CE6020 Homework2 Report

學號：110201532 姓名：范植緯

1. (10%) Policy Gradient 方法

看完範例code後最主要發現reward的問題是在於沒有把手續費這一部分算上去，單純只看兩天之間的利率就互換，會導致換的次數較為頻繁，讓手續費變多，整體profit下降，因此我修改reward的方法是將reward設為 將(經過動作後新的淨利/還沒經過動作的淨利)取對數，會要取log的原因是因為當在動作後的錢沒有比還沒動作的錢還多的話，代表作這個動作是會降低目前的總profit的，故為負，如果有讓整體profit上升，就為正

而在設計上也有一些改變，首先，在step()方面我有發現到範例code上面是先做

_calculate_reward()再_update_profit()，當我在_calculate_reward呼叫total_profit時應是還沒做動作的total_profit，故在我自己設計的code中step的順序是先_update_profit，再呼叫_calculate_reward，而在_calculate_reward中，我有預設一個global變數專門儲存”還沒動作前的total_profit”，這樣就可以得出這個方法，並在其他超參數皆相同的情況下，我發現這個新的reward方法比範例code得到的score還要高

以下圖為我如何實際操作和得到的score。

實際操作:

```
  def step(self, action):
      self._action = action
      self._truncated = False
      self._current_tick += 1

      if self._current_tick == self._end_tick:
          self._truncated = True
      self._update_profit(action)
      step_reward = self._calculate_reward(action)
```

```
def my_calculate_reward(self, action):
    step_reward=np.log(self.get_total_profit()/self.global_previous_total_profit)
    self.global_previous_total_profit=self.get_total_profit()

    return step_reward
```

得出來的分數:上面是範例CODE SCORE，下面是修改後的SCORE

```
Score: 0.987281976744186
```

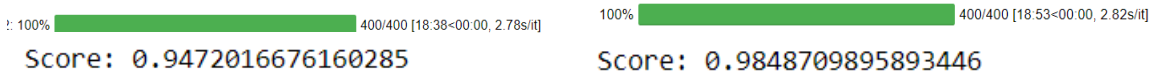
```
Score: 0.9978797516280482
```

然而這種做法容易讓model認為完全不要轉換是比較好的profit，所以其實效果也沒有很好，我也使用過讓範例code的reward再加上(sell_price-buy_price)/buyprice，想說這樣可以把手續費跟轉換匯率兩者相加，然而這樣的效果卻也沒有到特別好。

2. (15%) 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。

修改神經網路大小:當我只修改了神經網路大小(更改神經元的數量)時，我發現雖然影響的時間沒多很多，但整體的score差距卻非常大，我想這樣的原因是因為雖然較小得神經網路比較好訓練，但也因此無法抓出複雜一點的模型還有特徵，以這題來說我將神經元數調到32後就可以達到更好的score

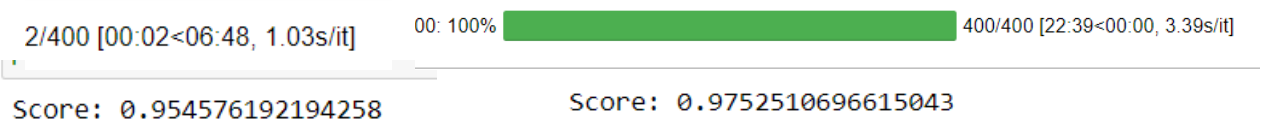
左下角為每層所跑的神經元數量為8的時候，右下角為每層的神經元為32的時候



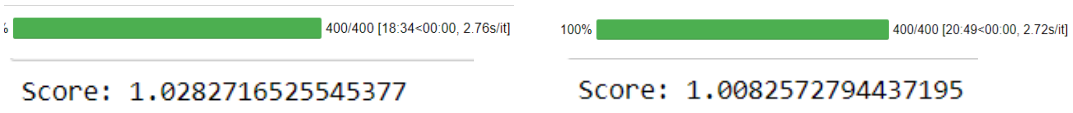
但如果我們把神經元數條太高反而會讓score下降，我想這是因為可能與資料太過擬和，而讓在testing data上面效果不太好，以下的圖為將神經元調到64後所花的時間及分數，可發現分數卻比在32時還要低。

修改batch回合數:當我只修改了一個batch的回合數，在其他因子都沒改變的情況下，如果batch數越小分數會較低，但執行速度會快了很多，batch數越高，分數則會越高，我認為會有這樣的結果主要是因為在我們更新模型時所使用的數據量多寡的差異，如果增加回合數將導致整體數據更多，可能可以找到更好的模型，但也因為數據量增加，會導致cost(時間)便很長。

左下圖為我將一個batch的回合數設為2時所跑出來的分數和所花費的時間，右下圖則是我將一個batch的回合數設為7時所跑出來的分數和所花費的時間。



修改learning rate:當我只修改learning rate，將整體的learning rate 從0.001調整到0.01，發現整體的score可以上升到1.028，然而在繼續調整learning rate到0.03時，卻發現整體的score下降了，我認為這是因為learning rate一次太大，導致policy gradient無法收斂到極小值的緣故。左下角圖為lr為0.01的時間以及score，右下角圖為lr=0.03的時間以及score。



3. (15%) 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。

在做這份作業時，我剛開始先使用了助教給的範例DQN model，藉由隨機取出樣本，並找出當前樣本當下的eval_network，也計算了對next_state當下target Q，藉由兩者算出loss，最後optimize。

```
class DQN(object):
    def __init__(self,network, n_states, batch_size=32, lr=0.001, epsilon=0.1, gamma=0.01, target_replace_iter=1000, memory_capac
        self.eval_net, self.target_net=network,network
        self.network = network
        self.memory=np.zeros((memory_capacity,n_states*2+2))#每個memory中的experience大小為(state+action+reward+next_state)
        self.optimizer = torch.optim.Adam(q_network.parameters(), lr=lr)#創建優化器，利用梯度下降來優化參數
        self.loss_func=nn.MSELoss()#均方根損失函數
        self.memory_counter=0
        self.learn_step_counter=0 #計算當前累積幾次了
        self.epsilon = epsilon
        self.memory_capacity=memory_capacity
        self.n_states=n_states
        self.gamma=gamma
        self.target_replace_iter=target_replace_iter#target network多久要更新一次(不能常常更新)
        self.batch_size=batch_size
    def learn(self):

        sample_index=np.random.choice(self.memory_capacity,self.batch_size,replace=False)#隨機抽取樣本，並且打散
        b_memory=self.memory[sample_index, : ]#把抽取的樣本經驗放入b_memory
        b_state=torch.FloatTensor(b_memory[:,:self.n_states])#取出有關state的資料
        b_action=torch.LongTensor(b_memory[:,self.n_states:self.n_states+1])#取出有關action的資料
        b_reward=torch.LongTensor(b_memory[:,self.n_states+1:self.n_states+2])#取出有關reward的資料
        b_next_state=torch.FloatTensor(b_memory[:,-self.n_states:])#取出有關next_state的資料
        #計算現有eval net和target net得出的Q value落差
        q_eval = self.eval_net(b_state).gather(1,b_action)#重新計算這些experience當下eval_network的Q
        q_next=self.target_net(b_next_state).detach()#對Q下一個狀態的估計
        q_target=b_reward+self.gamma*q_next.max(1)[0].view(self.batch_size,1)#計算這些experience當下target_net的Q
        loss=self.loss_func(q_eval,q_target)#找差距

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

然而在整個實際跑過一次後發現效果一直不太好，score會很低，我認為這可能的原因在於我設計的reward不太好(無論是我第一題更改的新方法，又或是範例code給的方法)，導致整個model沒朝我要的方向前進，因此我又設計了較為單純的Q-learning，以下為我實作的細節，最主要的差距在於Agent的方法，

```python
class QLearningAgent():
    def __init__(self, q_network):
        self.q_network = q_network
        self.optimizer = torch.optim.SGD(q_network.parameters(), lr=0.0001)

    def learn(self, state, action, reward, next_state, done):
        # 計算 Q-value 的目標值
        gamma=0.95
        q_value_target = reward + (1 - done) * gamma * torch.max(self.q_network(next_state))

        # 計算 Q-value 的預測值
        q_value_pred = self.q_network(state)[action]

        # 計算均方誤差損失
        loss = F.mse_loss(q_value_pred, q_value_target)

        # 更新網路
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def choose_action(self, state, epsilon):
        # 以 epsilon-greedy 方式選擇動作
        if np.random.rand() < epsilon:
            return np.random.choice(env.action_space.n)
        else:
            # 根據 Q-value 選擇最佳動作
            with torch.no_grad():
                q_values = self.q_network(state)
            return torch.argmax(q_values).item()
```

在初始化的部分，q_network是由神經網路來做的，因為如果用原本的Q-table來做，可能會導致太多的state，所以我是用神經網路去逼近，

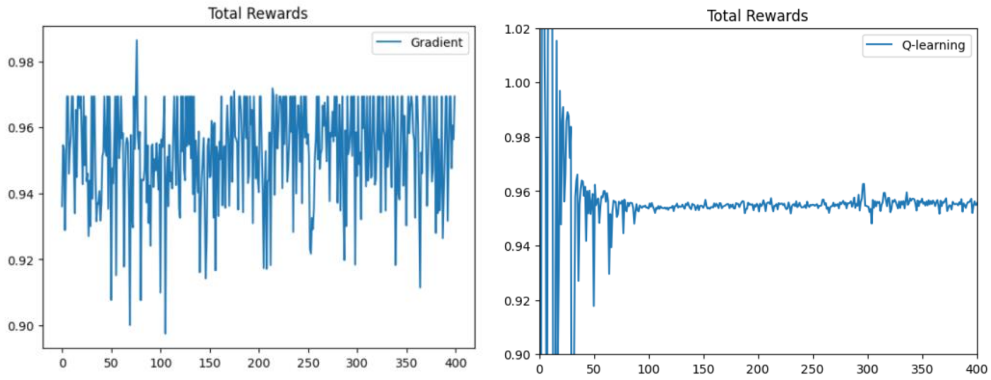
而在learn方面，q_value_target是我們當前的目標Q值，q_value_pred是要去用當前的state和動作去神經網路查預測值，最後去看差多少，藉由梯度下降調整神經網路的參數。

而在choose_action的方面，也是跟助教給的DQN範例差不多，如果小於epsilon，就去看可不可以學到更好的經驗，如果是大於epsilon，則選擇目前最好的動作，比較特別的是我在主程式中epsilon不是固定的，下方為我在train時用的程式碼，epsilon會因為跑的次數越多而越來小，這是因為在剛開始train時，我們應該是要多先去exploration，而在後面快結束時，反而最好是找當前最好的方法來做才能提升整體的score，這是和exploitation的trade-off，其餘迴圈內的最主要就是做資料處理，然後丟給agent去learn

```
total_episodes = 800
epsilon_start = 1.0
epsilon_final = 0.1
epsilon_decay_steps = 800
history_total_rewards=[]
CHECKPOINT_PATH = './model.ckpt' # agent model 儲存位置
# 訓練循環
epsilon = epsilon_start
epsilon_decay = (epsilon_start - epsilon_final) / epsilon_decay_steps
agent.q_network.train()  # 訓練前，先確保 network 處在 training 模式

for episode in range(total_episodes):
     # 重置環境並獲取初始狀態

    state = torch.FloatTensor(env.reset()[0]).reshape(-1)
    total_reward = 0

    while True:
        action = agent.choose_action(state, epsilon)
        next_state, reward, done, _ = env.step(action)
        #return observation, step_reward, self._truncated, info
        # 將元組中的所有元素轉換為 NumPy 陣列，同樣檢查並轉換維度
        next_state_elements = [np.array(item) for item in next_state]
        for i in range(len(next_state_elements)):
            if next_state_elements[i].ndim == 0:
                next_state_elements[i] = np.array([next_state_elements[i]])

        next_state = torch.FloatTensor(np.concatenate(next_state_elements))
        agent.learn(state, action, reward, next_state, done)

        total_reward += reward
        state = next_state

        if done:
            break

    # 更新 epsilon
    epsilon = max(epsilon_final, epsilon - epsilon_decay)
    history_total_rewards.append(total_reward)
    print(f"Episode {episode + 1}, Total Reward: {total_reward}")
```

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。

以下為policy gradien在學習過程中更改數值的圖表，和在Q-learning時學習曲線的圖表(其中我是以Q-value來作為是否有在學習的指標，並且因為gradien是以total_profit為依據，故會以1為目標，所以我最後為了讓兩圖表叫好比對，在Q-learning的圖表中，每個數值都有加一，這樣在Y軸才差不多)，可以從兩張圖片明顯發現，policy gradient的作法在**穩定性方面**比Q-learning還要差，且在**收斂方面**也是Q-learning比較快。
而以理論上來看，由於我們的Q-learning是基於epsilon-greedy的策略進行探索，而在policy-gradient反而是較為單純的以機率進行learning，因此兩者在學習效率和最終的性能本身就有差別。



整體在實做這份作業的過程中，認為Q-learning和policy-gradient有以下優缺點:
Q-learning:
優點:**收斂性較佳**，若在離散行為的問題上會較policy gradient還要好，也較**好解釋**
缺點:**對高維度空間較不友好**，如果今天state太多，維度整體過高，那用table的做法就不太可行了，這也是為什麼在這題我使用的是神經網路來做
policy gradient:
優點:**較適用於連續得動作**
缺點:**穩定性較差、收斂速度相對較慢**