FAI Final Project Report

B10401006 洪愷希

June 15, 2024

Heuristic Algorithm

首先我嘗試的是貪心的 heuristic 演算法。在每次 `declare_action` 時，使用 **Monte Carlo method** 估計目前的勝率，再從 look-up table 決定 action。具體而言，估計勝率的方式是進行 N 次抽樣，每次將對手和尚未掀開的 community card 隨機抽樣，計算勝負並統計總勝利次數。計算出勝負之後以下表選擇行動：

勝率	$[0, 0.4)$	$[0.4, 0.5)$	$[0.5, 0.6)$	$[0.6, 0.7)$	$[0.7, 0.8)$	$[0.8, 0.9)$	$[0.9, 1]$
行動	`fold`	`call`	`raise 10`	`raise 50`	`raise 100`	`raise 200`	`all in`

其中參數為經過搜尋之後所選取。
以上方法相較於 random agent 已經有了顯著的進步，也是在不考慮對手行為下的最佳策略。不過，若能根據對手的行為調整策略，應當能進一步提升則勝率，這在撲克牌術語稱作 **exploit**。我在和 `baseline7` 的對局中觀察到對手似乎也採取了與前述 Monte Carlo method 相似的策略，只有在手牌較好時才會進行 raise，且絕大多數是在 preflop 的小額加注，只有在牌況極好時才會 all-in。對此，我加入了三個簡單的策略：

- 設定參數 $p_{\text{bluff}}, x_{\text{bluff}}$, 使得 agent 在勝率低於 0.5 時，仍有 p_{bluff} 的機率進行 x_{bluff} 的加注。
- 若對手在 preflop 進行加注，並且我方勝率小於一定門檻，則有機率直接棄牌。
- 若對手 all-in，並且我方勝率小於一定門檻，則直接棄牌。

若對手只會依據手牌勝率選擇行動，則以上策略在長期而言應當能夠獲利。
Heuristic agent 的表現顯然並不夠好，不過它有一個重要的目的：產生 neural network 的訓練資料。

Deep Q-Learning Network (DQN)

Basic Algorithm

Q-learning 是一種 reinforcement learning 的方法，它的目標是學習一個 action-value function $Q(s, a)$，其中 s 是狀態，a 是行動，使得在狀態 s 下採取行動 a 的期望報酬最大。Q-learning 的更新規則如下：

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

其中 α 是 learning rate，γ 是 discount factor，r 是獲得的報酬，s' 是下一個狀態。理論上 $s \to_a s'$ 是一個狀態轉移的隨機過程，所以在這裡進行的是一種 sampling。根據 Bellman equation，以上更新式能夠使得 Q 收斂到 optimal Q^*。
對於連續且高維的狀態空間，我們可以使用 neural network 來近似 Q 函數，這就是 **Deep Q-Learning Network (DQN)**。DQN 的目標是學習一個 neural network $Q(s, a; \theta)$，其中 θ 是參數，使得 $Q(s, a; \theta)$ 能夠近似 $Q^*(s, a)$。以 temporal difference (TD) 的角度，我們希望能夠使以下兩者的差距最小：

$$Q(s, a) \approx r + \gamma \max_{a'} Q(s', a'; \theta)$$

因此我們可以定義 loss function 為：

$$L(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right)^2\right]$$

在實作上為了穩定訓練，我們會額外使用一個 **target network** 來計算 $Q(s', a')$，並且以較慢的速度更新 target network 的參數。
另外，因為訓練中所使用的資料僅和當前節點及行動有關，因此 DQN 是一種 off-policy 的方法，也就是我們可以利用不同的策略產生的資料來訓練，這也是使用 heuristic agent 來產生訓練資料的原因。

State Representation

我將狀態轉化為一個 vector s，其中包含了以下資訊：

個數	資訊	備註
1	round count	
4	street	one hot encoding
1	stacks	除以 2000
1	hand strength	以 `HandEvaluator.eval_hand` 計算，除以 2^16
1	win rate	Monte Carlo method 估計
52	community cards	one hot encoding
1	pot amount	除以 2000

因此轉化後的狀態為一個 66 維的向量。這個狀態向量提供 agent 更多的特徵來計算，並且透過 stacks 來衡量對手的狀態。比較不足之處在於這裡並沒有加入同一局遊戲中過去的紀錄，因此無法很好地學習對手的策略。在實作中，狀態會在每次輪到我們行動時由場上資訊計算而得。

Action Space

目前主流的 deep RL 方法中已有許多針對 continuous action space 的方法，例如 DDPG、SAC 等，但是根據我查到的資料，DQN 在 discret action space 上的表現較好。因此我將 raise amount 設定為離散值，從而讓 action space 為離散空間。完整的 action space 如下：

action	amount
`fold`	0
`call`	`call amount`
`raise`	`raise max`
`raise`	10
`raise`	25
`raise`	50
`raise`	100
`raise`	150
`raise`	200
`raise`	250
`raise`	300
`raise`	400
`raise`	500

另一個可能的選擇是以 pot size 的比例做為 raise amount，不過我並未嘗試。

Network Sturcture

我使用三層神經元，以 ReLU 為 activation function，最後一層則為 linear。

Exploration vs. Exploitation

在 RL 中一個很重要的議題是 exploration vs. exploitation，也就是如何在探索新的策略和利用已有的策略之間取得平衡。在 DQN 中，我使用了 ϵ-greedy 的策略，也就是以 ϵ 的機率隨機選擇行動，以 $1-\epsilon$ 的機率選擇 Q 函數估計的最佳行動。在訓練過程中，我將 ϵ 設定為一個指數下降的函數，如下：

$$\epsilon(t) = e^{-t/\lambda}\epsilon_{\text{start}} + (1 - e^{-t/\lambda})\epsilon_{\text{end}}$$

其中 λ 決定衰減的速率。此策略僅適用於訓練，在實際遊戲中 ϵ 應當設定為 0。
作為另外一種 exploration 的方法，我也嘗試了 **Boltzmann exploration**，也就是以 softmax 的方式選擇行動。具體而言，對於 action a，我們選擇的機率為：

$$\frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}$$

其中 T 為參數。

Replay Buffer

前面提及 DQN 是 off-policy 的方法，因此我們可以先利用不同的 agent （如：heuristic）產生訓練資料，而後再進行訓練。為了穩定訓練，我們會使用 **replay buffer** 來存儲過去的狀態、行動、報酬等資訊，並且在訓練時從中隨機抽樣。這樣做的好處是可以減少樣本間的相關性，從而穩定訓練。在訓練的過程中，我們也會將目前 agent 的狀態存入 replay buffer，從而使 replay buffer 中的資料更加多樣化。

Reward

一開始，我將一個 action 所帶來的 reward 定義為該回合最終的獲利。不過嘗試後發現效果不佳，我推測這和 RL 中 **credit assignment** 的議題相關，也就是應如何決定一個 action 在 reward 當中的貢獻。思考後我認為既然未來的獲利會由下一個 state 的 Q 函數決定，那麼 reward 應當僅和當前狀態及下一狀態的差異有關。因此我將 reward 定義為下一狀態的 stacks 減去目前的 stacks。例如，若在 flop 時 `raise 100`，則 reward 為 turn 時的 stacks 減去 flop 時的 stacks，即 100。若在進行動作後回合結束，則 reward 為下一回合 preflop 時的 stacks 減去目前的 stacks。

Improvements

Double DQN

Double DQN 是 DQN 的一個 variant，它的目的是解決 DQN 在估計 Q 函數時的 overestimation 問題。在 DQN 中，我們使用 $Q(s', a'; \theta)$ 來估計 $Q(s', a')$，這將會使得 Q 函數的估計值偏向於 overestimation，因為若一個 action a 的 Q 值較大，那麼在更新 Q 函數時，它將會被選擇的機率也較大，進而使得 Q 值更大。為了解決這個問題，double DQN 將決定 action 的 network 和估計 Q 值的 network 分開。在實作上，可以使用 target network 來估計 Q 值，也就是：

$$Q(s', a'; \theta) \approx r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)$$

Dueling DQN

Dueling DQN 是 DQN 的另一個 variant，它的目的是將 Q 函數分解為 state value 和 action advantage 兩部分。具體而言，我們將 Q 函數定義為：

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

也就是在 network structure 上將 Q 值拆分為了 V（value）和 A（advantage）兩部分，其中 V 表示和狀態較相關而和行動無關的價值，A 則和兩者皆有關，從而使 network 更加靈活。在實作上，我們會將 A 進行 normalization。

Dueling Double DQN (D3QN)

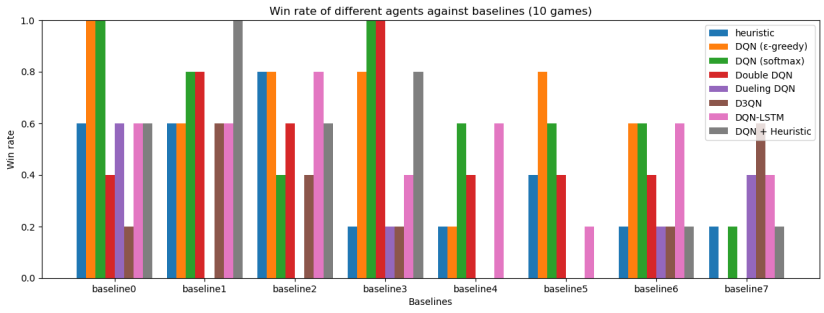
即結合前二者的方法。

DQN with LSTM

前述提及我設計的 state features 並沒有包含雙方行動的歷史，因此較不利於推測對手的行動。然而，從 game theory 的角度而言，二人零和遊戲中對手的行動應當是一個重要的特徵，從而衍伸出 **Nash equilibrium** 及 **game theory optimal (GTO)** 的概念。雖然 DQN 本身與 GTO 並無直接關係，不過我認為將對手的行動納入 state features 應當能夠提升 agent 的表現，因此嘗試用 LSTM 來記憶過去的行動。我將過去 L 個狀態作為 sequence 輸入 LSTM，而後再輸入原先的三層神經元中，並同樣以 Q 函數作為輸出。遺憾的是其效果並不如預期，我推測可能是 hyperparameter 的設定不當，或是 LSTM 並不適合於這個問題。

Comparison

下圖為比較各個作法的勝率，可以看到表現相對好的反而是 vanilla DQN。礙於有的 baseline 跑太久了，能夠玩的局數很少，因此難以判斷這些數據是由於其他方法確實表現較差，或是由於撲克牌的隨機性所造成。整體而言，我認為這個 project 相當程度增進了我對 reinforcement learning 的理解，也讓我對於如何設計 agent 有了更多的想法。不過在 RL 中 domain knowledge 的重要性也是不可忽視的，而我對於德州撲克的了解就相當不足，因此可能在 feature 的設計還有評估上有所不足。



References

- 李宏毅老師課程
- PyTorch turorial