Deep Reinforcement Learning HW4

110062126 林禾堃

Overview

在這一份作業當中我使用 DDPG 來解決這次的問題。其中訓練上使用的技巧包含 Skip Frame, Layer Normalization, NoisyNet, Ornstein–Uhlenbeck process 以及調整 Reward。

2017 年 NIPS 競賽中的結論有幾個。首先，前幾名的選手普遍都採用 DDPG 訓練，並且都有加上 Layer Normalization。接下來，普遍都有採用 CPU 平行化加速訓練。對於環境的處理上都會採用 Frame Skip，並且設定為 5。而 Layer Normalization 似乎能給出更好的訓練成效。這次的實作大量參考了這裡得出的結論。

實驗上使用 NVIDIA GeForce RTX 4060 Laptop GPU 做訓練，最優模型大約訓練 37 小時，以 214KB 的模型在測試階段得到 19.87 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」、「Backbone 設計」、「Actor 設計」、「Critic 設計」、「ReplayBuffer 設計」、「Ornstein-Uhlenbeck Process 設計」、「Agent 設計」、「主程式設計」、「參數細節」。

Wrapper 處理

在 Wrapper 上首先對於 Observation 處理，這裡選擇直接將狀態壓平，可以得到一個形狀為(339,)的狀態。

```python
def observation(self, observation):
    # Flatten dict into one numpy array with shape (339, )
    # v_tgt_field
    a = observation['v_tgt_field'].flatten()
    # pelvis
    b = [val for val in observation['pelvis'].values() if isinstance(val, float)]
    b.extend(observation['pelvis']['vel'])
    b = np.array(b)
    # r_leg
    c = np.array([val for v in observation['r_leg'].values() for val in (v if isinstance(v, list) else v.values())])
    # l_leg
    d = np.array([val for v in observation['l_leg'].values() for val in (v if isinstance(v, list) else v.values())])
    return np.concatenate((a, b, c, d)).astype('float32')
```

在 step 上由於後續設計的模型輸出的 action 範圍介在$[-1,1]$，因此先將其轉換到$[0,1]$，接下來套上 Frame Skip，設定跳過 5 個 Frames。並且針對 Reward 也有乘上 10 倍做訓練。

```python
def step(self, action):
    # [-1, 1] -> [0, 1]
    action = self.denormalize_action(action)

    # Skip Frames
    total_reward = 0
    for _ in range(self.skip_frames):
        obs, reward, done, _ = self.env.step(action)
        total_reward += reward
        self.env_step += 1
        if done:
            if self.env_step < 1000:
                total_reward += self.fail_reward
            break

    total_reward *= self.reward_scale
    obs = self.observation(obs)
    return obs, total_reward, done, None
```

Backbone 設計

在 Backbone 設計上只有簡單的 Linear Layers，並且包含了 Layer Normalization。

```python
class LinearNet(nn.Module):
    def __init__(self, layers, activation=torch.nn.ELU, layer_norm=False, linear_layer=nn.Linear):
        # Create a simple Linear Network
        # layers[0] -> layers[1] -> ... -> layers[-2] -> layers[-1]
        super(LinearNet, self).__init__()
        self.input_shape = layers[0]
        self.output_shape = layers[-1]
        self.net = nn.Sequential()

        for i in range(len(layers)-1):
            in_shape = layers[i]
            out_shape = layers[i+1]
            self.net.add_module(f'linear_{i}', linear_layer(in_shape, out_shape))
            if layer_norm == True:
                self.net.add_module(f'layer_norm_{i}', nn.LayerNorm(out_shape))
            self.net.add_module(f'act_{i}', activation())

    def forward(self, x):
        x = self.net.forward(x)
        return x
```

搭配的 NoisyNet 與前幾次作業中使用的相同，因此這裡不再贅述架構。

Actor 設計

Actor 的設計中包含了兩個 Linear Networks，分別是擷取特徵的 Feature Network 以及決定 Policy 的 Policy Network。初始化上 Feature Network 採用 kaiming_normal，而 Policy Network 採用 uniform。

```python
class Actor(nn.Module):
    def __init__(self, n_observation, n_action, layers,
                 activation=nn.ELU, layer_norm=True, last_activation=nn.Tanh, init_w=3e-3):
        super(Actor, self).__init__()

        linear_layer = NoisyNetLayer
        self.feature_net = LinearNet(
            layers=[n_observation] + layers,
            activation=activation,
            layer_norm=layer_norm,
            linear_layer=linear_layer)
        self.policy_net = LinearNet(
            layers=[self.feature_net.output_shape, n_action],
            activation=last_activation,
            layer_norm=False
        )
        self.init_weights(init_w)

    def init_weights(self, init_w):
        for layer in self.feature_net.net:
            if isinstance(layer, nn.Linear):
                layer.weight.data = nn.init.kaiming_normal_(layer.weight.data, mode='fan_out', nonlinearity='relu')

        for layer in self.policy_net.net:
            if isinstance(layer, nn.Linear):
                layer.weight.data.uniform_(-init_w, init_w)
```

Critic 設計

Critic 設計上也包含兩個 Linear Networks，分別是擷取特徵的 Feature Network 以及一個 Value Network。初始化也與 Actor 相同，一個是以 kaiming_normal 初始化，一個是由 uniform 初始化。

```python
class Critic(nn.Module):
    def __init__(self, n_observation, n_action, layers,
                 activation=nn.ELU, layer_norm=False, init_w=3e-3):
        super(Critic, self).__init__()
        linear_layer = NoisyNetLayer
        self.feature_net = LinearNet(
            layers=[n_observation + n_action] + layers,
            activation=activation,
            layer_norm=layer_norm,
            linear_layer=linear_layer)
        self.value_net = nn.Linear(self.feature_net.output_shape, 1)
        self.init_weights(init_w)

    def init_weights(self, init_w):
        for layer in self.feature_net.net:
            if isinstance(layer, nn.Linear):
                layer.weight.data = nn.init.kaiming_normal_(layer.weight.data, mode='fan_out', nonlinearity='relu')
        self.value_net.weight.data.uniform_(-init_w, init_w)

    def forward(self, observation, action):
        observation = torch.from_numpy(observation) if isinstance(observation, np.ndarray) else observation
        action = torch.from_numpy(action) if isinstance(action, np.ndarray) else action
        x = torch.cat((observation, action), dim=1)
        x = self.feature_net.forward(x)
        x = self.value_net.forward(x)
        return x
```

ReplayBuffer 設計

Replay Buffer 的部分這次也單純採用基本的型態，沒有使用 Prioritize。

```python
class ReplayBuffer(object):
    def __init__(self, size):
        self.buffer = deque(maxlen=size)
        self.max_size = size

    def __len__(self):
        return len(self.buffer)

    def store(self, state, action, reward, next_state, done):
        data = (state, action, reward, next_state, done)
        self.buffer.append(data)

    def sample(self, batch_size):
        indice = np.random.randint(0, len(self.buffer), size=batch_size)

        obses, actions, rewards, next_obses, dones = [], [], [], [], []
        for i in indice:
            data = self.buffer[i]
            obs, action, reward, next_obs, done = data
            obses.append(np.array(obs, copy=False))
            actions.append(np.array(action, copy=False))
            rewards.append(reward)
            next_obses.append(np.array(next_obs, copy=False))
            dones.append(done)
        return np.array(obses), np.array(actions), np.array(rewards), np.array(next_obses), np.array(dones)
```

Ornstein-Uhlenbeck Process 設計

在 Exploration 除了透過 NoisyNet 來協助外，這裡也透過對 action 加上 noise 來提升探索。實際的做法是透過 Ornstein-Uhlenbeck Process 去 sample 噪點。

```python
class AnnealedGaussianProcess(object):
    def __init__(self, mu, sigma, sigma_min, n_steps_annealing=int(1e5)):
        self.mu = mu
        self.sigma = sigma
        self.n_steps = 0

        if sigma_min is not None:
            self.m = -float(sigma - sigma_min) / float(n_steps_annealing)
            self.c = sigma
            self.sigma_min = sigma_min
        else:
            self.m = 0.
            self.c = sigma
            self.sigma_min = sigma

    def reset_states(self):
        pass

    @property
    def current_sigma(self):
        sigma = max(self.sigma_min, self.m * float(self.n_steps) + self.c)
        return sigma
```

```python
class OrnsteinUhlenbeckProcess(AnnealedGaussianProcess):
    def __init__(self, theta, mu=0., sigma=1., dt=1e-2,
                 x0=None, size=1, sigma_min=None, n_steps_annealing=int(1e5)):
        super(OrnsteinUhlenbeckProcess, self).__init__(
            mu=mu, sigma=sigma, sigma_min=sigma_min, n_steps_annealing=n_steps_annealing)
        self.theta = theta
        self.mu = mu
        self.dt = dt
        self.x0 = x0
        self.size = size
        self.reset_states()

    def sample(self):
        x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + \
            self.current_sigma * np.sqrt(self.dt) * np.random.normal(size=self.size)
        self.x_prev = x
        self.n_steps += 1
        return x

    def reset_states(self):
        self.x_prev = self.x0 if self.x0 is not None else np.zeros(self.size)
```

Agent 設計

在初始化 Agent 時會先處理儲存的資料夾以及訓練、測試使用的環境。接著初始化 wandb，再對 arguments 做一些處理。

```python
class Agent(object):
    def __init__(self, args, env, eval_env):
        self.save_dir = Path(args.logdir) / datetime.datetime.now().strftime("%Y-%m-%dT%H-%M-%S")
        self.save_dir.mkdir(parents=True)
        self.env = env
        self.eval_env = eval_env

        with open(self.save_dir / "args.json", "w") as fout:
            json.dump(vars(args), fout, indent=4, ensure_ascii=False, sort_keys=True)

        self.run = wandb.init(
            entity="koioslin",
            project="DRL_HW4",
            config=vars(args)
        )

        args.n_action = env.action_space.shape[0]
        args.n_observation = env.observation_space.shape[0]

        args.actor_layers = list(map(int, args.actor_layers.split('-')))
        args.critic_layers = list(map(int, args.critic_layers.split('-')))

        args.actor_activation = activations[args.actor_activation]
        args.critic_activation = activations[args.critic_activation]

        self.args = args
```

接著就可以初始化 Actor, Critic, Target Actor, Target Critic Networks。

```python
# Step1: Initialize actor and critic networks
self.actor, self.critic = create_model(args)
self.actor.train()
self.critic.train()
self.actor.to(device)
self.critic.to(device)
self.actor_lr_decay_fn = create_decay_fn("linear", initial_value=args.actor_lr, final_value=args.actor_lr_end, ma
self.critic_lr_decay_fn = create_decay_fn("linear", initial_value=args.critic_lr, final_value=args.critic_lr_end,

# Step2: Initialize target actor and target critic networks
self.target_actor, self.target_critic = create_model(args)
self.target_actor.load_state_dict(self.actor.state_dict())
self.target_critic.load_state_dict(self.critic.state_dict())
self.target_actor.to(device)
self.target_critic.to(device)
```

Learning Rate Decay 這裡選擇用 Linear。

```
def create_decay_fn(decay_type, initial_value, final_value, max_step=None, cycle_len=None, num_cycles=None):
    if decay_type == "linear":
        def decay_fn(step):
            relative = 1. - step / max_step
            return initial_value * relative + final_value * (1. - relative)
        return decay_fn
```

最後是初始化 Replay Buffer 以及 Random Process 等。

```
# Step3: Initialize Replay Buffer
self.buffer = ReplayBuffer(args.buffer_size)

self.random_process = OrnsteinUhlenbeckProcess(size=args.n_action, theta=args.rp_theta, mu=args.rp_mu, sigma=args.rp_sigma,
self.total_steps = 0
self.update_steps = 0
self.epsilon_cycle_len = random.randint(args.epsilon_cycle_len // 2, args.epsilon_cycle_len * 2)
self.epsilon_decay_fn = create_decay_fn(
    "cycle",
    initial_value=args.initial_epsilon,
    final_value=args.final_epsilon,
    cycle_len=self.epsilon_cycle_len,
    num_cycles=args.max_episodes // self.epsilon_cycle_len)
```

Agent 的動作選擇在經過 Actor Network 之後會再加上 noise，最終 clip 到[−1,1]之間。

```
def act(self, observation, noise=0):
    action = to_numpy(self.actor(to_tensor(np.array([observation], dtype=np.float32)))).squeeze(0)
    action += noise
    action = np.clip(action, -1.0, 1.0)
    return action
```

接下來進到 Train 的部分。這次使用 DDPG，參考 DDPG 論文當中的 pseudo code 去實作。前面我們已經完成了 Step1~Step3 的初始化。

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

首先是 optimizer，這裡對 Actor 以及 Critic 都是使用 Adam。

```
def train(self):
    args = self.args
    env = self.env
    self.actor_optim = torch.optim.Adam(self.actor.parameters(), lr=args.actor_lr)
    self.critic_optim = torch.optim.Adam(self.critic.parameters(), lr=args.critic_lr)
    ep_rewards = []
```

接下來進入 Training Loop。首先初始化 Random Process，並且取得 initial state。

```
# Loop episodes
for self.episode in trange(args.max_episodes):
    # Step4: Initialize a random process N, here we use Ornstein-Uhlenbeck Process
    self.random_process.reset_states()
    # Step5: Receive initial observation state
    seed = random.randrange(2 ** 32 - 2)
    observation = env.reset(seed=seed)
    done = False
```

接下來把 Learning Rate, Criterion 以及 epsilon 等都先設定好。

```
self.actor_lr = self.actor_lr_decay_fn(self.update_steps)
self.critic_lr = self.critic_lr_decay_fn(self.update_steps)
self.actor_lr = min(args.actor_lr, max(args.actor_lr_end, self.actor_lr))
self.critic_lr = min(args.critic_lr, max(args.critic_lr_end, self.critic_lr))

self.criterion = QuadricLinearLoss(clip_delta=args.clip_delta)

self.epsilon = min(args.initial_epsilon, max(args.final_epsilon, self.epsilon_decay_fn(self.episode)))
ep_reward = 0
ep_critic_losses = []
ep_policy_losses = []
ep_steps = 0
```

接下來進入到一個 episode 當中，首先選擇一個 action，並且得到對應的 observation, reward 等。將這些儲存到 replay buffer 當中。當訓練的步數足夠時，我們會從 replay buffer 當中 sample 一個 batch 出來做 update。訓練到一個階段後會儲存當前的模型。

```
while not done:
    # Step6: Select action according to current policy
    action = self.act(observation, noise=self.epsilon * self.random_process.sample())
    next_observation, reward, done, _ = env.step(action)
    ep_reward += reward
    # Step8: Store transition in Replay Buffer
    self.buffer.store(observation, action, reward, next_observation, done)
    # Step9: Sample a random minibatch of N transitions from Replay Buffer
    if self.total_steps >= args.train_steps:
        observations, actions, rewards, next_observations, dones = self.buffer.sample(batch_size=args.batch_size)
        # Step10: Update
        metrics, info = self.update(observations, actions, rewards, next_observations, dones, self.actor_lr, self.critic_lr)
        ep_critic_losses.append(to_numpy(metrics['value_loss']))
        ep_policy_losses.append(to_numpy(metrics['policy_loss']))
        self.update_steps += 1

        if self.update_steps % args.save_step == 0:
            with torch.no_grad():
                self.save(self.update_steps)
                self.evaluate()
```

接下來說明 update 的部分。首先要計算y_i。

$$y_i = r_i + Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

```
# Step10: Set y_i = r_i + Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})
next_v_values = self.target_critic(
    to_tensor(next_observations, volatile=True),
    self.target_actor(to_tensor(next_observations, volatile=True)),
)
with torch.no_grad():
    reward_predicted = dones * args.gamma * next_v_values
td_target = rewards + reward_predicted
```

接著更新 Critic 與 Policy Networks。

```
# Step11: Update critic
self.critic.zero_grad()
v_values = self.critic(to_tensor(observations), to_tensor(actions))
value_loss = self.criterion(v_values, td_target, weights=weights)
value_loss.backward()

torch.nn.utils.clip_grad_norm(self.critic.parameters(), args.grad_clip)
for param_group in self.critic_optim.param_groups:
    param_group["lr"] = critic_lr

self.critic_optim.step()

# Step12: Update policy
self.actor.zero_grad()
policy_loss = -self.critic(
    to_tensor(observations),
    self.actor(to_tensor(observations))
)
policy_loss = torch.mean(policy_loss * weights)
policy_loss.backward()

torch.nn.utils.clip_grad_norm(self.actor.parameters(), args.grad_clip)
for param_group in self.actor_optim.param_groups:
    param_group["lr"] = actor_lr

self.actor_optim.step()
```

最後是更新 target networks。

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

```
# Step13: Update target networks
for target_param, param in zip(self.target_actor.parameters(), self.actor.parameters()):
    target_param.data.copy_(target_param.data * (1.0 - args.tau) + param.data * args.tau)
for target_param, param in zip(self.target_critic.parameters(), self.critic.parameters()):
    target_param.data.copy_(target_param.data * (1.0 - args.tau) + param.data * args.tau)
```

主程式設計

主程式的部分首先接收 arguments，接下來把環境套上 Wrapper，製作 Training 以及 Evaluate 使用的環境。最後將 agent 初始化後開始訓練。我的訓練過程當中由於有中斷，因此還會再多一個 load，從過去的 check point 繼續訓練。

```python
if __name__ == '__main__':
    args = parse_args()
    env = create_env(args)
    eval_env = create_env(args, visualize=True)
    agent = Agent(args=args, env=env, eval_env=eval_env)
    agent.load()
    agent.train()
```

參數細節

- Random seed: 1
- Skip Frame: 5
- Fail Reward: -0.2
- Reward Scale: 10
- Actor Linear Layers: 64-64
- Actor Activation: ReLU
- Actor Layer Normalization: True
- Actor Learning Rate: 0.001
- Actor Learning Rate End: 0.00005
- Critic Linear Layers: 64-32
- Critic Activation: ReLU
- Critic Layer Normalization: False
- Critic Learning Rate: 0.002
- Critic Learning Rate End 0.00005
- γ: 0.9
- Gradient Clip: 10
- τ: 0.0001
- Batch Size: 200
- Buffer Size: 1e6
- Initial ϵ: 0.5
- Final ϵ: 0.001
- ϵ cycle length: 200
- Random Process θ: 0.15
- Random Process σ: 0.2
- Random Process min σ: 0.15
- Random Process μ: 0
- Save Step: 1000

Testing Methodology

在 Test 階段基本大致與 Training 無異，但 Wrapper 需要改寫進 Actor 當中，而 Network 只需要留下 Actor 即可。首先是初始化的部分。

```
class Agent(object):
    def __init__(self):
        # Step1: Initialize actor and critic networks
        self.device = "cpu"
        self.actor = create_model()
        self.actor.to(self.device)
        self.load()
        self.frame_skip = 0
        self.last_action = None
```

在 act 這邊需要自己做一下 Frame Skip，並且把 observation 壓成一維，也記得要把 action 做 clip 以及換成[0,1]之間。

```
def act(self, observation):
    if self.frame_skip % 5 == 0:
        observation = self.modify_observation(observation)
        action = to_numpy(self.actor(to_tensor(np.array([observation], dtype=np.float32)))).squeeze(0)
        action = np.clip(action, -1.0, 1.0)
        action = action * 0.5 + 0.5
        self.last_action = action
    self.frame_skip += 1
    return self.last_action
```

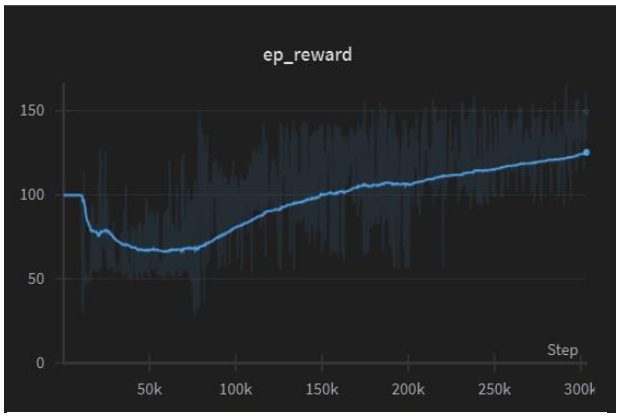
最後就是把模型 load 進來。

```
def load(self):
    data = torch.load(open("110062126_hw4_data", 'rb'), map_location=torch.device('cpu'))
    self.actor.load_state_dict(data)
    self.actor.eval()
```

實驗與相關數據

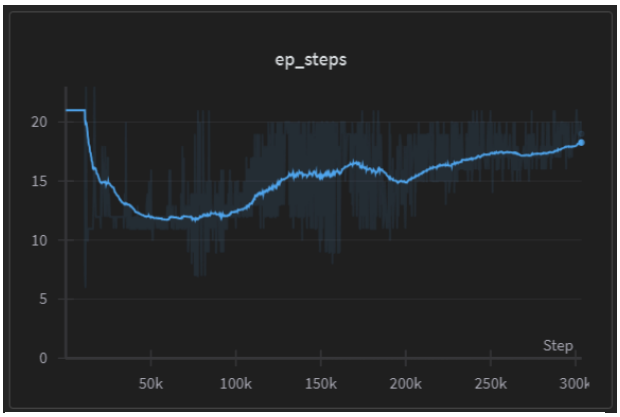
在實驗上首先有測試對 state 以及 action 做 flip 的測試。由於一個人可以縱切對稱，我們可以把左右的狀態互換去得到一個對應的結果。例如我們知道跨出右腳之後得到的 observation 以及 reward，那跨出左腳到同樣幅度的位置也能預期得到對應翻轉的 observation 以及 reward。如此一來理想上可以讓 Replay Buffer 更快累積各種經驗，並得到更好的學習曲線。

然而在實驗的結果上得到的結果是會得到成長更加緩慢的曲線，並且模型在視覺上來看並不會像前進，反而會後空翻。



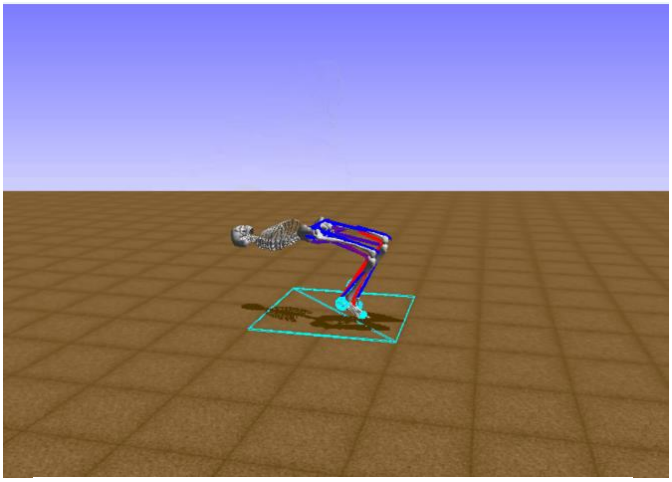
圖一：Flip 對應 Reward。

橫軸為訓練 step，縱軸為 Reward*10。



圖二：Flip 對應 Episode Steps。

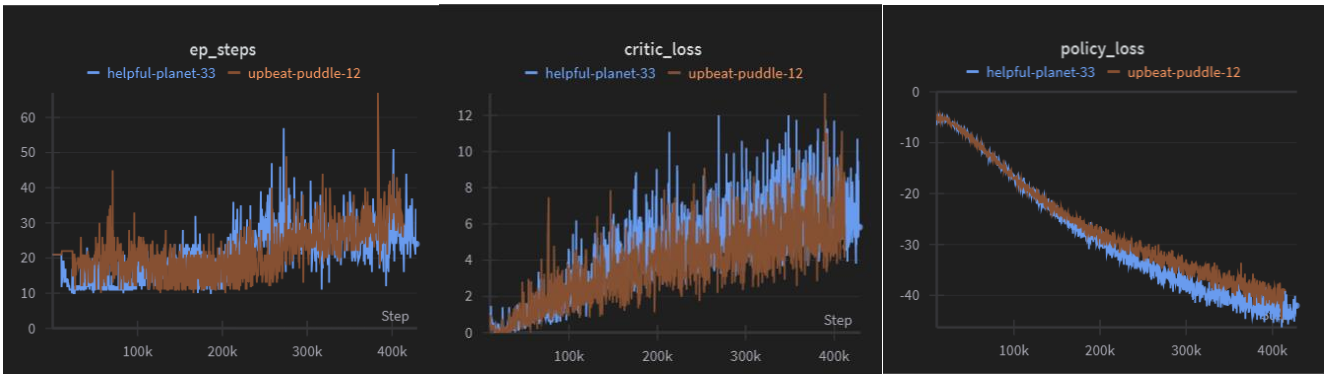
橫軸為訓練 step，縱軸為 Episode 的步數。



圖三：Flip 後反而只會後空翻。

因此經過實驗後決定不要採用 Flip Observation。

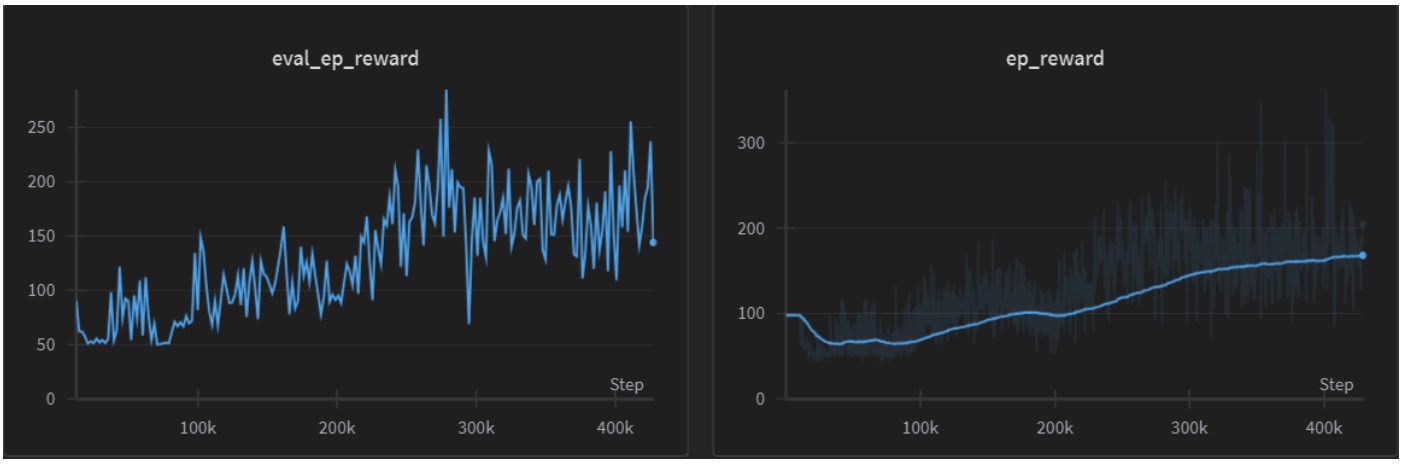
接下來是針對 Layer Normalization 的測試，由於實驗過程的疏失，因此這邊無法直接看到 Reward 的差異，但可以從 Episode 使用的步數看出其實使用 Layer Normalization 與否似乎沒有明顯的差異。Loss 的成長趨勢也相同。



圖四：LN 差異比較。

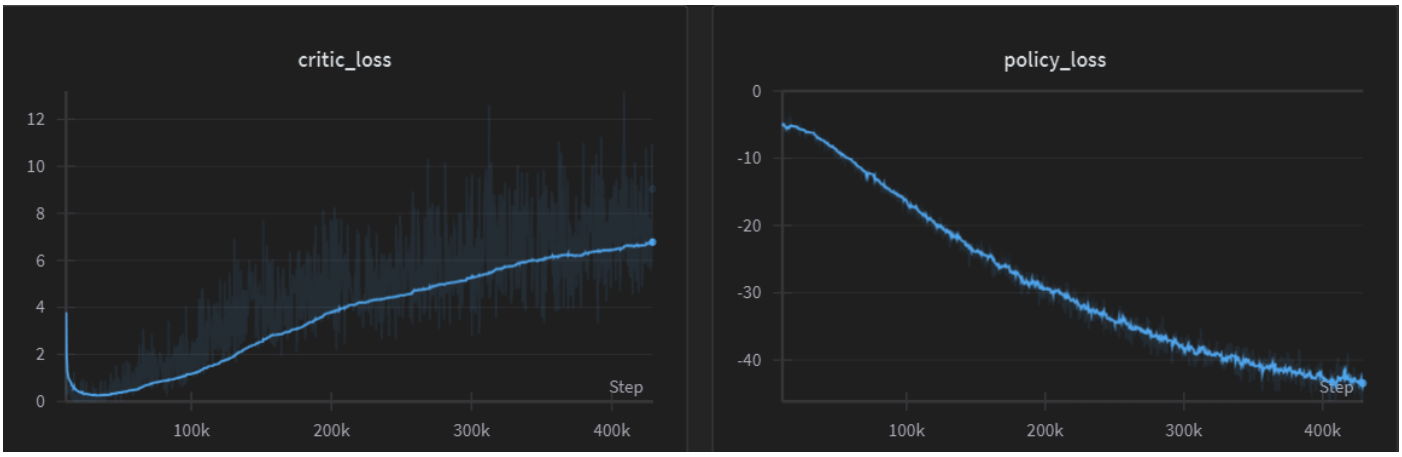
棕色線沒有使用 LN，藍色線有 LN。

最終訓練的結果如下。



圖五：最終結果

Evaluation ep reward & Training ep reward



圖六：最終結果

Critic Loss & Policy Loss

此次的最終結果仍然無法順利跑起來，頂多走兩步左右就會失去平衡。對於 Flip State 反而導致訓練不佳我還蠻意外，這部分我相當懷疑是自己的實作出了問題。此外，2017 年的結果當中大家幾乎都有採用 CPU 平行化，因此可以在大約 8 小時就訓練起來，相較於我訓練使用的時間，看起來也尚未收斂，也許以單一 CPU 還需要再更多的時間去訓練才行。