Deep Reinforcement Learning HW3

110062126 林禾堃

Overview

在這一份作業當中我嘗試過 PPO 以及 SAC 來解決這次的問題，不過由於 PPO 訓練不太起來，因此後續完全以 SAC 為主軸。其中訓練上使用的技巧包含 Skip Frame, Frame Stack, Gray Scale, Crop Image 以及調整 Reward。

實驗上使用 NVIDIA GeForce RTX 4060 Laptop GPU 做訓練，最優模型大約訓練 2.3 小時，以 1.86MB 的模型在測試階段得到 628 的平均分數。

Training Methodology

接下來依序說明訓練過程中使用的方法，包含「Wrapper 處理」、「CNN Backbone 設計」、「Actor 設計」、「Critic 設計」、「ReplayBuffer 設計」、「Agent 設計」、「主程式設計」、「參數細節」。

Wrapper 處理

在訓練過程中我對於環境做了 RewardModify, SkipFrame, CropImage, GrayScale, FrameStack。部分的 Wrapper 與上一次作業同樣都是參考 PyTorch 教學修改。

- RewardModify

在第一輪的 SAC 訓練過程中發現 agent 出現停在草地轉動方向盤不前進的狀況，因此希望透過在 agent 進入草地給予 negative reward 來避免。這裡發現到環境當中有 driving_on_grass 紀錄當前 agent 是否在草地上，在這個狀況下 RewardModify 會給予-0.1 的 reward。

```
class RewardModify(gym.RewardWrapper):
    def __init__(self, env):
        super().__init__(env)
    def reward(self, reward):
        if self.env.driving_on_grass[0] == True:
            return np.array([-0.1], dtype=np.float32)
        return reward
```

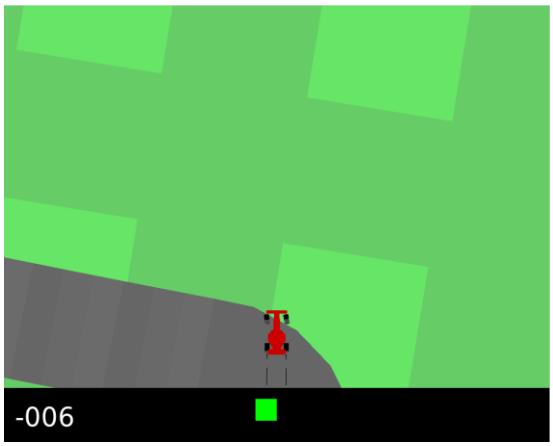
- SkipFrame

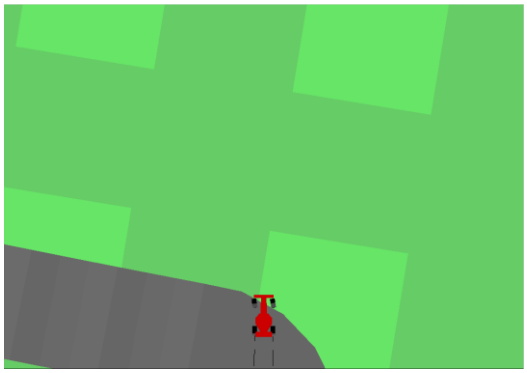
這裡選擇跳過 4 個 Frame。做法是重複同一個 action 連續 4 個 frame，並且累計 reward。

```
class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip=4):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        for _ in range(self._skip):
            obs, reward, end, info = self.env.step(action)
            total_reward += reward
            if end:
                break
        return obs, total_reward, end, info
```

- CropImage

在畫面底下有分數以及當前 agent 動作的資訊，在有 FrameStack 的狀態下模型應該有能力了解過去動作的狀況，所以認為這部分可以移除。計算後移除底下的 12 個 pixels 移除。

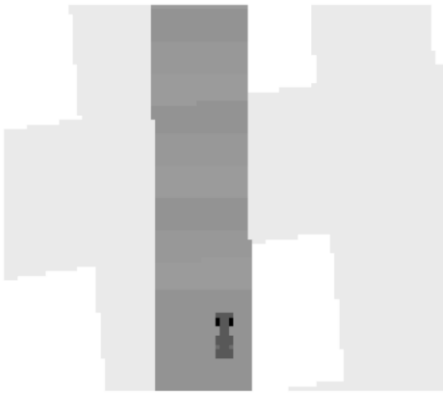




```
class CropImage(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        new_shape = list(env.observation_space.shape)
        new_shape[0] -= 12
        self.observation_space = Box(low=0, high=255, shape=tuple(new_shape), dtype=np.uint8)
    def observation(self, obs:np.ndarray):
        obs = obs.copy()[:, :-12, :, :]
        return obs
```

- GrayScale

顏色的部分即便不是 RGB 也還是能夠透過明暗判別道路以及草地，因此這裡直接轉換成灰階。



與上一份作業不同的地方在於不直接使用 Torchvision 處理，留下 Numpy 的格式。

```
class GrapScale(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        self.observation_space = Box(low=0, high=255, shape=self.observation_space.shape[:2], dtype=np.uint8)

    def observation(self, obs:np.ndarray):
        gray = np.dot(obs[..., :], [0.299, 0.587, 0.114])
        gray = gray / 128.0 - 1.0
        return gray
```

- FrameStack

雖然 OpenAI Gym 本身就有提供 FrameStack，但是為了避免有 LazyFrame 結構，留下單純的 Numpy，因此這裡直接將 FrameStack 的 source code 改寫。

```
def observation(self):
    assert len(self.frames) == self.num_stack, (len(self.frames), self.num_stack)
    return np.array(list(self.frames))
```

經過上述幾個 Wrapper 處理後，單一 obervation 會變成(1,4,84,96)的大小。不過這次我們只需要一台車去跑，因此可以移除第一個 dimension，這部分我在 Training Loop 才處理。

CNN Backbone 設計

雖然看了許多 SAC 的設計上都是直接接收 Linear 的輸入，但是遇到圖片的 state 還是會希望透過 CNN 去擷取特徵，因此這裡與上一份作業使用類似的 CNN 架構。包含了三層的 CNN，經過 Flatten 後再經由 Linear 統一壓成(batch size, 256)的大小。參數初始化都採用 kaiming normal。

```
def build_net(self):
    # Convolution 1
    conv1 = nn.Conv2d(in_channels=c, out_channels=32, kernel_size=4, stride=2)
    nn.init.kaiming_normal_(conv1.weight, mode='fan_out', nonlinearity='leaky_relu')
    cnn.add_module("conv_1", conv1)
    cnn.add_module("relu_1", nn.LeakyReLU())

    # Convolution 2
    conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=2, stride=2)
    nn.init.kaiming_normal_(conv2.weight, mode='fan_out', nonlinearity='leaky_relu')
    cnn.add_module("conv_2", conv2)
    cnn.add_module("maxpool1", nn.MaxPool2d(kernel_size=2))

    # Convolution 3
    conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=1)
    nn.init.kaiming_normal_(conv3.weight, mode='fan_out', nonlinearity='leaky_relu')
    cnn.add_module("conv_3", conv3)
    cnn.add_module("maxpool2", nn.MaxPool2d(kernel_size=2))

    # Reshape CNN output
    cnn.add_module("flatten", torch.nn.Flatten())

    # # Calculate input size
    state = torch.zeros(*(self.input_dim))
    dims = cnn(state)
    line_input_size = int(np.prod(dims.size()))

    # Linear 1
    line1 = nn.Linear(line_input_size, 256)
    nn.init.kaiming_normal_(line1.weight, mode='fan_out', nonlinearity='relu')
    cnn.add_module("line_1", line1)
    cnn.add_module("relu_2", nn.LeakyReLU())
```

Summarize 結果如下。

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 41, 47]           2,080
         LeakyReLU-2          [-1, 32, 41, 47]               0
            Conv2d-3          [-1, 64, 20, 23]           8,256
         MaxPool2d-4          [-1, 64, 10, 11]               0
            Conv2d-5           [-1, 64, 9, 10]          16,448
         MaxPool2d-6            [-1, 64, 4, 5]               0
           Flatten-7                [-1, 1280]               0
            Linear-8                 [-1, 256]         327,936
         LeakyReLU-9                 [-1, 256]               0
================================================================
Total params: 354,720
Trainable params: 354,720
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.12
Forward/backward pass size (MB): 1.29
Params size (MB): 1.35
Estimated Total Size (MB): 2.76
----------------------------------------------------------------
```

Actor 設計

Actor 的目的是要去給出 action，在連續的 action 上我使用 Normal Distribution 去 sample，然後再透過簡單的 Linear layers 產出μ,σ，調整 sample 出來的結果。不過看到其他人在實作的時候是讓 Linear layer 輸出取過 log 的σ，似乎會比較好訓練，因此這裡也這樣實作。另外，也透過 clamp 避免一些怪異的數值，讓訓練更加穩定。

```python
def forward(self, state):
    x = self.cnn(state).to(self.device)
    x = F.relu(self.fc1(x), inplace=True)
    x = F.relu(self.fc2(x), inplace=True)
    mu = self.mu(x)

    log_std = self.log_std_linear(x)
    log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)
    return mu, log_std
```

在實際選擇 action 的時候需要特別注意 action space 的數值範圍，這裡我是將$\mu + e \times \sigma$經過 tanh 函數，因此對於最後兩個範圍是[0,1]的 actions 都分別加上 1 再除 2。

```python
def get_action(self, state):
    mu, log_std = self.forward(state)
    std = log_std.exp()
    dist = Normal(0, 1)
    e = dist.sample().to(self.device)
    action = torch.tanh(mu + e * std)[0].cpu() + torch.tensor([0., 1., 1.]) / torch.tensor([1., 2., 2.])
    action = action.cpu()
    return action
```

Critic 設計

Critic 就比較簡單了，目的是要預測 Return，因此輸出只有一個數值。這裡選擇再經過三層 Linear layers。

```python
def forward(self, state, action):
    state = self.cnn(state).to(self.device)
    if type(action) == np.ndarray:
        action = torch.from_numpy(action).to(self.device)
    else:
        action = action.to(self.device)
    x = torch.cat((state, action), dim=1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    return self.fc3(x)
```

ReplayBuffer 設計

這次還沒採用 PER 就有不錯的效果，因此只留下最簡易的 Replay Buffer 實作。

```
def __init__(self, buffer_size=20000, batch_size=32, random_seed=1, device='cuda'):
    self.buffer         = deque(maxlen=buffer_size)
    self.size           = 0
    self.buffer_size    = buffer_size
    self.batch_size     = batch_size
    self.random_seed    = random.seed(random_seed)
    self.device         = device

def store(self, state, action, reward, next_state, done):
    experience = self.to_experience(state, action, reward, next_state, done)
    self.buffer.append(experience)
    self.size = min(self.size+1, self.buffer_size)
```

Sample 也沒有任何優化，單純透過 random 找出對應 experiences 疊起來回傳。

```
def sample_batch(self):
    experiences = random.sample(self.buffer, k=self.batch_size)
    states = torch.from_numpy(np.vstack([e["state"] for e in experiences])).float().to(self.device)
    actions = torch.from_numpy(np.vstack([e["action"] for e in experiences])).float().to(self.device)
    rewards = torch.from_numpy(np.vstack([e["reward"] for e in experiences])).float().to(self.device)
    next_states = torch.from_numpy(np.vstack([e["next_state"] for e in experiences])).float().to(self.device)
    dones = torch.from_numpy(np.vstack([e["done"] for e in experiences]).astype(np.uint8)).float().to(self.device)
    return states, actions, rewards, next_states, dones
```

Agent 設計

主要 SAC 的設計，架構上包含一個 Actor、兩個 Critics、對應 Critic 的 Critic Targets、一個 Replay Buffer。此外在後續更新與計算使用的α是選擇 non-fixed 的版本。

每當執行一個 step，agent 就會將對應的 experience 存入 Replay Buffer 當中。而當 buffer 大小足夠去 sample 一個 batch size 時就會從 buffer 當中抽一個 batch 出來學習。

```
def step(self, state, action, reward, next_state, done, timestep, episode):
    self.buffer.store(state, action, reward, next_state, done)
    if self.buffer.size > self.batch_size:
        experiences = self.buffer.sample_batch()
        self.learn(timestep, experiences, episode)
```

學習的部份對應到 SAC algorithm 的 Step12~15。首先要計算 Critic Target y。

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{targ,i}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \tilde{a}' \sim \pi_\theta(\cdot|s')$$

先取得 Critic Target Min，透過 Actor 取得 log probability，就可以照上式計算結果y。

```
next_action, nxta_log_prob = self.actor.evaluate(next_states)
Q_target1 = self.critic1_target(next_states, next_action)
Q_target2 = self.critic2_target(next_states, next_action)
Q_target_min = torch.min(Q_target1, Q_target2).to(self.device)
nxta_log_prob = nxta_log_prob.to(self.device)
y = rewards.to(self.device) + self.gamma * (1 - dones.to(self.device)) * (Q_target_min - self.alpha * nxta_log_prob)
```

下一步要更新 Critics，算式實際上就單純的 MSE Loss。

```
Q1 = self.critic1(states, actions)
Q2 = self.critic2(states, actions)

critic_loss1 = F.mse_loss(Q1, y.detach())
critic_loss2 = F.mse_loss(Q2, y.detach())

self.critic1_optimizer.zero_grad()
critic_loss1.backward()
self.critic1_optimizer.step()

self.critic2_optimizer.zero_grad()
critic_loss2.backward()
self.critic2_optimizer.step()
```

接著要更新 Actor。但是因為我使用的是 non-fixed α，因此我先更新α。

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{a_t \sim \pi_t^*}[-\alpha_t \log \pi_t^*(a_t|s_t;\alpha_t) - \alpha_t \bar{\mathcal{H}}]$$

在 SAC 論文中$\bar{\mathcal{H}}$是設定為-action_dimension，這裡也這樣設定。

```
alpha = torch.exp(self.log_alpha).to(self.device)
actions_pred, pred_action_log_prob = self.actor.evaluate(states)
alpha_loss = - (self.log_alpha.cpu() * (pred_action_log_prob.cpu() + self.target_entropy).detach().cpu()).mean()
self.alpha_optimizer.zero_grad()
alpha_loss.backward()
self.alpha_optimizer.step()
self.alpha = alpha
```

接下來 Actor 的更新依照 Step14 的式子。

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \quad for\ i = 1,2$$

需要特別注意的是這裡要做的是 Gradient Ascent，因此會多一個負號。

```
Q1 = self.critic1(states, actions_pred.squeeze(0))
Q2 = self.critic2(states, actions_pred.squeeze(0))
Q_min = torch.min(Q1, Q2)
actor_loss = -(Q_min - alpha * pred_action_log_prob.squeeze(0)).mean()
```

最後就是更新 Target Networks，使用 EMA 更新。

```
for target_param, local_param in zip(self.critic1_target.parameters(), self.critic1.parameters()):
    target_param.data.copy_(self.rho*local_param.data + (1.0-self.rho)*target_param.data)
for target_param, local_param in zip(self.critic2_target.parameters(), self.critic2.parameters()):
    target_param.data.copy_(self.rho*local_param.data + (1.0-self.rho)*target_param.data)
```

主程式設計

主程式的部份，先把環境套上 Wrapper，初始化 agent，並開始 Training Loop。原先的 state 會有一個 dimension 表示表示不同 agent，但現在我們只需要一個，因此透過 squeeze 移除。而丟入 Network 當中的資料又需要有一個 dimension 描述 batch 數量，因此也透過 expand_dims 增加。

```
env = CreateEnv()
env.reset()
env.close()
agent = Agent(**config, device=device, wandb_run=run)
# load_ckpt(save_dir, episode, agent)

# Training Loop
for i_episode in trange(episode, n_episodes+1):
    state = env.reset()
    state = np.expand_dims(state.squeeze(), axis=0)
    score = 0
    for t in range(config['max_t']):
        action = agent.act(torch.from_numpy(state).float()).numpy()
        next_state, reward, done, info = env.step(action)
        next_state = np.expand_dims(next_state.squeeze(), axis=0)
        agent.step(state, action, reward, next_state, done, t, i_episode)
        state = next_state
        score += reward.item()

        if done:
            break

    scores_deque.append(score)
```

參數細節

- Random seed: 1
- Hidden layer size: 256
- Actor learning rate: 3.0e-4
- Critic learning rate: 3.0e-4
- Buffer size: 50000
- Batch size: 32
- ρ: 1.0e-2
- γ: 0.99
- Max_t: 2000

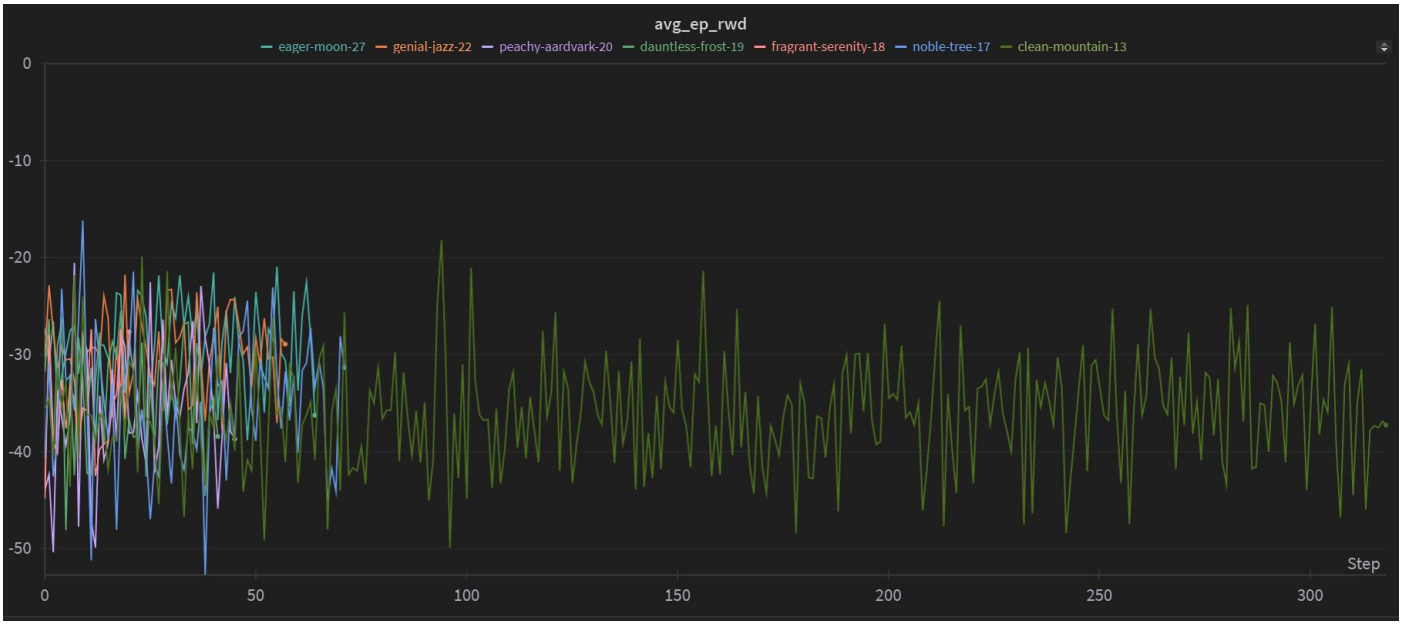
Testing Methodology

在 Test 階段基本上跟 Training 無異，但 Wrapper 需要改寫進 Actor 當中，並且需要 load 模型。

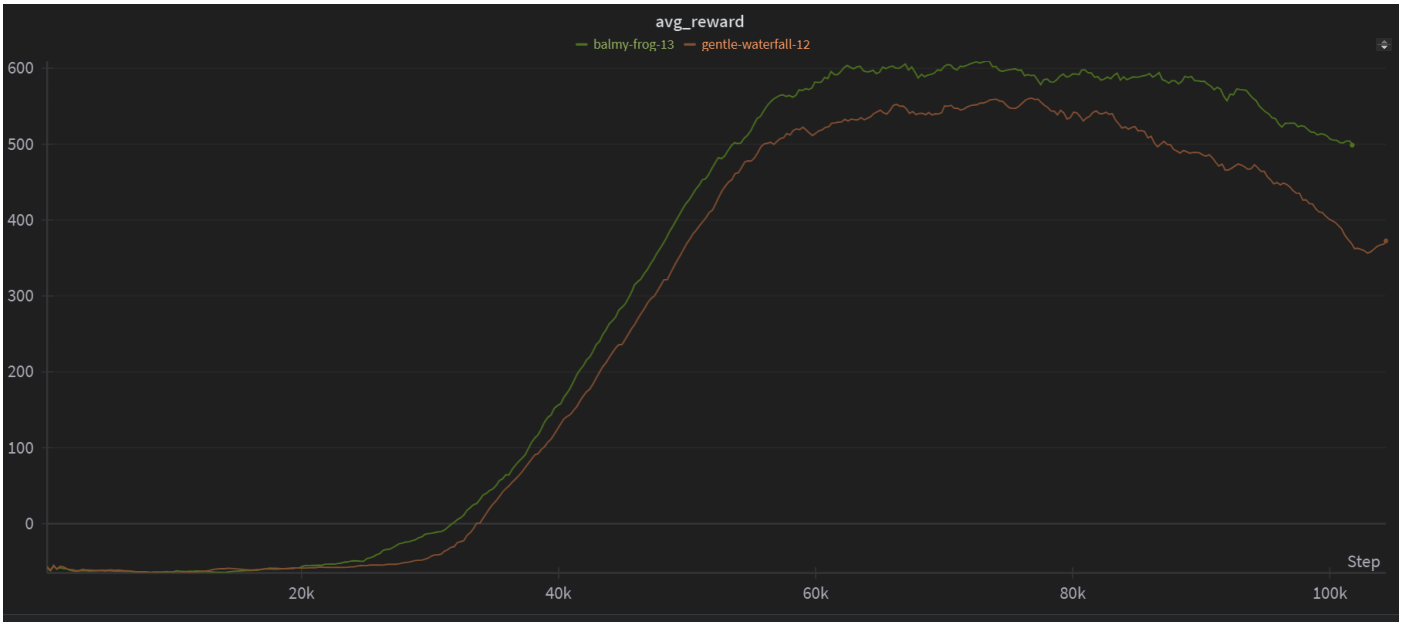
```python
def load(self, filepath='./110062126_hw3_data'):
    data = torch.load(open(filepath, 'rb'))
    self.actor.load_state_dict(data)

def act(self, state:np.ndarray):
    if self.frame_skip % 4 == 0:
        # Crop Image
        state = state.copy()[:, :-12, :, :]
        # Gray Scale
        state = np.dot(state[..., :], [0.299, 0.587, 0.114])
        state = state / 128.0 - 1.0
        while len(self.frames) < 4:
            self.frames.append(state)
        self.frames.append(state)
        state = np.array(list(self.frames))
        state = np.expand_dims(state.squeeze(), axis=0)
        state = torch.from_numpy(state).float()
        action = self.actor.get_action(state).detach().numpy()
        self.last_action = action
    self.frame_skip += 1
    return self.last_action
```

實驗與相關數據

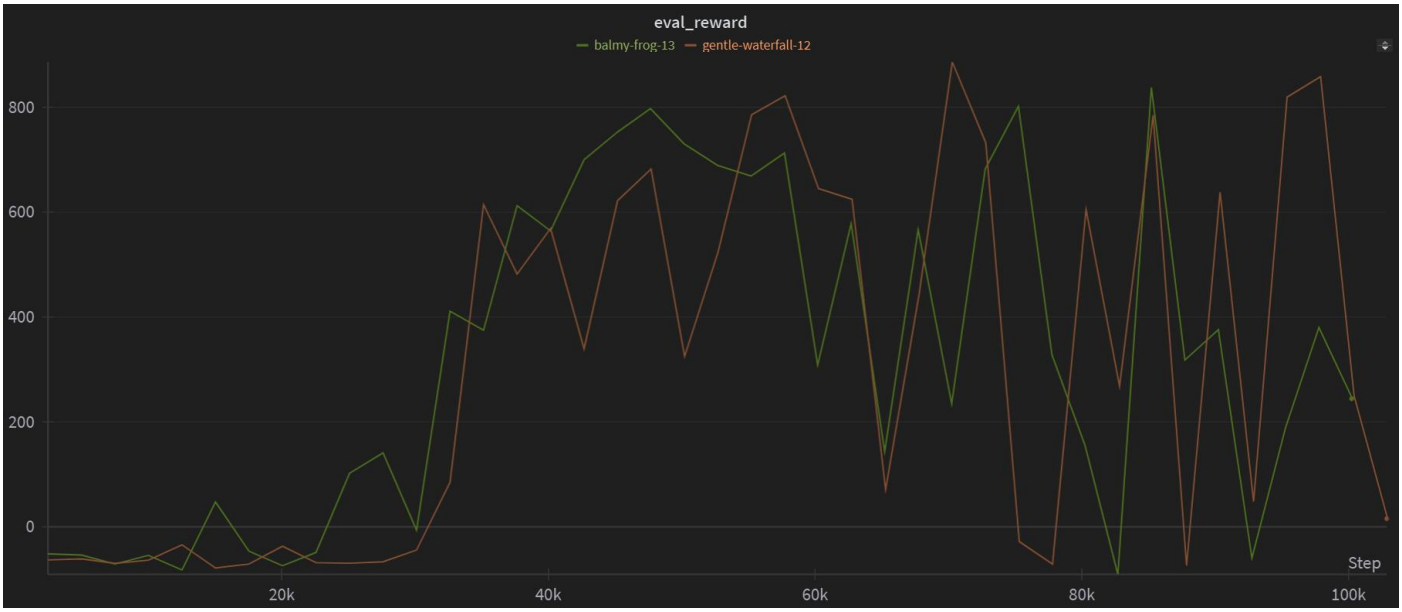
在起初選擇 PPO 去訓練，但是都沒有獲得好的成效。底下是當時的實驗結果，橫軸表示 Episode，縱軸表示平均的 Reward，可以發現到基本上都在-50~-20 之間震盪，即便經過 8 小時以上的訓練也並沒有明顯成長的趨勢，因此後續放棄 PPO 的實作。有聽說可以跟 LSTM 結合，但也許放在 Future Work 當中嘗試。



在 SAC 的部分，最一開始經過訓練後發現到模型時常會有卡在草叢不前進的狀況，後續認為可以透過在草地上給予懲罰，並且增加 Buffer Size，使得過去的記憶可以再更久才遺忘。結果如下圖所示。橫軸表示訓練的 step 數量，縱軸表示最新 100 個 episode 的平均 reward。棕色線是第一輪的訓練，綠色線則是改善後。可以發現到平均的 Reward 有獲得大約 50 上下的提升。



然而兩種狀況下單純 Evaluate 一個 episode 得到的 Reward 都是很不穩定的。



從視覺上來看則有觀察到，第一次訓練的結果時常會傾向快速移動，但很容易栽進草叢，接著就沒有前進動作。而第二次訓練則相較起來用稍慢的速度過彎，面對直線則會加速，然而當過彎進到草叢則會有一些機率同樣不再前進。

目前針對這個問題還沒有找到合適的解法，猜測是因為訓練過程中順利通過的狀況較多，導致 Replay Buffer 當中缺乏卡進草叢的狀況，所以忘記如何面對這種狀況，或許可以加上 PER 去改善。