Lez18_ReinforcementLearning

December 1, 2023

0.1 Errori comuni nelle consegne

- Le operazioni di pre-processing, come rimozione ed aggiunta di nuove righe per preparare il training set, devono essere applicate anche al testing set.
- NON tutto può essere fatto sia su training set sia su testing set. La rimozione di una feature va bene, ma non posso scegliere arbitrariamente quali dati mi piacciono o meno.
- La scelta del modello è veloce, però il funzionamento di una rete neurale richiede molta ottimizzazione di iperparametri. Esistono dei framework per la scelta.

Nella soluzione del prof (che non è la migliore) troviamo: - La colonna `country` ha tanti valori categorici diversi. Se ci applichiamo `OneHotEncoding` otteniamo una feature per ogni valore. Se ho 20 valori, allora 20 feature. Ma la nazione era così informativa? Oltretutto, il dataset è portoghese, con molti dati di clienti Portoghesi. Potevamo introdurre una feature `ospite internazionale`, che indica se il cliente è portoghese (valore 0) o meno (valore 1).

- Potevamo accorpare *bambini* e *ragazzi*, numero di notti, se un cliente ha mai cancellato una prenotazione. La colonna relativa ai pasti poteva essere rimossa.

Passiamo al modello `build.model(hp)`: Introduciamo **Keras Tuner**, ovvero la definizione del modello riceve *hp* (hyperparameter), in cui invece di un valore vero mettiamo un riferimento, ad esempio:

```
units=hp.Int("units", min_value = 50, max_value = 150, step = 15)
```

E' parametrico, e i valori ancora non sono stati introdotti.

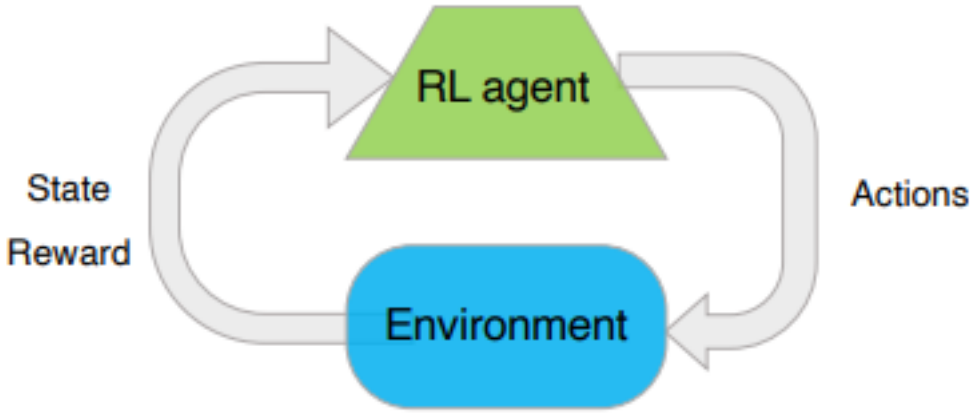
Tramite il pezzo di codice `tuner` vengono cercati gli iperparametri migliori, in vari modi. Possiamo specificare l'obiettivo, una cartella dove memorizzare i risultati, *overwrite* per ripartire ogni volta da 0 la ricerca o meno. Poi avviamo la `tuner.search` sui dataset.

Non possiamo provare ogni combinazione possibile, un approccio è **RandomSearch**, la ricerca è casuale. Più veloce, ma non è detto che individui la miglior combinazione possibile. Alternativa è l'ottimizzazione **bayesiana**, in cui prendo un primo valore a caso, e per gli altri tentativi mi muovo in un intorno di quel primo valore. Esploro di più i valori vicini a valori aventi buoni risultati. Anche qui dipende molto dalla scelta iniziale, e quindi altra ottimizzazione è partire con più valori iniziali, le sviluppa per un po', trova le migliori, e le sviluppa ulteriormente.

1 Introduzione al Reinforcement Learning

Non parliamo più di reti neurali, branca del Machine Learning dedicata al *sequential decision-making*. Dobbiamo quindi prendere decisioni in maniera sequenziale, con lo scopo di ottimizzare

qualche obiettivo.



Nella figura, il protagonista è un *agente*, entità che decide. E' una parte di software. Le decisioni che prende, sono scelte di azioni che può compiere. Tale azioni hanno impatto sull'*ambiente* (resto del mondo che lo circonda) su cui l'agente si trova. L'ambiente ritorna un *reward*/ricompensa, cioè un valore numerico o un vettore, comunque un feedback. La scelta viene fatta in base allo *stato corrente*. Le nuove scelte portano ad un nuovo stato.

L'obiettivo è massimizzare le ricompense nel tempo. Vista in termini di minimizzazione, vogliamo compiere le azioni che costano di meno.

Esempio:

Robot, se si blocca ha un reward negativo, se completa il percorso riceve un reward positivo. Lo *stato*, in questo caso, può essere rappresentato dalle sue coordinate.

Con *reinforcement learning* l'agente, mediante *trial and error* (a tentativi sostanzialmente, *ndr*)

Esempio Tris:

Posso usare *Reinforcement Learning*, in cui:

- *Stato del sistema*: matrice 3x3, cioè se cella è vuota, se c'è una X oppure una O.
- *Azione*: dipende dallo stato, se inizio io ho 9 azioni, dopo 7 (anche l'avversario gioca).
- *Ricompensa*: In questo caso, lo diamo se vinciamo la partita. Nel mezzo, non mettiamo ricompense.

Con tale ricompensa, l'agente impara solo dalla mossa finale, non da quelle nel mezzo. Ma come stabilisco se una mossa è buona? Alla fine è con la sequenza di mossa che posso vincere o meno. L'idea è che, se la mossa x_i è la mossa vincente, allora deve capire che anche x_{i-1} al tempo precedente era una buona mossa, non devo dirglielo io mossa per mossa.

Quindi lo stato deve memorizzare tutto? No, solo lo stato presente. Si gioca in base a ciò che vedo, non mi interessa come ci sono arrivato. Il passato non interessa, anche perchè metterla nello stato porterà, come vedremo, a dei problemi:

- Numero di stati eccessivi.

- Voglio fare la giusta mossa in modo indipendente da come sono arrivato in una certa posizione. (Se ho cavallo in $A3$, la mossa successiva non dipende da come sono arrivato in questa cella).

1.1 AlphaDev

Se addresto agente contro qualcuno che gioca *random*, l'agente non sarà ottimale. Allora viene fatto giocare con se stesso. Quindi gioca Agente_0 contro se stesso, chi vince diventa il nuovo *avversario*. Poi addestro un nuovo Agente_1 contro tale avversario, e se lo batte, allora Agente_1 sarà il nuovo avversario.

AlphaDev è molto recente, non do io il reward (vorrebbe dire conoscere già la soluzione giusta), bensì viene dato solo alla vittoria, in proporzione alla velocità con cui ci è arrivato.

Ovviamente esistono tantissimi altri casi, come veicoli autonomi, trading, auto-scaling nei sistemi cloud,…

1.2 R.L. nel dettaglio

Tutti gli aspetti descritti dipendono dal contesto che analizziamo. Formalmente parliamo di **Processi Decisionali di Markov MDP**, framework per modellare decisioni in situazioni dove le uscite sono parzialmente random.

1.3 MDP

Estensione, a tempo discreto, delle **Catene di Markov**. Ad ogni tempo t associamo lo stato s_t. L'agente sceglie l'azione a_t tra quelle disponibili nello stato s_t. (Ne è un esempio lo spostamento di un robot in un certo stato, magari davanti ho un muro e quindi in quello stato l'opzione di andare avanti è esclusa). Il compimento di a_t porta ad uno stato random s_{t+1}, random perchè in alcuni casi l'output può dipendere da fattori casuali. (Ad esempio, un drone abbassa l'altitudine di x, ma l'output è $x + \Delta$ perchè c'è il vento di mezzo).

Alla fine, l'agente riceve un *reward* (o paga un costo, dipende da come lo vediamo). Ne è un esempio la vittoria a scacchi o il robot che arriva a destinazione.

Graficamente:

Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).

1.3.1 Definizione di MDP

- $\mathcal{S}$: a (finite) set of states
- $\mathcal{A}$: a (finite) set of actions
- p: state transition probabilities

$$p(s'|s,a) = P[S_{t+1} = s'|S_t = s, A_t = a]$$

- r: reward function (or, c: cost function)
  1. $r(s,a) = E[R_t|S_t = s, A_t = a]$
  2. $r(s,a,s') = E[R_t|S_t = s, A_t = a, S_{t+1} = s'] \longrightarrow$ $r(s,a) = \sum_{s'} p(s'|s,a) r(s,a,s')$

- Gli stati sono *quasi sempre* finiti. Non sempre, ad esempio nel caso dell'auto, i numeri associati alla velocità non sono finiti.
- Le azioni sono *quasi sempre* finite, non sempre. Un'auto che sterza, può farlo con un numero

reali di angoli.

- p è la probabilità di entrare in uno stato s', partendo da stato s e compiendo l'azione a.
- Per ultimo, abbiamo la funzione di *reward* o di *costo*. Abbiamo due *varianti*, nella prima siamo in funzione dello stato ed azione attuale, nel secondo dipende anche dallo stato in cui andremo a finire, ma comunque possiamo ritornare sempre alla prima variante.

1.4 Proprietà di Markov

MDP eredita, da Markov, tale proprietà:

"La probabilità di entrare in uno stato successivo non dipende da tutti gli stati precedenti, bensì solo dallo stato attuale."

Questo lo abbiamo visto anche nel *tris* o *scacchi*.

Tuttavia, questa caratteristica può sembrare non sempre *adatta*.

Pensiamo ad un robot che ha reward se spegne luce e torna alla sua base. Il suo stato al tempo corrente, secondo tale proprietà, è ininfluente all'aver spento o meno la luce, poichè conta solo che sia tornato alla base.

Soluzione: sono io che modello il problema, devo ampliare lo stato del robot, aggiungendo variabile *booleana* che mi indica se ha spento la luce o meno. Quindi il reward dipenderà dall'aver spento la luce *e* essere tornato alla base.

Di contro, abbiamo complicato lo stato.

1.5 Formalizzazione dell'obiettivo

Consideriamo un *task episodico* :

- *task* rappresenta l' insieme di *ambiente, agente, stato,...*
- *episodico* rappresenta l'interazione agente ed ambiente ha stato conclusivo, come la fine della partita a tris.

Al tempo t, vogliamo massimizzare il *ritorno atteso* G_t, *atteso* perchè sono stocastici e non deterministici.

$G_t = R_t + R_{t+1} + ... + R_T$

Non tutto ha una fine, pensiamo ai semafori per il traffico, oppure ai bilanciamento in un sistema cloud. La sommatoria divergerebbe, e allora ciò che si fa è introdurre uno sconto $\gamma \in [0, 1)$, che moltiplica tutti i reward futuri. Andando avanti, γ descresce (perchè l'esponente aumenta), portando alla **convergenza**.

Chi è γ?

Un iperparametro.

- Se nullo, è un approccio *greedy*, non penso in prospettiva.
- Se pari ad 1, mi comporto come prima.
- Tipicamente, assume valori vicino ad 1, come 0.999, 0.99.

Altro aspetto è dare maggior peso ai *reward* più vicini nel tempo, ovvero ciò che faccio prima nel tempo, avrà peso maggiore. Privilegio i reward più vicini nel tempo.

$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$

Possiamo vederlo come probabilità che l'agente *muoia* (o che il sistema sia ancora in vita) per la ricompensa.

Meglio guadagnare qualcosa oggi rispetto a farlo domani, perchè domani non potrei essere vivo.

[ ]: