Implementazione di un algoritmo Double Dueling Deep Q-Learning

Michele Proverbio

2019-06-01

1 Introduzione

Il Reinforcement Learning è un'area del Machine Learning che studia l'apprendimento automatico di un task da parte di un agente. Quest'ultimo è immerso in un ambiente che restituisce all'agente un feedback proporzionale all'efficacia delle sue azioni rispetto al task.

Lo scopo del progetto è implementare un algoritmo Double Dueling Deep Q-learning [4] [5] [6] sviluppato da DeepMind [1] per il framework di benchmark Atari. L'algoritmo verrà testato nell'ambiente virtuale *CartPole-v1* di OpenAI Gym [2].

2 Double Dueling Deep Q-learning

2.1 Deep Q-Learning

Il breakthrough del Deep Learning applicato al Reinforcement Learning arriva con la successiva pubblicazione di *Playing Atari with Deep Reinforcement Learning* [4] su *Nature* nel 2013 da DeepMind. Nell'articolo viene introdotto un algoritmo per Reinforcement Learning (RL) che riesce ad imparare policy da un input a grande dimensionalità. Il modello proposto è una rete convoluzionale che riceve in input i render dall'emulatore Atari (RGB 210 x 160 processate in bianco e nero e ridotte a 84 x 84) e restituisce i Q-valori di tutte le possibili azioni in un unico passaggio feed-forward. Uno dei concetti fondamentali proposti è l'approssimazione di una variante della funzione Q: si passa da una funzione che dato uno stato e un'azione restituisce il reward aspettato, ad una funzione che dato uno stato restituisce il reward per ogni azione possibile.

Altro strumento introdotto è il *Replay Buffer*, che spezza concettualmente il ciclo di apprendimento classico del RL. L'esperienza derivante dal feedback dell'azione sull'ambiente non viene usata immediatamente per ottimizzare la funzione di costo, ma viene salvata in un buffer a dimensione fissa. La fase di learning avviene estraendo un mini batch dal replay buffer con distribuzione uniforme. Il principale vantaggio dell'utilizzo di un replay buffer è che viene spezzata la forte correlazione che esiste tra esperienze di stati consecutivi, e porta ultimamente a diminuire la probabilità che il modello abbia un apprendimento divergente.

L'algoritmo viene presentato in pseudo codice nella figura 1.

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

Figure 1: DQN algortihm presentato nell'articolo [4]

2.2 Double Q-learning

Nell'articolo *Deep Reinforcement Learning with Double Q-learning* di Silver et al. viene introdotta una variante dell'algoritmo DQN per attaccare il problema di sovrastimazione della funzione Q. L'algoritmo propone di disaccoppiare la selta greedy dell'azione dalla valutazione dell'azione stessa. Alla rete principale viene affiancata una seconda rete neuronale detta *target network* i cui pesi θ^- vengono aggiornati in base ai pesi θ della rete online in maniera asincrona ogni τ passi.

La funzione di target per il calcolo della loss diventa:

$$y_i^{DDQN} = r + \gamma Q\left(s', \arg\max_{a'} Q\left(s', a'; \theta_i\right); \theta^-\right)$$

2.3 Dueling DDQN

Wang et al. in *Dueling Network Architectures for Deep Reinforcement Learning* [6] propongono una variante al modello convoluzionale usato fino a quel momento. L'intuizione su cui si basa il lavoro è che se l'agente si trova in uno stato svantaggioso non avrà importanza quale azione sceglierà. Questa idea viene concretizzata disaccoppiando l'apprendimento delle coppie $Q(s, a)$ nell'apprendimento di due funzioni separate: una che valuta il valore dello stato in cui si trova l'agente e la seconda che valuta i vantaggi derivanti dalle azioni.

Nell'immagine 2 viene visualizzata la nuova architettura che divide esplicitamente il calcolo del valore dello stato da quello delle azioni.

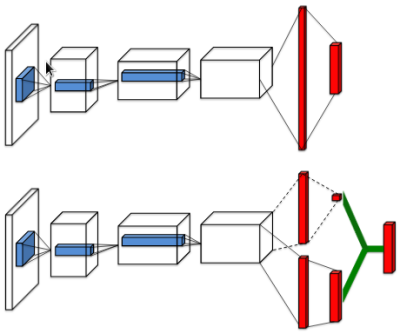

Figure 2: Immagine presa dall'articolo [6]. Viene mostrata la differenza tra l'architettura classica (in alto) con diversi strati convoluzionali e uno strato fully connected e l'architettura proposta (in basso) che sdoppia l'ultimo strato in due flussi: il primo risulta in uno scalare e rappresenta il valore (*value*) dello stato e il secondo risulta nei vantaggi (advantage) per ogni azione

3 Ambiente OpenAI Gym

Come ambiente virtuale di benchmark è stata utilizzata la Gym di OpenAI [2], in particolare è stato scelto il task CartPole-v1 nel quale l'agente controlla con azioni discrete (spinta a sx, spinta a dx, nessuna azione) un carrello con un'asta attaccata ad esso da un perno (come mostrato in figura 3). L'obiettivo dell'agente è mantenere in equilibrio l'asta per più tempo possibile.

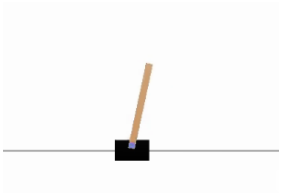

Figure 3: Fermo immagine di una sessione per il task CartPole-v1 di OpenAI Gym

L'ambiente restituisce un reward di +1 all'agente ad ogni timestep e chiude l'episodio se l'asta supera un'inclinazione di 15 gradi o se il carrello si sposta troppo dal centro.

4 Implementazione

4.1 Architettura del modello

L'architettura usata per il progetto è simile alla rete illustrata nell'immagine 2 ma con una rete a 3 strati *fully connected* al posto di quella convoluzionale.

rete fully connected	
input x 256 256 x 128 128 x 64	
value	advantage
64 x 32 32 x 1	64 x 32 32 x *actions*

4.2 Algoritmo

L'algoritmo si divide in una fase di inizializzazione e una fase di training.
Nella fase di inizializzazione:

- viene istanziato il replay buffer e la memoria viene inizializzata in base al parametro `REPLAY_BUFFER_SIZE`
- viene istanziato il modello e vengono caricati i pesi da file se specificato. $\theta = load\_or\_init()$
- viene creata la target network duplicando la rete appena istanziata. i pesi della rete primaria vengono copiati nella target network. $\theta^{-} = \theta$
- viene inizializzata la variabile ϵ a `E_START`

Il ciclo di training procede nel seguente modo:

Algorithm 4.1: DUELING DDQN(*epochs*)

repeat
 $s \leftarrow env.GetState()$
 repeat
  if $random() < \epsilon$
   then $a \leftarrow RandomAction()$
   else $a \leftarrow \arg\max(Q(s;\theta))$
  $s', r \leftarrow env.act(a)$
  insert (s, a, s', r) *into* $ReplayBuffer$
  $s_b, a_b, s'_b, r_b \leftarrow ReplayBuffer.sample(BATCH\_SIZE)$
  targetY $\leftarrow r_b + \gamma Q\left(s'_b, \arg\max_{a'_b} Q\left(s'_b, a'_b; \theta\right); \theta^-\right)$
  loss $\leftarrow MeanSquaredError(targetY - Y_b)$
  ApplyGradients(loss, θ)
  every τ *steps* $\theta^- \leftarrow \theta$
  update ϵ
 until episode_over
until *epochs*

4.3 Tecnologie utilizzate e dettagli implementativi

Tutto il progetto è stato sviluppato in python 3.7.
Per l'implementazione del modello è stato utilizzato tensorflow 2.0 per l'auto differenziazione del gradiente e keras per la definizione e il training della rete neuronale.
I dettagli implementativi più interessanti sono il *submoduling* del modello keras per l'inserimento in maniera trasparente dell'ultimo livello che combina i flussi di value e advantage, e l'uso del *gradient tape* per l'auto differenziazione di quest'ultimo.

Per questo progetto ho deciso di utilizzare una variante del classico decadimento ϵ-greddy dell'esplorazione resettando ϵ di nuovo al valore massimo ogni υ passi (per υ è stato usato un valore di 50000). Inoltre, il decadimento è di tipo esponenziale come si può vedere dalla figura 4. Questa variante dell'esplorazione ϵ-greedy permette all'agente di alternare fasi di micro ottimizzazione a fasi di macro esplorazione. In particolare è stato osservato che anche con un solo reset del ciclo di esplorazione è stato possibile rompere un'abitudine "lazy" dell'agente per migliorare sensibilmente la sua performance; nel caso appena accennato l'agente aveva imparato ad "accontentarsi" di uno stato suboptimale che gli permetteva di raggiungere punteggi molto alti arrivando al limite della sua zona legale poco prima della fine degli episodi. Un reset di ϵ e poche epoche di training sono bastate a raggiungere un livello molto più alto.

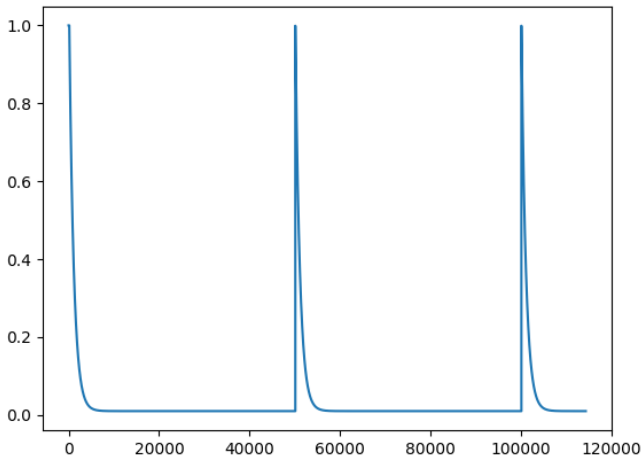

Figure 4: Decadimento esponenziale di ϵ con reset del ciclo

5 Risultati

5.1 Parametri utilizzati

Durante i test sono stati utilizzati i seguenti parametri

Batch size	50
Future discount γ	0.99
Replay buffer size	2000
ϵ start	1
ϵ minimum	0.01
ϵ decay	1000
ϵ reset frequency	50000
Azioni per episodio	1000

5.2 Training e reward totali

In figura 5 è possibile notare il rapido incremento del punteggio ottenuto dall'agente durante le prime fasi. Successivamente è possibile notare i picchi negativi successivi al reset dei cicli di ϵ-esplorazione. Sul sito del progetto [3] è possibile trovare dei video dell'agente addestrato alle prove con il task sotto la cartella *docs*.

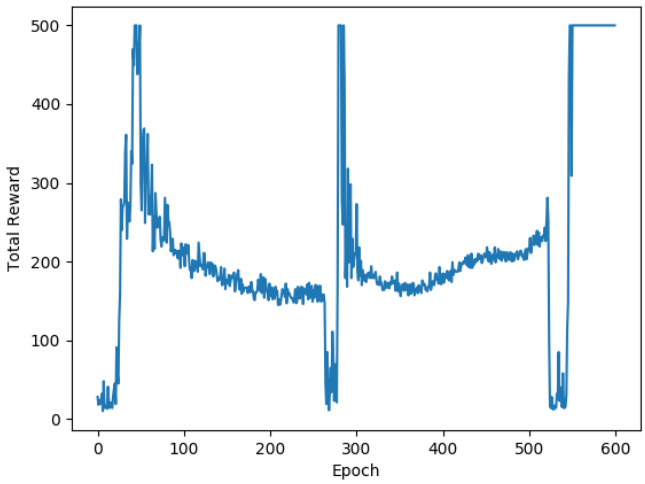

Figure 5: Andamento del reward totale guadagnato dall'agente in fase di training

References

[1] Deepmind. deepmind.com. Accessed: 2019-07-15.

[2] Openai gym. gym.openai.com. Accessed: 2019-07-12.

[3] Project git page. gitlab.com/proch92/sir/tree/master/project/rl/DDQN. Accessed: 2019-07-16.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.