Applied Deep Learning HW3

b05502087 資工系 王竑睿

1 Q1: Models (2%)

1.1 Describe your Policy Gradient & DQN Model

Policy Gradient

- Policy Gradient Model 的 agent 為一個 AgentPG class
- 此 class 內含有 PolicyNet，負責依據環境的 state 輸出所有 action 的機率
- 再利用 torch.distributions.Categorical，依照機率分佈 sample 出 action 與環境互動
- PolicyNet 為一個 online network
  - PolicyNet 架構如下

    x = relu(fully_connected(input_state))

    action_probability = softmax(linear(x))
  - LunarLander-v2 的 state 為 agent 與環境互動時的 8 個物理量組成的 numpy array
    - ⋆ [x座標, y座標, x方向速度, y方向速度, agent 轉動角度, agent 角速度, 左腳有沒有碰到地面, 右腳有沒有碰到地面]
  - 利用兩層 linear layer 得到 logits，再以 softmax 將 logits 映射成所有 action 組成的機率分佈 action_probability
    - ⋆ 由於 LunarLander-v2 有 4 個 action ([不開引擎，左引擎，右引擎，上升引擎])，因此 action_probability 會是 4 個 action 組成的分佈
  - 再依據此機率分佈 action_probability 以 torch.distributions.Categorical sample 出 action 來與環境互動
- 訓練時，每個 epoch 為一個 episode，會讓 agentPG 與環境互動一局直到該局遊戲結束
  - 該局遊戲中的每個 action 和 reward 都會紀錄在 action_list 與 reward_list 內
  - 利用 discount reward 計算每個 action 實際上獲得的 value，與 action 在時間上差愈久的 reward 將會被乘上較多的 discount
    - ⋆ $discount\_reward_t = r_t + \gamma * r_{t+1} + \gamma^2 * r_{t+2} + ...$
  - 以最小化 $-log(action\_probability) * discount\_reward$ 為 loss function 來 update model
    - ⋆ 目的為使得到高 reward 的 action 能有更高的機率被 sample 出來。因此將兩者的乘積加上負號並最小化。
- 採用RMSprop作為optimizer，learning rate為1e-4

DQN

- DQN Model 的 agent 為一個 AgentDQN class
- 此 class 內含有 ReplayBuffer 和兩個 DQN network
- ReplayBuffer 作用如同一個 queue，以先進先出的方式紀錄 agent 與環境互動後得到的 experience
  - 其中，experience 可以用 tuple = (state, action, reward, next_state) 表示
  - 每和環境互動一次 (一個 step) 就加入一筆 experience
  - 如果超過 ReplayBuffer 的容量，則直接以新的 experience 覆蓋舊的
  - 訓練 agent 時則隨機從 ReplayBuffer 中 sample 出一個 batch 的 experience 進行 update
  - 起始時會先和環境互動一段時間不作 update 以累積 ReplayBuffer 內的 experience
- 兩個 DQN network 分別作為 online network 以及 target network
  - DQN network 的架構如下

    x = relu(conv2d(input_state))

    x = relu(conv2d(x))

    x = relu(conv2d(x))

    x = relu(fully_connected(x))

    Q_values = linear(x)
  - 直接以環境每個 frame 整張圖的 numpy array 作為 state。

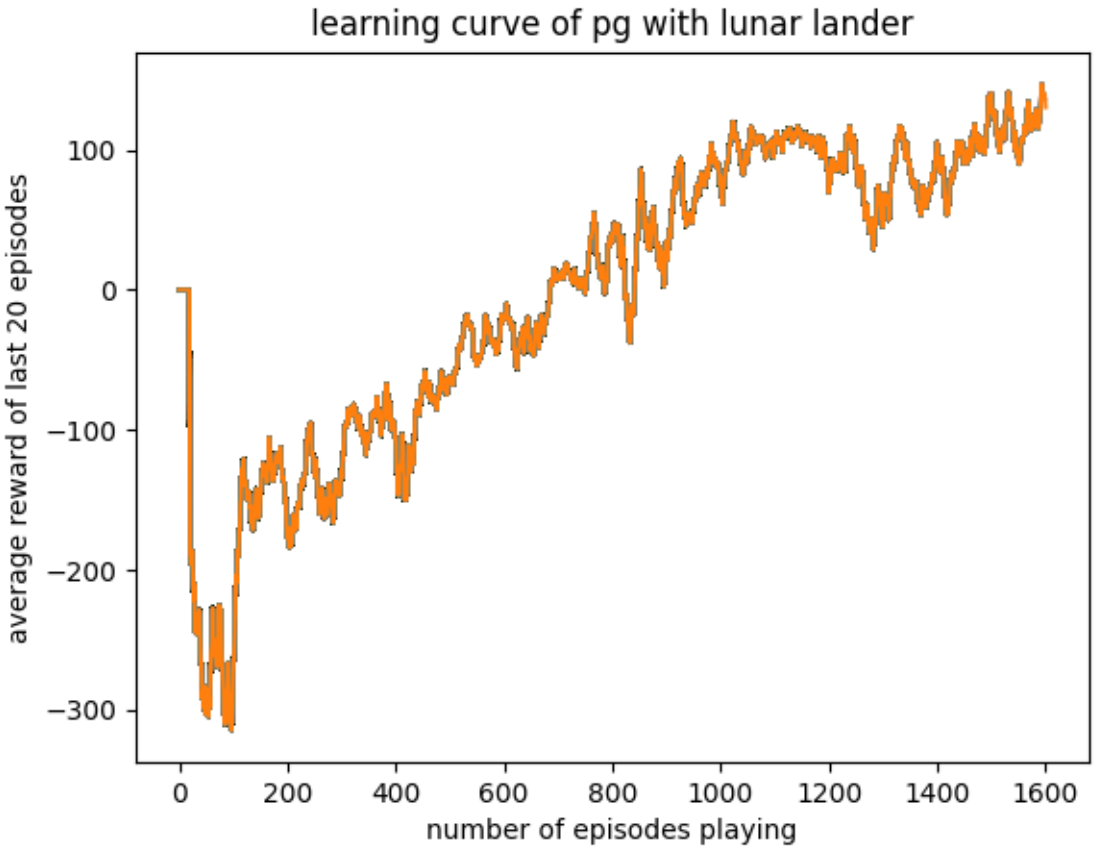
    ⋆ 本次作業 DQN baseline 的遊戲環境使用 MsPacmanNoFrameskip-v0，因此 frame size 為 (210, 160, 3)
  - 利用三層 convolution layer 和一層 fully connected layer 配合 relu activation 來取出 state 的 feature。再透過 linear layer 映射成 Q_values

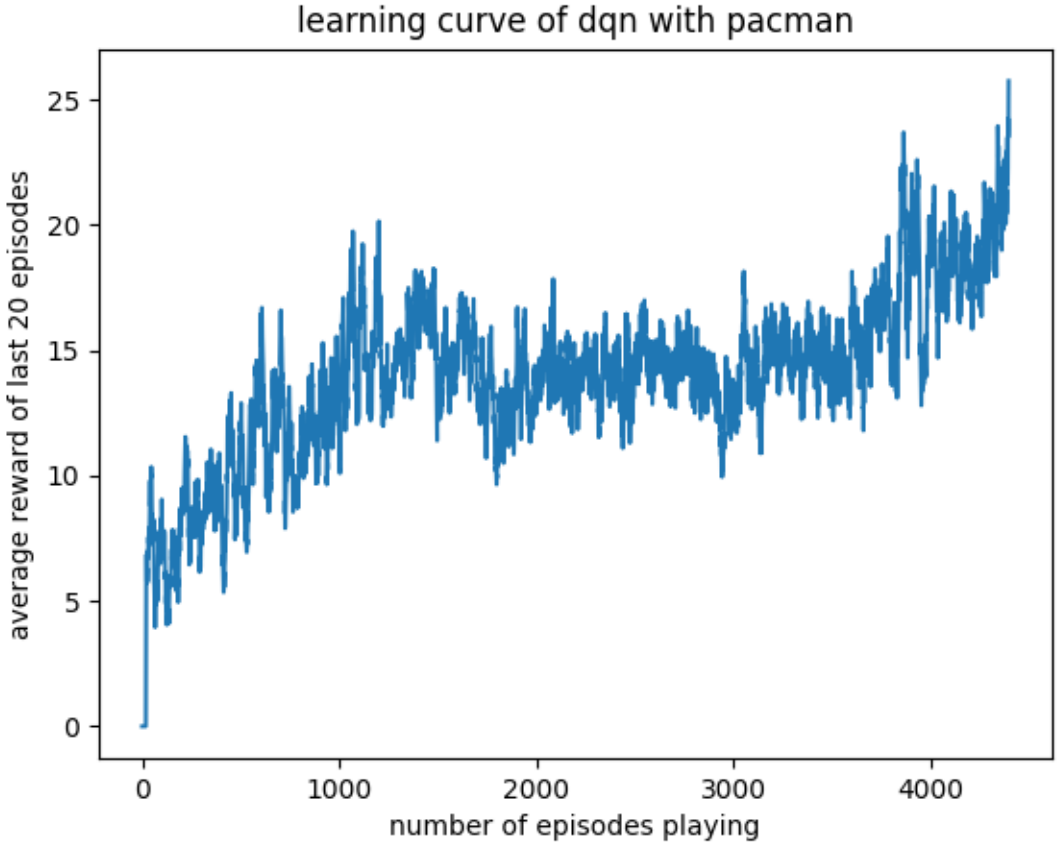
    ⋆ 由於 Action space 共有 9 種 actions，因此會產生 9 個值，代表 9 種動作可取得的 Q-values
- 訓練時，從 ReplayBuffer 中 sample 出一個 batch 的 experience (state, action, reward, next_state)
  - 每和環境互動一次 (每經過一個 step) 就對 model 進行一次 update
  - value_online = online_net(state).max(1)
  - value_target = target_net(next_states).max(1)
  - 利用 Temporal Difference 來定義 loss

    ⋆ loss = MSELOSS(value_online, reward + γ*value_target)

    ⋆ γ 為 discount rate，將每個採取 action 之後得到的 Q_value 依照時間間隔來打折，意即與 action 時間上愈相近的 Q_value 被認為是與此 action 愈相關
  - 其中，當 next_state 到達 done (遊戲已經完成)時，將會把 value_target assign 成 0
  - 每經過一定數量的 step 之後就會以 online_net 將 target_net 賦值
- 採用 RMSprop 作為 optimizer，learning rate 為 1e-4

1.2 Plot the learning curves of rewards



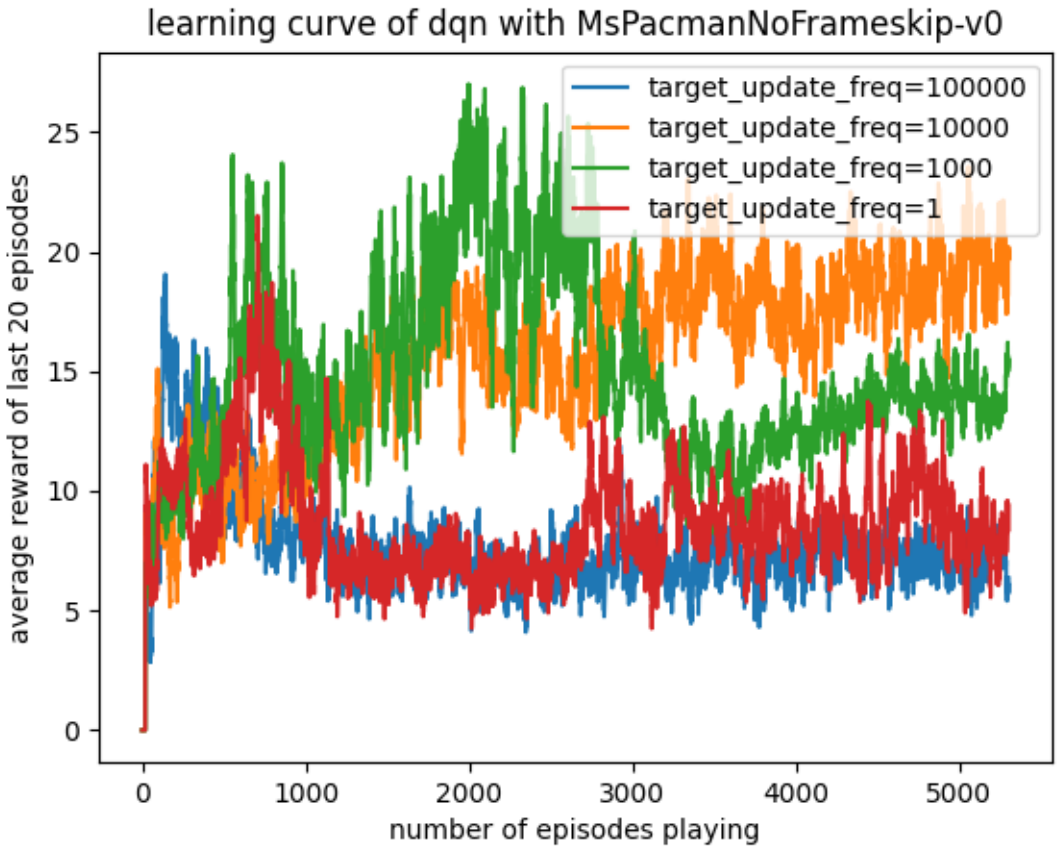


2 Q2: Hyperparameters of DQN (4%)

2.1 Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter

- 本次作業我選擇嘗試另外三種 target network update frequency 的設定

2.2 Plot all four learning curves in the same figure



2.3 Explain why you choose this hyperparameter and how it affect the results

- 因為 target_update_frequency 是 DQN 特有的一個 hyperparameter，因此我選擇觀察調整它，對 agent performance 的影響
- 由圖中可以發現，target_update_frequency 為 1 和 100000 時，agent 的 performance 明顯偏低
  - 這可能是因為 target_update_frequency 太低會使得學習效率太慢難以達到收斂
  - 但 target_update_frequency 太高又會使 agent 在學習時必須持續追逐新的 target 而不穩定，無法取得良好的 reward
- 較為折衷的 target_update_frequency 可以得到較好的結果
  - 可以發現 target_update_frequency 為 10000 時明顯表現較好，與 sample code default 設定 10000 相符

2.4 You can use any environment to show your results

- 本次作業採用 MsPacmanNoFrameskip-v0 作為比較此 hyperparameter 四種 settings 的環境

3 Q3: Improvements of Policy Gradient / DQN (4%)

Choose two improvements of Policy Gradient or DQN

3.1 Describe the improvements and why they can improve the performance

Off-policy learning by importance sampling

- 本次作業我實作於 Policy Gradient 上的 improvement 為 off-policy learning by importance sampling
- 一般的 online policy gradient 主要的問題是每次對於 model 的 update 都只能用 model 當下的 policy π 玩遊戲得到的 trajectory τ 更新 Model
- 如果能使用更早之前的 policy π' 留下的 trajectory τ' 一起進行 training，將可以使 training 時每個 episode 的 data 量累加
  - 這個作法主要的問題是過去的 trajectory τ' 可能與現在的 policy π 有所差異
  - 因此我們會以 importance sampling 來對過去的 data 進行修正，使其 sample 出來計算得到的 loss 可以更貼近現在的 policy π
- importance sampling 的目標是在給定兩個機率分佈 p, q下，以 q sample 出來的結果估計 p 的期望值

$$E_{x\sim p}[f(x)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = E_{x\sim q}[f(x)\frac{p(x)}{q(x)}]$$

  - 利用 $\frac{p(x)}{q(x)}$ 作為修正權重
  - $E_{x\sim p}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$
    - ⋆ 可利用取 sample 的方式，以 sample 結果的平均值逼近期望值
- 使用在本次作業，可以將 policy gradient 的式子重寫成

$$\nabla_\theta J(\theta) = E_{\tau\sim\overline{\pi}_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_\theta(a_{t'}|s_{t'})}{\overline{\pi}_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\right)\right]$$

  - $\overline{\pi}_\theta$ 表示過去的 policy，π_θ 則表示現在的 policy
  - 利用 $\left(\prod_{t'=1}^{t}\frac{\pi_\theta(a_{t'}|s_{t'})}{\overline{\pi}_\theta(a_{t'}|s_{t'})}\right)$ 修正過去的 policy 的 sample

Double DQN

- 本次作業我實作於 DQN 上的優化是 Double DQN (DDQN)
- 一般的 DQN 使用 online network 與 target network。每次都朝向 target network 更新 online network。
  - 原本使用的 target 為 $Y_t^{DQN} = R_{t+1} + \gamma * max_a Q(s_{t+1}, a, \theta_{target\_net})$
  - 然而在 target net 預測出的 Q-value 中挑選最大的來計算 target 可能會 overestimate 實際情形
- 改進的方式為先以 online_net 預測出在 next_state 下會挑選的 action，再以此 action 挑選 target net 中的 Q-value 作為 target
  - $Y_t^{Double\_DQN} = R_{t+1} + \gamma * Q(s_{t+1}, argmax_a Q(s_{t+1}, a, \theta_{online\_net}), \theta_{target\_net})$

3.2 Plot the learning curves and compare results with and without improvement