學號：b06502158 系級：機械三 姓名：陳柏元

1. (20%) Policy Gradient 方法

    1. 請閱讀及跑過範例程式，並試著改進 reward 計算的方式。

    2. 請說明你如何改進 reward 的算法，而不同的算法又如何影響訓練結果?

    我再計算 reward 時，加入了一 discounted reward 遞迴式，計算當前 reward 和進入下一個 state 後所能獲得最大 discounted reward 的和。這邊的 gamma、γ 稱為 discount factor，可以說是對未來 reward 的重視程度。γ 越低，agent 越重視當前所獲得的 reward，並覺得未來獲得的 reward 太遙遠，不足以在當前 state 的決策過程中佔有太大份量。而我將其設為 0.99，希望能對未來預測有較多的想像。計算 discount 方式

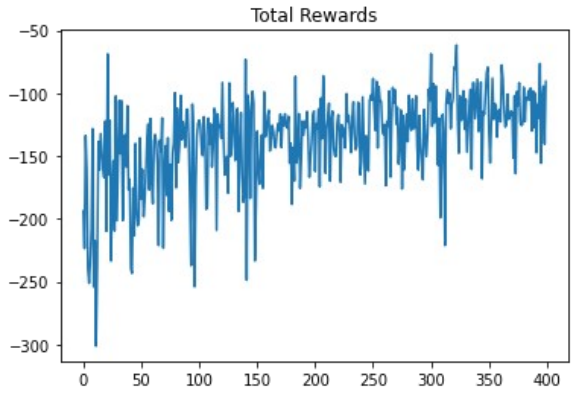
```
    return tensor
def discount(gamma, rewards, dones, bootstrap=0.0):
    #rewards = _reshape_helper(rewards)
    #dones = _reshape_helper(dones)
    # msg = 'dones and rewards must have equal length.
    # assert rewards.size(0) == dones.size(0), msg
    R = torch.zeros_like(rewards) + bootstrap
    discounted = torch.zeros_like(rewards)
    length = discounted.size(0)
    for t in reversed(range(length)):
        R = R * (1.0 - dones[t])
        R = rewards[t] + gamma * R
        discounted[t] += R[0]
    return discounted
```

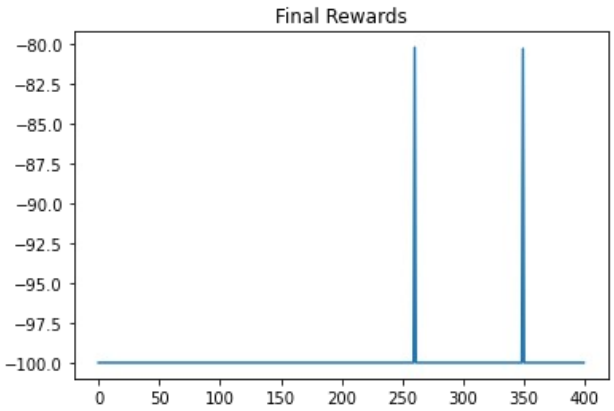
加入方法

```
agent.network.train()  # 訓練前，先確保 network 處在 training 模式
EPISODE_PER_BATCH = 5  # 每蒐集 5 個 episodes 更新一次 agent
NUM_BATCH = 400        # 總共更新 400 次
avg_total_rewards, avg_final_rewards = [], []
prg_bar = tqdm(range(NUM_BATCH))
for batch in prg_bar:
    log_probs, rewards, dones = [], [], []
    total_rewards, final_rewards = [], []
    # 蒐集訓練資料
    for episode in range(EPISODE_PER_BATCH):
        state = env.reset()
        total_reward, total_step = 0, 0
        while True:
            action, log_prob = agent.sample(state)
            next_state, reward, done, _ = env.step(action)
            log_probs.append(log_prob)
            state = next_state
            total_reward += reward
            total_step += 1
            rewards.append(reward)
            dones.append(done)
            if done:
                final_rewards.append(reward)
                total_rewards.append(total_reward)
                # rewards.append(np.full(total_step, total_reward))  # 設定同一個 episode 每個 action 的 reward 都是 total reward
                break
    # 紀錄訓練過程
    avg_total_reward = sum(total_rewards) / len(total_rewards)
    avg_final_reward = sum(final_rewards) / len(final_rewards)
    avg_total_rewards.append(avg_total_reward)
    avg_final_rewards.append(avg_final_reward)
    prg_bar.set_description(f"Total: {avg_total_reward: 4.1f}, Final: {avg_final_reward: 4.1f}")
    # 更新網路
    rewards = torch.Tensor(rewards)
    dones = torch.Tensor(dones)
    rewards = discount(gamma=0.1, rewards=rewards, dones=dones)
    rewards = (rewards.numpy() - np.mean(rewards.numpy())) / (np.std(rewards.numpy()) + 1e-9)  # 將 reward 正規標準化
    agent.learn(torch.stack(log_probs), torch.from_numpy(rewards))
```

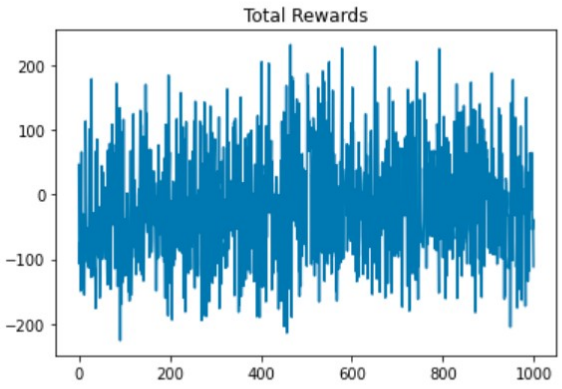
2. (30%) 試著修改與比較至少三項超參數（神經網路大小、一個 batch 中的回合數等），並說明你觀察到什麼。

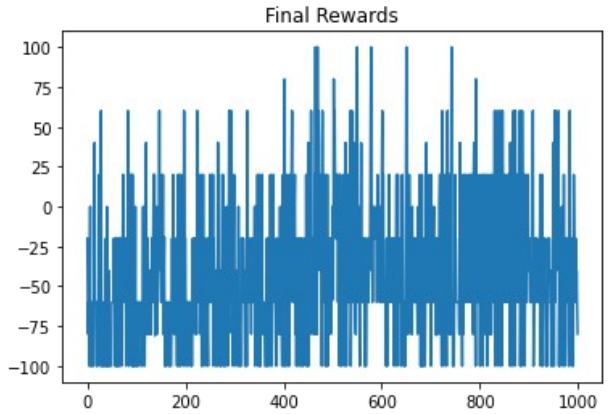
鑑於原本的 final reward 較沒有可觀察性，所以我將 discounted rewards 加入後，再進行三項超參數的比較。原先比照圖：



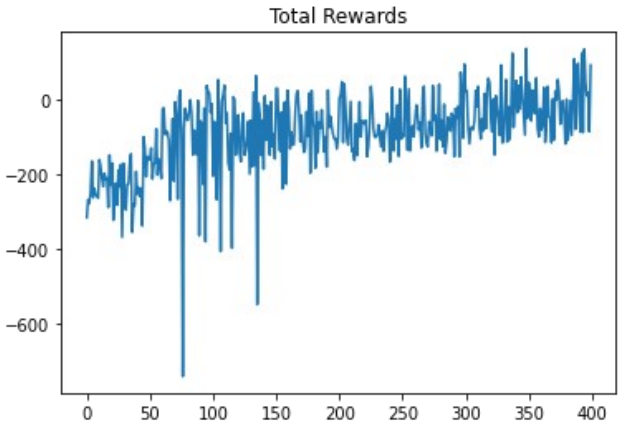


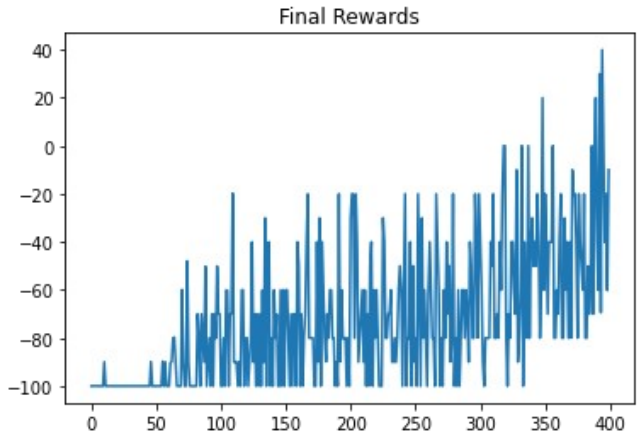
1. 延續上題加入 discounted rewards 以後，我再將 `NUM_BATCH`，也就是總共更新 `400` 次改為更新 `1000` 次，發現他的 `total reward` 一開始有上升，但上升幅度相較 `400` 次，上升幅度在 `400` 次之後，基本上無改變。`final reward` 則是來回振盪，大概 `-50` 分即是其平均值。



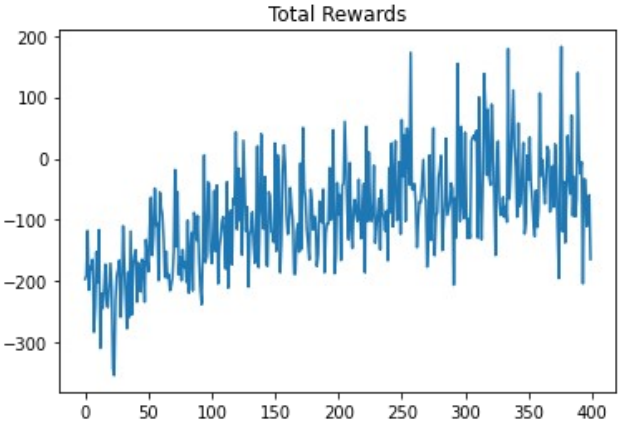


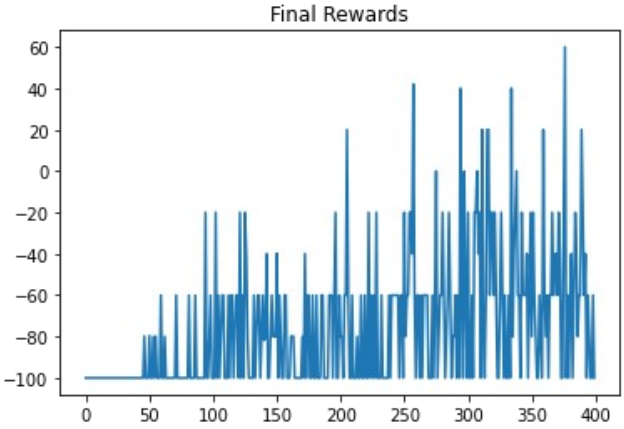
2. 延續上題加入 discounted rewards 以後，我改了 EPISODE_PER_BATCH，從每 5 個 episodes 更新一次 agent，到每 10 個 episodes 更新一次 agent，發現單 NUM_BATCH 更新 total reward 和 final reward 的上升斜率提升，且較慢抵達趨緩的趨勢。final reward 甚至有明顯持續上升的趨勢。





3. 延續上題加入 discounted rewards 以後，我將 optimizer 改成了 Adam，且 lr 設為 0.01，則發現相較於原本，adam 較 SGD 快達到收斂，total reward 和 final reward 的上升幅度較大，且 final reward 較快脫離-100，且穩定上升。

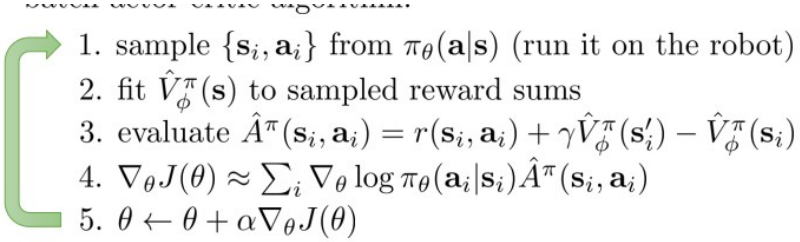


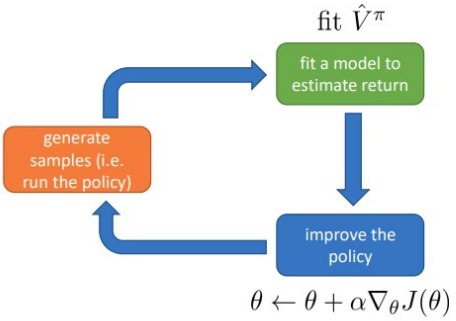


3. (20%) Actor-Critic 方法

請同學們從 REINFORCE with baseline、Q Actor-Critic、A2C 等眾多方法中擇一實作。請說明你的實做與前者（Policy Gradient）的差異。

這裡我實作了 Batch Actor-Critic with discounted rewards 的方法，相較於前者，多了 Critic 這個引導。Actor 是演員的意思，Critic 是評論家的意思，這種算法就是通過引入一種評價機制來解決高方差的問題。具體來說，Critic 就類似於策略評估，去估計動作值函數。Critic：更新動作值函數參數；Actor：以 Critic 所指導的方向更新策略參數。而在這裡我設定的 network 更新次數一樣為 400 次，而每蒐集 5 個 episodes 更新一次 agent，optimizer 用 adam，gamma = 0.99。code 如下





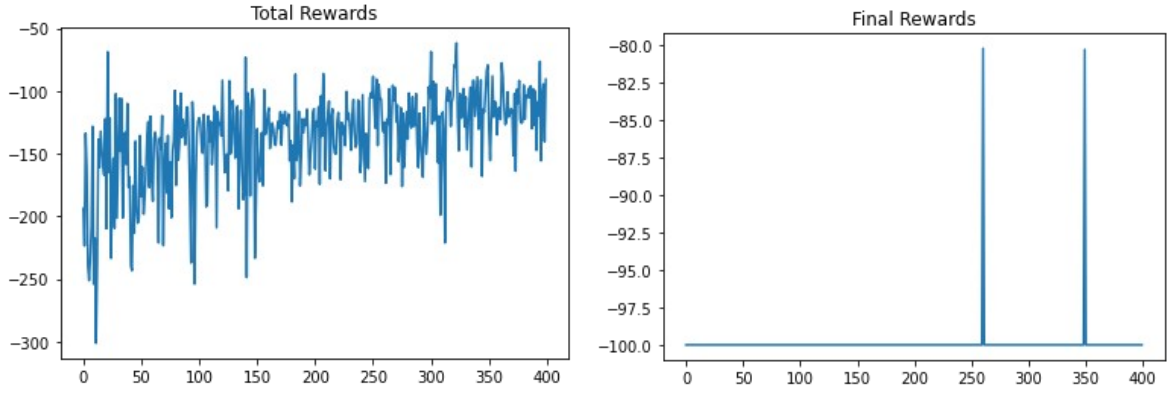

```
def calculateLoss(self, gamma=0.99):

    # calculating discounted rewards:
    rewards = []
    dis_reward = 0
    for reward in self.rewards[::-1]:
    dis_reward = reward + gamma * dis_reward
    rewards.insert(0, dis_reward)
    # normalizing the rewards:
    rewards = torch.tensor(rewards)
    rewards = (rewards - rewards.mean()) / (rewards.std())
    loss = 0
    for logprob, value, reward in zip(self.logprobs, self.state_values,
    rewards):
    advantage = reward - value.item()
    action_loss = -logprob * advantage
    value_loss = F.smooth_l1_loss(value, reward)
    loss += (action_loss + value_loss)
    return loss
```

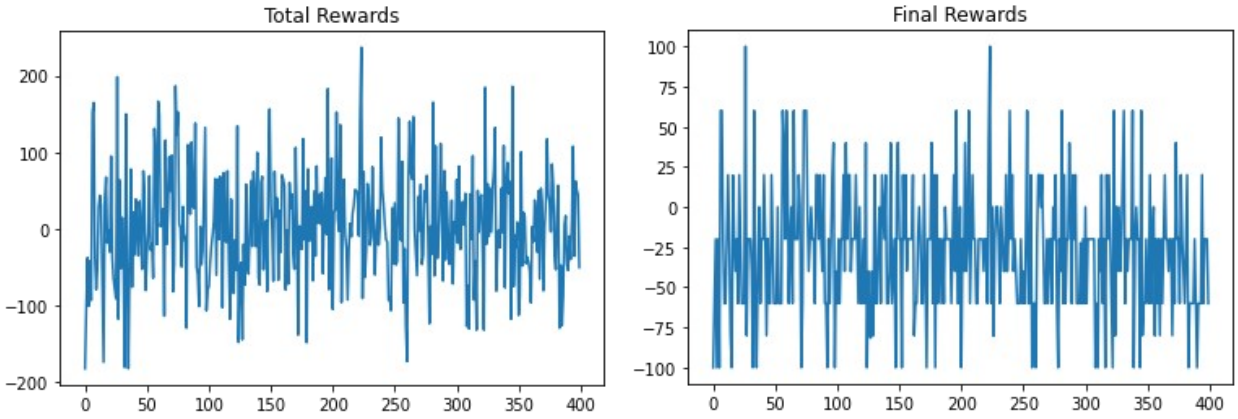
參考了 https://github.com/nikhilbarhate99/Actor-Critic-PyTorch

4. (30%) 具體比較（數據、作圖）以上幾種方法有何差異，也請說明其各自的優缺點為何。

在 Policy Gradient 加入了 discounted reward 後，可以明顯觀察到，如圖，在 total reward 上升了不少，而 final reward 更有顯著的上升，優點如同第 1 題敘述，多了預判能力。



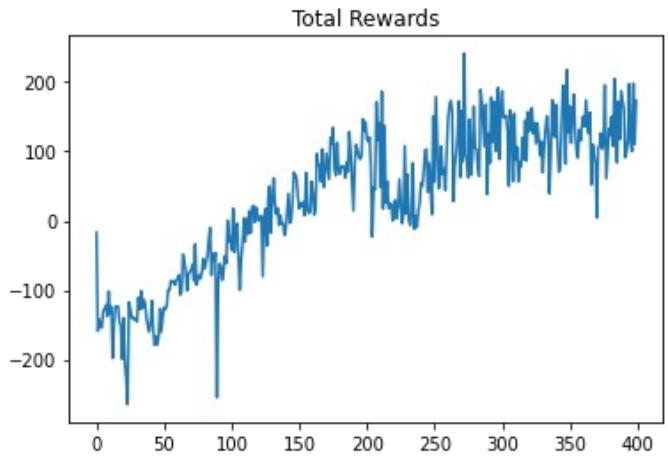
Policy Gradient 未加 discounted reward

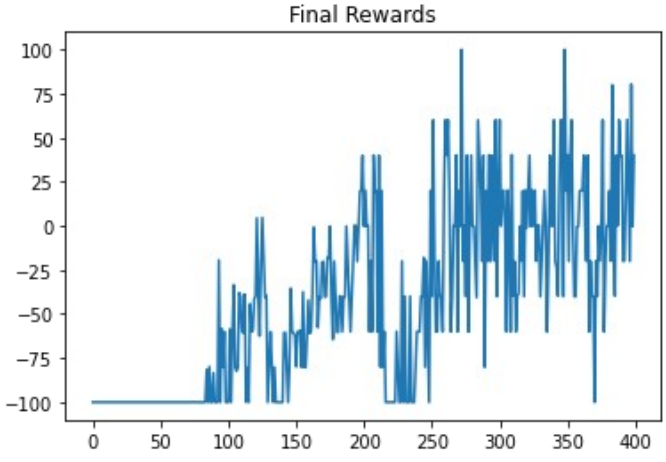


Policy Gradient 加 discounted reward

經測試後發現加入了 Actor-Critic 後，有 lower variance (due to critic)，相較於 Policy Gradient（discounted reward），有 higher variance 。Total Rewards 上下擺盪幅度比 Policy Gradient（discounted reward）小許多，相對於 Policy Gradient 穩定許多。而 Final rewards 則有穩定上升之趨勢，亦較 Policy Gradient（discounted reward）穩定。缺點是若 Critic 不佳，可能導致 bias 非常大，

Policy Gradient（discounted reward）則無 bias 問題。


Total Rewards
200
100
0
−100
−200
0
50
100
150
200
250
300
350
400


Final Rewards
100
75
50
25
0
−25
−50
−75
−100
0
50
100
150
200
250
300
350
400