1. (10%) Policy Gradient 方法　　薛竣祐 111526009

a. **請閱讀及跑過範例程式，並試著改進 reward 計算的方式。**

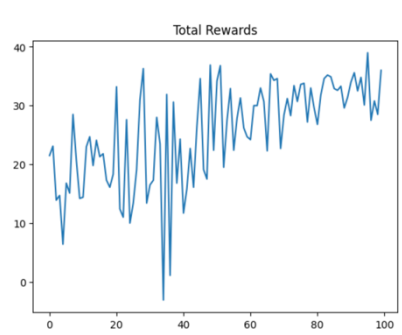
原本的範例程式使用的reward是直接將total_reward設定為整個episode的reward，但事實上儘管結果是好的，在episode裡有可能會有會導致reward下降的動作，因此不能讓全部都套用total_reward。

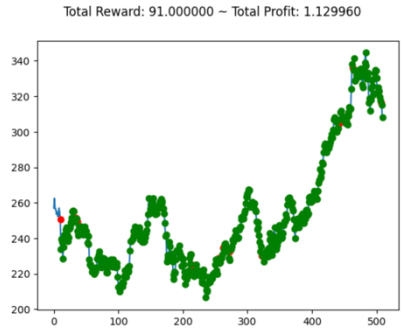
b. **請說明你如何改進 reward 的算法，而不同的算法又如何影響訓練結果？**

我用了幾個方式嘗試改進reward的計算方式，最根本的是記錄完整episode的各個reward，最後像原本的作法與total_reward相加。並在過程中使用以下策略可使測試reward增加：

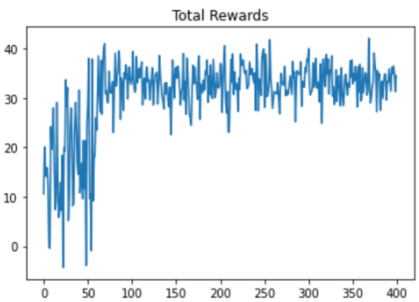
- 如果當下reward<0，把reward再減10。此目的是加重模型的懲罰，相當於買進股票後，盡量漲了才賣。
- 紀錄最近n次的action，如果最近n次的action都一樣，將當下reward減10並清除紀錄。此目的相當於鼓勵模型多做操作，避免因為前一項的reward更改導致模型買了股票後一直到結束都不賣出。

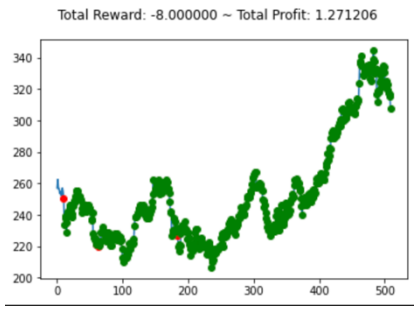
更改後的訓練結果如下（左為訓練total_rewards，右為測試reward）：





範例程式訓練結果如下（左為訓練total_rewards，右為測試reward）：



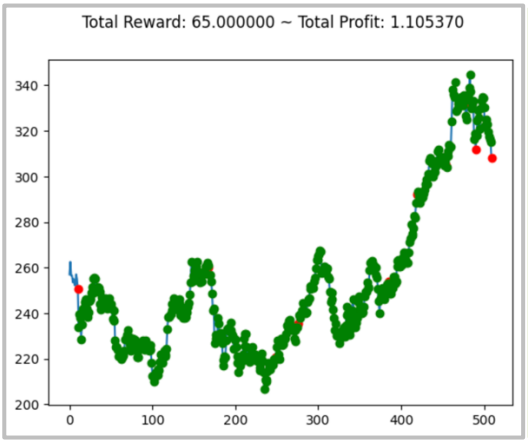


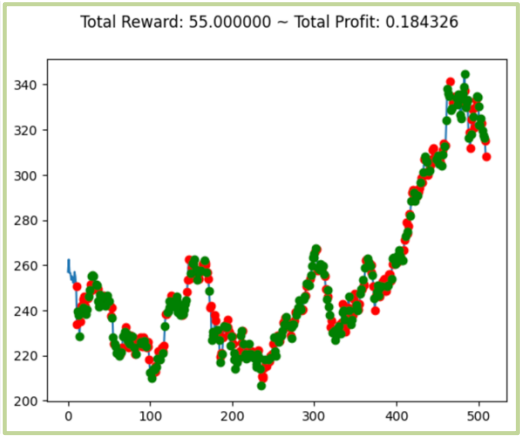
2. (15%) **試著修改與比較至少三項超參數（神經網路大小、一個 batch 中的回合數等），並說明你觀察到什麼。**

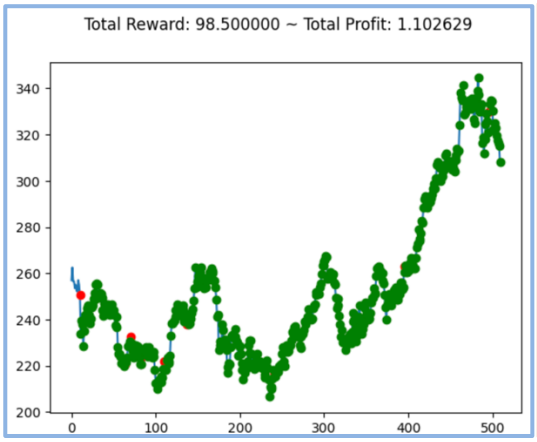
在此以第3題實作模型(Actor-Critic)來對各超參數進行實驗，固定seed排除隨機因素影響，並觀察測試集Reward及Profit結果。並以黃底標示較佳參數設定值。

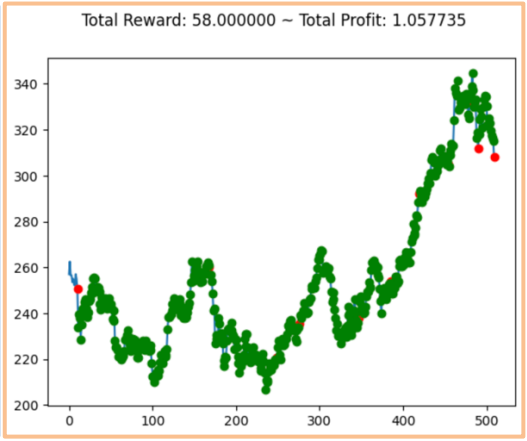
可以發現learning rate過小會使profit驟降；episode數量提升可加強訓練強度，提升reward；而gamma設定於0.85或0.5無顯著差異，但0.5略差一點，猜測是0.5會使discount reward降至太小。

超參數	設定值	測試Reward	測試Profit
learning rate	0.001	65.0	1.105
	0.0001	55	0.184
episode_per_batch	5	65.0	1.105
	10	98.5	1.102
gamma	0.85	65.0	1.105
	0.5	58.0	1.057









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。**

我選擇實作Actor-Critic模型，實際作法為將Actor與Critic做為兩個模型，並和進Agent裡交互使用與訓練。Actor的目的為預測出當下state的各動作可能性。Critic的目的是對預測當下state可能的reward。其內部Layer都為簡單的全連接層與激活函數而已，Actor與Critic的個別定義如下：

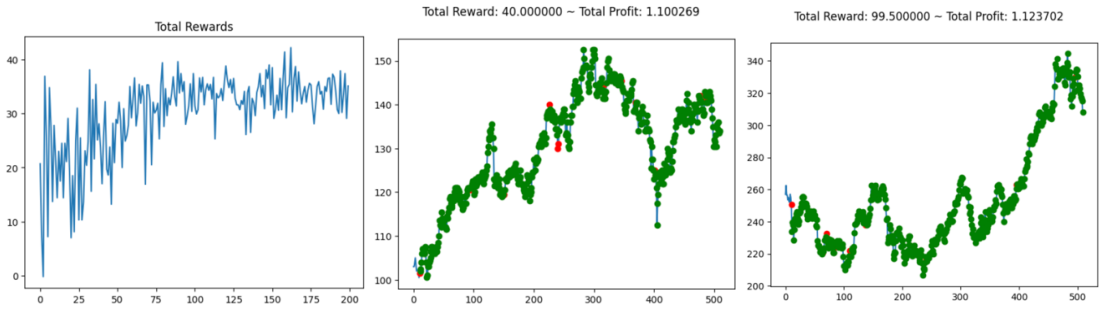
```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=32):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x): # input state, output prob of action
        x = self.fc1(x)
        x = torch.tanh(x)
        x = self.fc2(x)
        x = torch.softmax(x, dim=-1)
        return x

class Critic(nn.Module):
    def __init__(self, state_dim, hidden_dim=32):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 1)

    def forward(self, x): # input state, output score
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x
```

Agent內部將Actor與Critic整合，使用Adam作為共同的optimizer，同時訓練Actor與Critic，另外也定義gamma值用來計算之後的discounrt reward。預測函示定義為sample_action與evaluate_score，前者將各個action機率預測出來後，依照機率sample出其中一個action，同時計算機率的log_probility，可在未來計算loss時用到。最後便是learn function，這是模型最複雜處，須先計算discount reward，再將reward進行正規劃，並使用之前的reward、log_prob、score來計算loss，最後進行backward及最佳化。Agent實際定義如下：

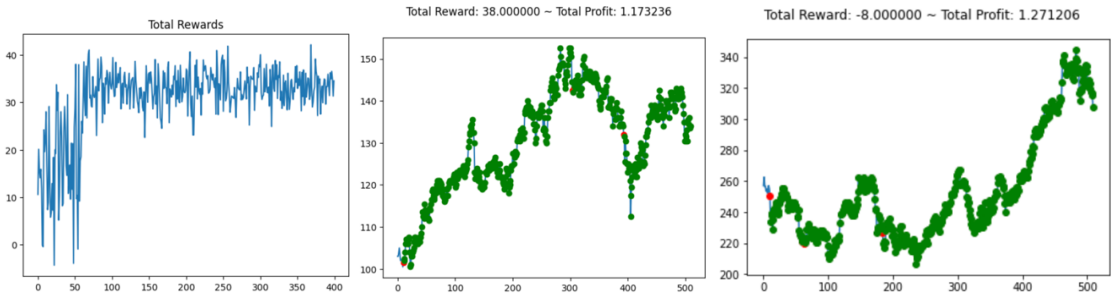
```python
class ACAgent():
    def __init__(self, state_dim, action_dim, lr=0.001, gamma=0.85):
        self.actor = Actor(state_dim, action_dim)
        self.critic = Critic(state_dim)
        self.opt = optim.Adam([{"params":self.actor.parameters()}, {"params":self.critic.parameters()}],lr=lr)
        self.gamma = gamma
        # records
        self.log_prob = []
        self.score = []
        self.reward = []

    def recode(self, log_prob:torch.Tensor, score:torch.Tensor, reward:torch.Tensor):
        self.log_prob.append(log_prob)
        self.score.append(score)
        self.reward.append(reward)

    def sample_action(self, state):
        action_prob = self.actor(state)
        action_dist = Categorical(action_prob)
        action = action_dist.sample()
        log_prob = action_dist.log_prob(action)
        return action.detach().item(), log_prob

    def evaluate_score(self, state):
        return self.critic(state)[0]

    def learn(self):
        # calculating discounted rewards
        rewards = []
        dis_reward = 0
        for reward in self.reward[::-1]:
            dis_reward = reward + self.gamma * dis_reward
            rewards.insert(0, dis_reward)
        # normalizing the rewards
        rewards = torch.tensor(rewards)
        rewards = (rewards - rewards.mean()) / (rewards.std() + 1e-9)
        # counting the loss
        loss = 0
        for log_prob, score, reward in zip(self.log_prob, self.score, rewards):
            advantage = reward - score.item()
            action_loss = -log_prob * advantage
            value_loss = F.smooth_l1_loss(score, reward)
            loss += (action_loss + value_loss)
        self.opt.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_value_(list(self.actor.parameters())+list(self.critic.parameters()), 0.5)
        self.opt.step()
```

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。**

Actor-Critic實驗結果（Train total reward, Train profit, Test profit）：



Policy Gradient實驗結果（Train total reward, Train profit, Test profit）：



從以上幾張圖可以發現，在training時的total reward與total prodfit差異並不大，但在testing時就有很大的落差。我認為此原因正是因為policy gradient不像actor-critic一樣有critic的模型可以去評估state的期望值（而非action的期望值），使得actor-critic結構更能夠去因state而去改變action的決定，這也是actor-critic的優點之一。

但actor-critic的其一缺點是難以訓練或收斂，最直接的就是因為參數量增加許多，另一方面因為actor-critic涉及兩個模型之間的互相影響，沒有足夠的資料及訓練時間是難以磨合兩個模型的。