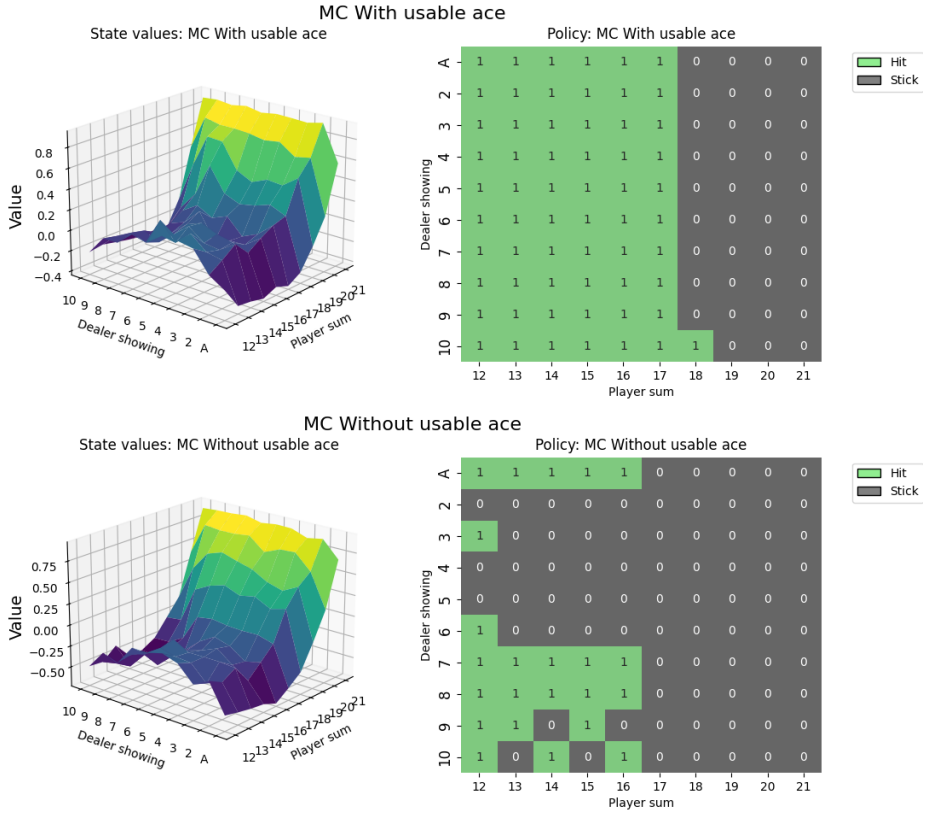

Figura 9: MC con e senza jolly. Valori della q-function e policy (esempio per confronto con policy casuale)

sono i seguenti:

- seed = 101
- i parametri di epsilon sono gli stessi utilizzati quando abbiamo provato piu' configurazioni
- fattore di decay lineare
- $\alpha = 0.01$
- $\gamma = 0.95$

Le vittorie ottenute in episodi lunghi centomila sono:

- 42.945 % per la policy trovata dall'agente MonteCarlo
- 42.809 % circa per la policy trovata dall'agente Temporal Difference
- 28.01 % circa per la policy casuale

Come possiamo vedere, la policy casuale performa peggio.

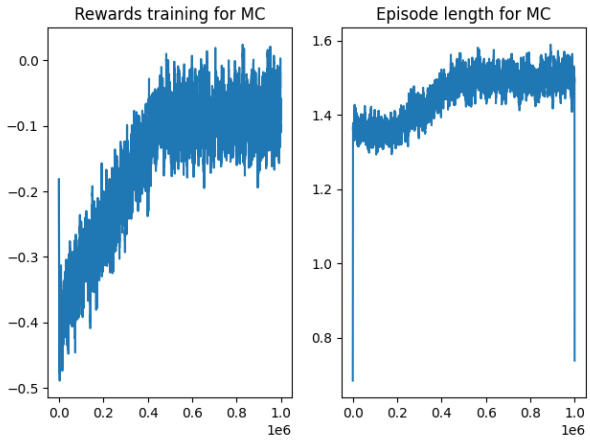

Figura 10: Rewards e test nel training di MonteCarlo (esempio per confronto con policy casuale)

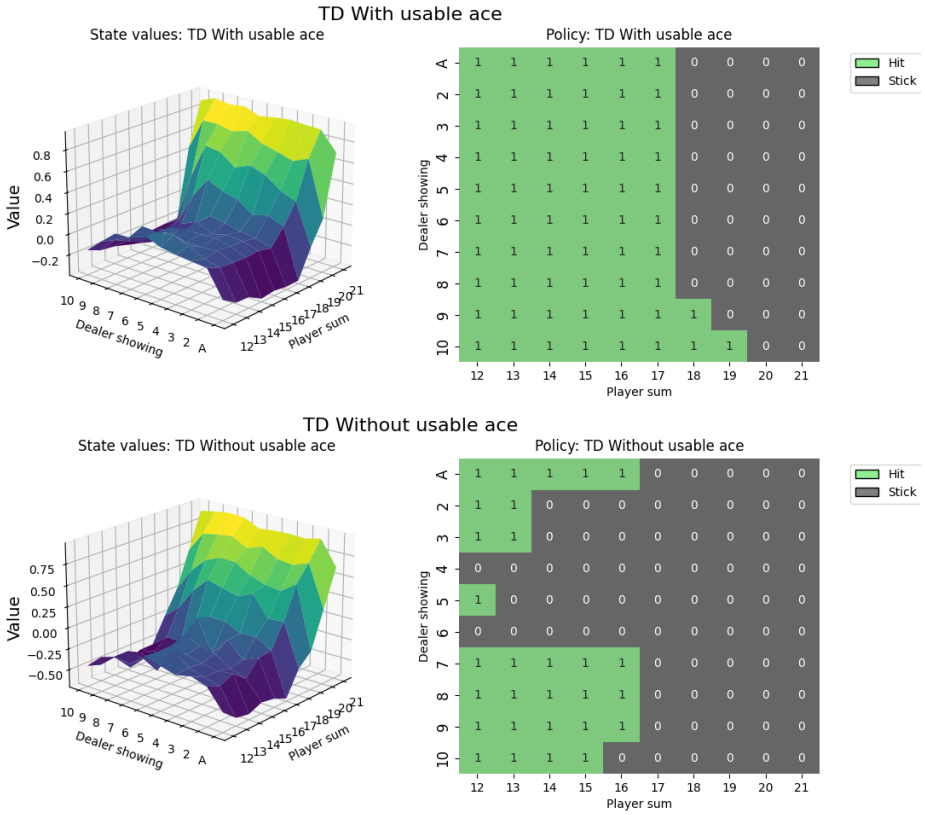

Figura 11: Temporal difference con e senza ace. Valori della q-function e policy (esempio per confronto con policy casuale)

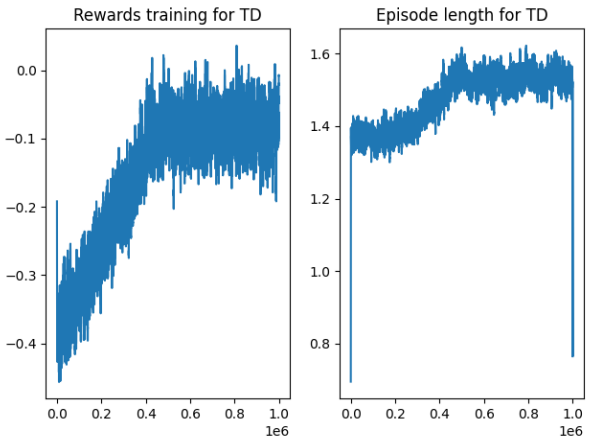

Figura 12: Rewards temporal difference e lunghezza degli episodi in training (esempio per confronto con policy casuale)

5.6 CONSIDERAZIONI FINALI E COMMENTI PERSONALI

Noi ci siamo limitati solamente ad osservare e a commentare alcuni risultati che abbiamo ottenuto durante alcuni settaggi di parametri. Questo tipo di progetto non e' stato facile da realizzare: molte volte ho avuto il dubbio relativo all'implementazione di qualcosa. Questo e' dovuto al fatto che sono difficilmente riscontrabili eventuali errori, a meno che non si verifichino eventi molto particolari. Anche se la matematica dietro questi algoritmi probabilmente e' piu' semplice di quel che si pensi, richiede un po' di tempo per essere assimilata e compresa a fondo. Inoltre e' stato particolarmente ostico impostare i seed in modo da rendere sia riproducibili i risultati, sia rendere onesti i confronti tra i vari agenti, fornendo loro episodi diversi su cui allenarsi (idealmente se si allenassero e testassero le loro abilita' sugli stessi scenari si controllerebbe solamente quanto un agente performa meglio su uno scenario specifico, perdendo l'aspetto relativo alla generalita' della sua risoluzione).

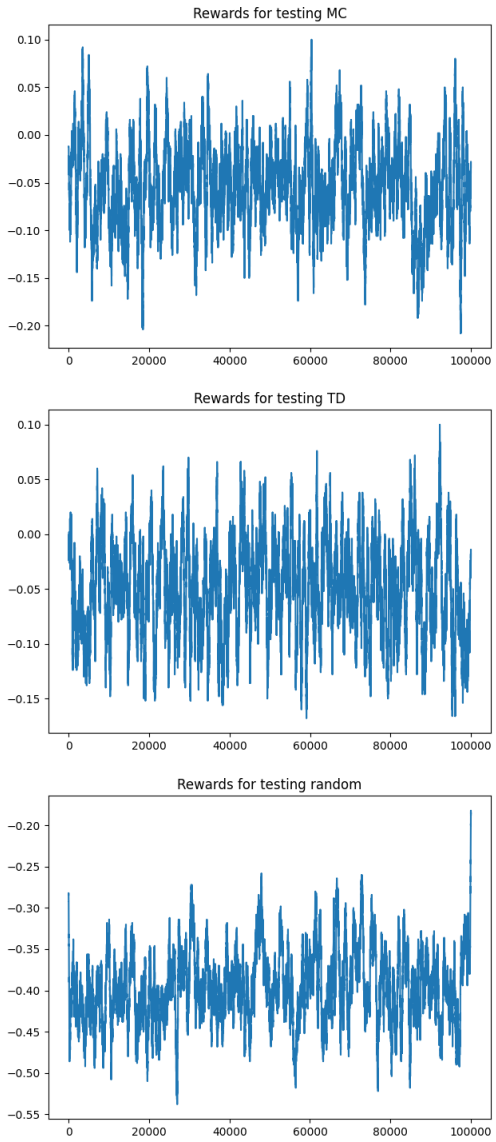

Figura 13: Andamento delle rewards per quanto riguarda le tre policy. Possiamo gia' notare che, se guardiamo l'asse delle y, l'agente casuale tende sempre ad avere rewards negative, a differenza degli altri due agenti

BIBLIOGRAFIA

[1] Autore Daniele Castellana - *Note sul Corso B031235 (B059) - COMPUTATIONAL LEARNING (CURRICULUM: DATA SCIENCE - E57) 2023-2024* (Cited on pages 4, 5, 8, and 9.)

[2] Autore R. Sutton e A. Barto - *Reinforcement Learning: An Introduction 2020. [Online]* - http://www.incompleteideas.net/book/RLbook2020.pdf

[3] *-https://gymnasium.farama.org/tutorials/training_agents/blackjack_tutorial/* (Cited on pages 8, 9, and 13.)

[4] *-https://en.wikipedia.org/wiki/Exponential_decay*
(Cited on pages 10 and 14.)