CE6020 Homework2 Report

學號： 112584001 姓名：陳宣文

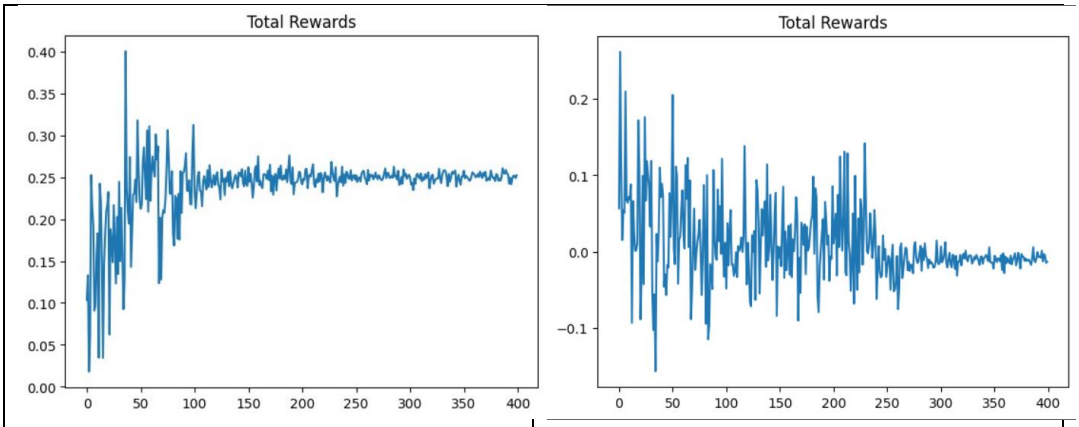
1. (10%) Policy Gradient 方法

   a. 請閱讀及跑過範例程式，並試著改進 reward 計算的方式。

   根據執行的具體動作（買入或賣出），採用了不同的計算方式。當執行賣出（賣出外幣）時，報酬是根據價格變動的比例計算；當執行買入（持有外幣）時，報酬則是價格變動的相反數。

```
if action == self._position.value:
    # 如果動作是持有外幣，則報酬為價格變動的相反數
    step_reward = -(current_price - last_day_price) / last_day
else:
    # 如果動作是賣出外幣，則報酬為價格變動的比例
    step_reward = (current_price - last_day_price) / last_day
```

   b. 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？



範例程式	修改後

兩條曲線的總趨勢是相似的，但在細節上略有不同。第一條曲線的回報在前 50 個步驟中急劇上升，然後在接下來的 150 步驟中保持相對平穩。第二條曲線的回報在前 100 個步驟中上升得更緩慢，然後在接下來的 300 步驟中保持相對平穩。

兩條曲線之間的一個關鍵區別是，第一條曲線在 200 步驟處出現了一個小幅下降。這可能是由於程式在這個步驟中採取了一個不太有效的行動。修改後的程式沒有出現這種下降，這表明它在做出決策時更加穩定。

修改後的程式似乎比原始程式更有效。它在早期階段的回報增長較慢，但它在後期階段的回報保持穩定。這表明它在長期內更有可能取得成功。

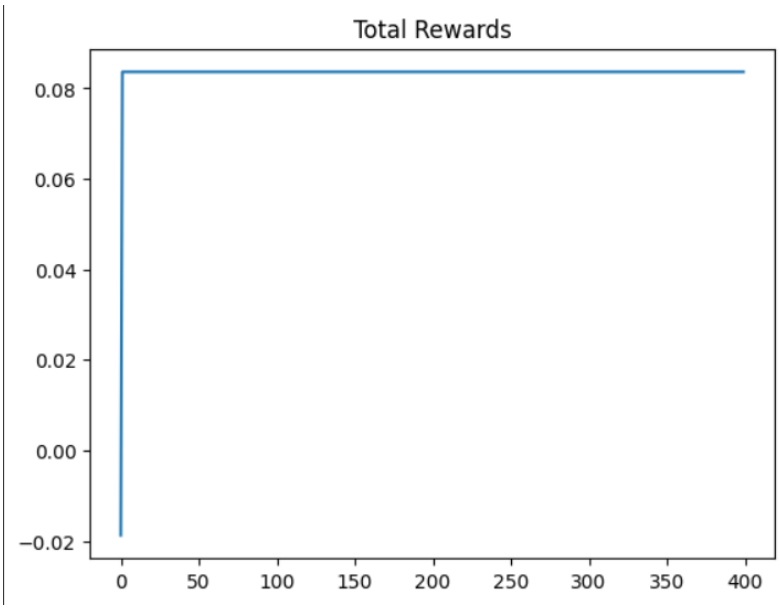
2. (10%) 試著修改與比較至少三項超參數（神經網路大小、一個 batch 中的回合數等），並說明你觀察到什麼。

2.1. 修改神經網路大小：

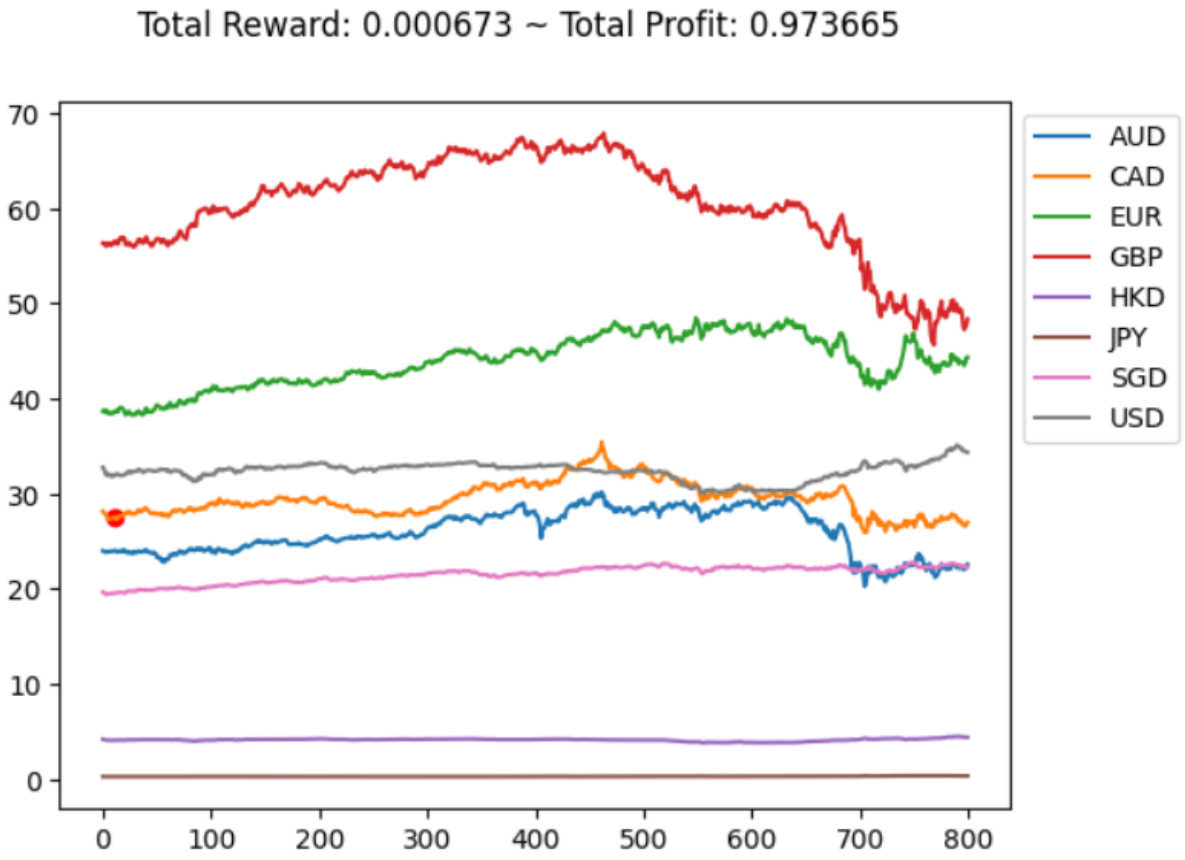
```
class PolicyGradientNetwork(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 9)

    def forward(self, state):
        hid = F.relu(self.fc1(state))
        hid = F.relu(self.fc2(hid))
        hid = F.relu(self.fc3(hid))
        return F.softmax(self.fc4(hid), dim=-1)
```

使用 ReLu 及增加神經網路層數與原本範例城市的差異在於它在最初步驟就已經學會怎麼獲得比較多的 Reward ，也有可能環境太複雜它沒有進行任何買賣(如下圖)。



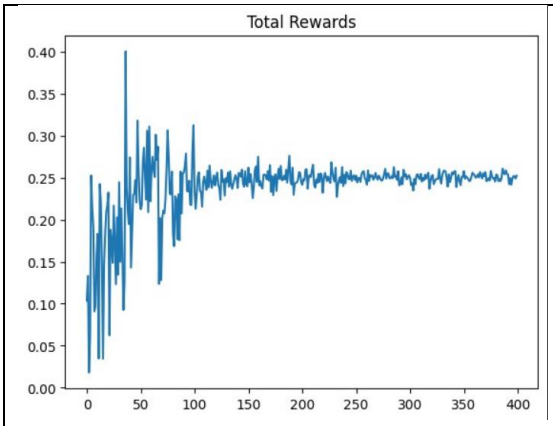
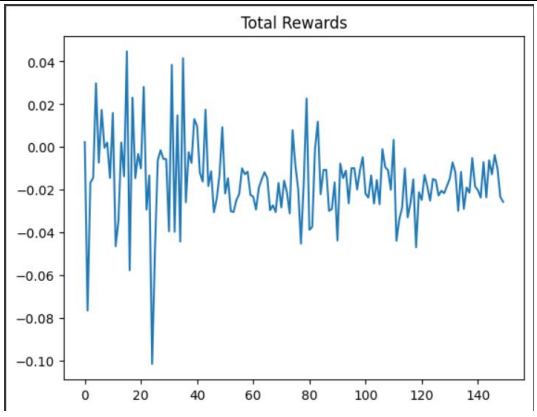
但從第二張圖可以看到 Total Profit (0.97…)卻沒有比原本範例程式(1.17…)高，代表模型還需要調整。但 Score 卻變成了 1.0 比原本範例程式的 0.9762 還要高。



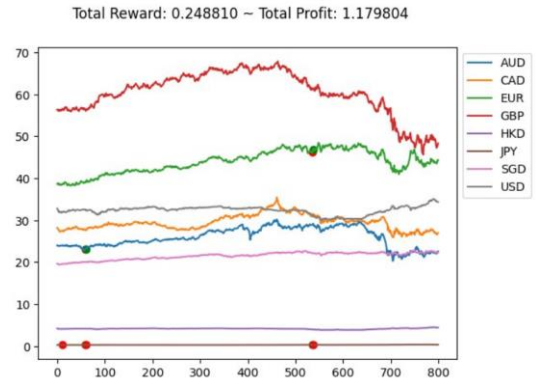
2.2. 修改一個 batch 中的回合數：

修改為

```
EPISODE_PER_BATCH  =  10
NUM_BATCH  =  150
```

	
範例程式	修改 Batch 後

修改 Batch 後的圖顯示 Reward 有逐漸穩定的趨勢，表示他有在慢慢學習如何找到最佳 Reward，或許需要把 NUM_BATCH 數字調高才能知道後面的表現如何。

	
範例程式	修改 Batch 後

從圖中可以看出修改 Batch 後 Total Reward 變成了負值，代表在投資外匯上他的表現極差，也可以從 Total Profit 上看出這個投資沒有很成功，且這個 Batch 設置比範例程式跑出來的結果還要差。最後的 Score 也只有 0.5699 (範例 Score: 0.9762)

3. (10%) 請同學們從 Q Learning、Actor-Critic、PPO、 DDPG、TD3 等眾多 RL 方法中擇一實作，並說明你的實作細節。

使用 Q Learning

```
class DQN(object):

  def __init__(self, network, n_states, batch_size, lr, epsilon, gamma,
target_replace_iter, memory_capacity):

    self.eval_net, self.target_net = network, network

    self.network = network

    self.memory = np.zeros((memory_capacity, n_states * 2 + 2))

    self.optimizer = torch.optim.Adam(self.eval_net.parameters(), lr=lr)

    self.loss_func = nn.MSELoss()

    self.memory_counter = 0

    self.learn_step_counter = 0

    self.epsilon = epsilon
```

```
    self.n_states = n_states
    self.batch_size = batch_size
    self.lr = lr
    self.gamma = gamma
    self.target_replace_iter = target_replace_iter

  def choose_action(self, state):
    x = state.flatten()

    if np.random.uniform() < self.epsilon:
        action = np.random.randint(0, 9)
        log_prob = torch.tensor(0.0)
    else:
        actions_value = self.eval_net(torch.FloatTensor(x))
        m = Categorical(F.softmax(actions_value, dim=0))
        action = m.sample()

        log_prob = m.log_prob(action)

    return action.item(), log_prob



  def store_transition(self, state, action, reward, next_state):

    transition = np.hstack((state, [action, reward], next_state))

    index = self.memory_counter % self.memory_capacity
    self.memory[index, :] = transition
    self.memory_counter += 1
```

```
  def learn(self, log_probs, rewards):


    sample_index = np.random.choice(self.memory.shape[0], self.memory.shape[0])

    b_memory = self.memory[sample_index, :]


    b_state = torch.FloatTensor(b_memory[:, :self.n_states])

    b_action = torch.LongTensor(b_memory[:,
self.n_states:self.n_states+1].astype(int))

    b_reward = torch.FloatTensor(b_memory[:, :self.n_states+1:self.n_states+2])

    b_next_state = torch.FloatTensor(b_memory[:, -self.n_states:])


    q_eval = self.eval_net(b_state).gather(1, b_action)


    q_next = self.target_net(b_next_state).detach()


    q_target = b_reward + self.gamma *
q_next.max(1)[0].view(self.memory.shape[0], 1)


    loss = self.loss_func(q_eval, q_target)




    self.optimizer.zero_grad()

    loss.backward()

    self.optimizer.step()


    self.learn_step_counter +=1


    if self.learn_step_counter % self.target_replace_iter == 0:

      self.target_net.load_state_dict(self.eval_net.state_dict())
```

```
  def sample(self, state):
    state = state.flatten()
    action_prob = self.network(torch.FloatTensor(state))
    action_dist = Categorical(action_prob)
    action = action_dist.sample()

    log_prob = action_dist.log_prob(action)
    return action.item(), log_prob

  def load_ckpt(self, ckpt_path):
    if os.path.exists(ckpt_path):
      checkpoint = torch.load(ckpt_path)
      self.network.load_state_dict(checkpoint['model_state_dict'])
      self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    else:
      print("Checkpoint file not found, use default settings")

  def save_ckpt(self, ckpt_path):
    torch.save({
        'model_state_dict': self.network.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        }, ckpt_path)

n_states = env.shape[0] * env.shape[1]
batch_size = 32
lr = 0.001
epsilon = 0.5
gamma = 0.99
```

```
target_replace_iter = 100
memory_capacity = 10000


network = PolicyGradientNetwork(n_states)
agent = DQN(network, n_states, batch_size, lr, epsilon, gamma, target_replace_iter, memory_capacity)
```

以上是深度強化學習 (Deep Q Network, DQN) 模型。以下是對程式碼中主要元件的解釋：

1. 初始化（__init__) 函數：

network: 代表神經網路模型，這個模型是由 PolicyGradientNetwork 類別實例化的。

n_states: 表示環境的狀態空間大小。

batch_size: 每次從記憶中抽樣的資料批次大小。

lr: 學習速率。

epsilon: epsilon-greedy 策略中的 epsilon 參數。

gamma: 折扣因子。

target_replace_iter: 每隔多少步更新一次目標網路。

memory_capacity: 記憶體緩衝區的容量。

2.choose_action 函數：

接受當前狀態，使用 epsilon-greedy 策略選擇動作。當隨機數小於 epsilon 時，隨機選擇動作；否則，使用現有模型的預測選擇動作。

返回選擇的動作及對應的對數機率（用於 Policy Gradient 方法中）。

3.store_transition 函數：

將當前狀態、動作、獎勵和下一個狀態組成的轉移儲存在記憶體中。

4.learn 函數：

從記憶體中抽樣資料。

計算 Q 值的目標和預測值，並使用均方誤差（MSE）作為損失函數。

使用優化器（Adam）進行梯度下降。

定期更新目標網路的權重。

5.sample 函數：

採樣一個動作，同樣使用神經網路模型的預測，並返回動作及其對應的對數機率。

6.load_ckpt 函數：

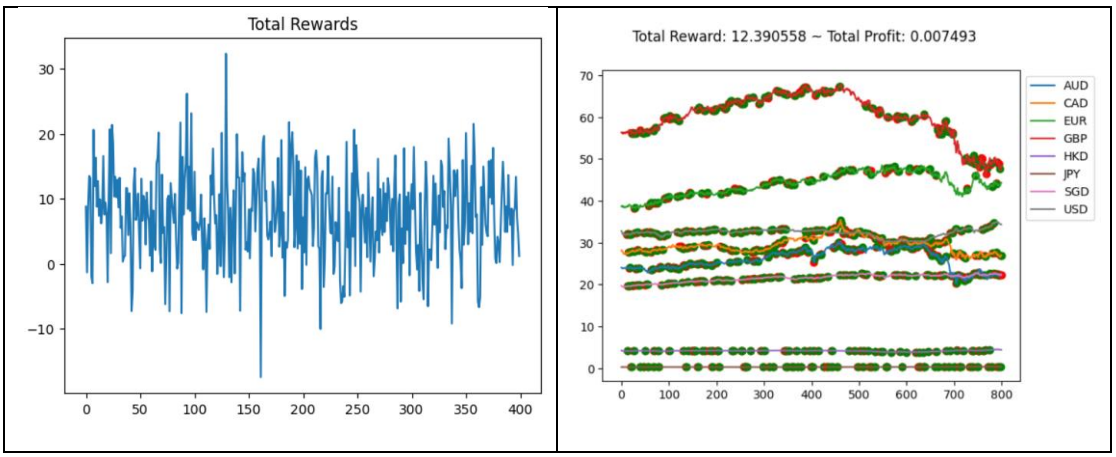
從指定路徑載入模型的權重及優化器的狀態，用於模型的恢復。

7.save_ckpt 函數：

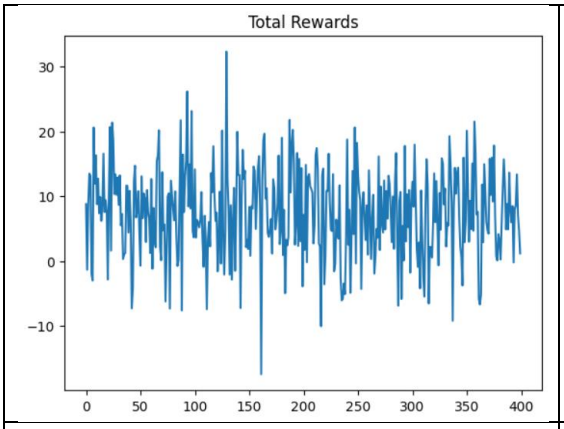
將模型的權重及優化器的狀態保存到指定路徑，用於模型的備份。

在程式碼的最後，創建了一個 DQN 類別的實例 agent，並初始化了相關的環境參數。這個 agent 就可以在強化學習的環境中進行訓練。

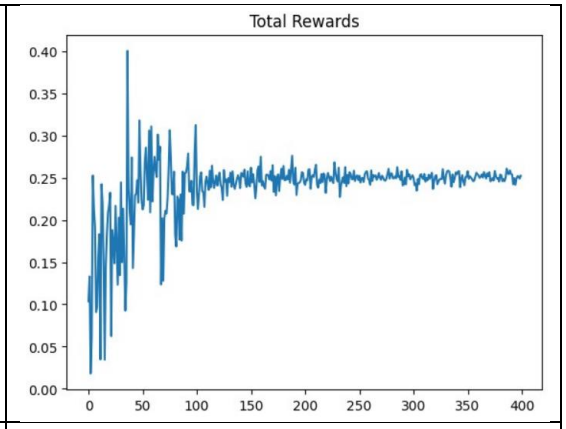
使用 DQN 在 Total Reward 上看起來起伏很大，代表他沒抓到如何取得更加的 Reward，且最後的 score 也只有 0.209 而已。



4. (10%) 請具體比較（數據、作圖等）你實作的方法與 Policy Gradient 方法有何差異，並說明其各自的優缺點為何。



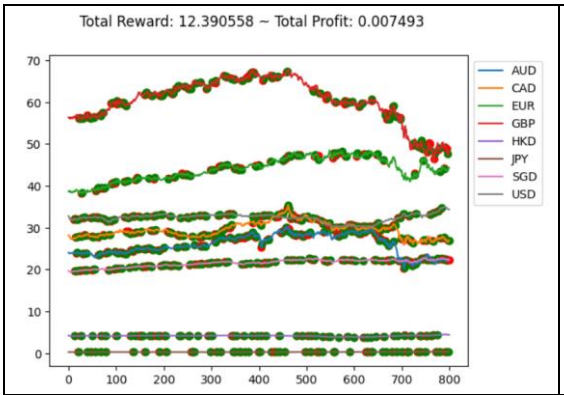
DQN



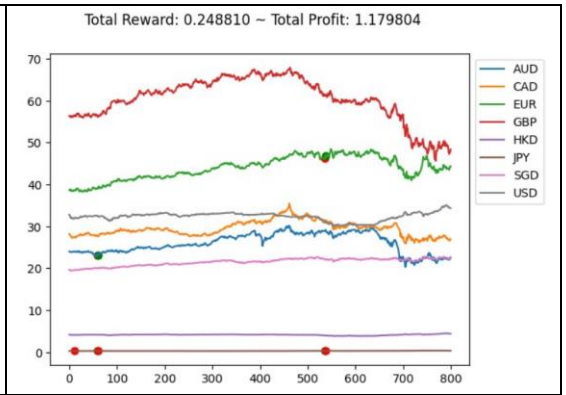
Policy Gradient

DQN 的回報在前幾百步中更高，代表 DQN 能夠更快地學習基本策略，但 Policy Gradient 的回報在最後幾百步中更高，表示 Policy Gradient 能夠在學習到基本策略後繼續改進性能。

DQN 的回報曲線更平滑，代表 DQN 的學習過程更穩定，Policy Gradient 的回報曲線更陡峭，這表示 Policy Gradient 的學習過程更不穩定。



DQN



Policy Gradient

DQN 適用於離散動作空間且穩定性較高的情境，而 Policy Gradient 更適合連續動作空間，但可能需要更多的調參和訓練時間。選擇哪種方法取決於問題的性質，並且有時候混合使用這兩種方法，以發揮它們各自的優勢。