Deep Reinforcement Learning HW2

110062126 林禾堃

Overview

在這一份作業當中我嘗試將 Dueling DQN, Double DQN, Prioritize Experience Replay 以及 Noisy Net 合併在一起。由於最初就已經選定這樣的模型設定，也得到還不錯的結果，因此並沒有嘗試過單純使用 Double DQN、Dueling DQN 等其他的架構設計。圖片經過灰階化、Downsample 處理，並且搭配 Skip Frame 以及 Frame Stack 預處理輸入狀態。

實驗上使用 NVIDIA GeForce RTX 4060 Laptop GPU 做前期的訓練，後期則轉移到 NVIDIA GeForce RTX 2080 Ti 上訓練。最優的模型在訓練約 56 小時，以 2.63MB 的模型得到 8525 的分數。

Training Methodology

接下來依序說明訓練過程當中的方法，包含「Wrapper 處理」、「NoisyNet 設計」、「Dueling DQN 設計」、「Double DDQN 設計」、「Replay Buffer 設計」、「Agent 設計」、「主程式設計」、「參數細節」。

Wrapper 處理

在訓練當中我對於環境做了 Skip Frame, Gray Scale, Resize, Frame Stack 的處理。由於 OpenAI Gym 本身就有提供 Wrapper 可以自定義使用，因此根據教學寫出了底下的 Wrapper。

- Skip Frame

  重複 action、累積 reward 持續 4 個 frame，只取最後一個 frame。

```python
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
```

- Gray Scale

  轉換至 Tensor，交由 Torchvision 處理灰階。

```python
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 permute_orientation(self, obs):
        # Since torchvision use [C, H, W] rather than [H, W, C], we should transform it first
        return T.ToTensor()(obs.astype('int64').copy())

    def observation(self, obs):
        obs = self.permute_orientation(obs)
        obs = T.Grayscale()(obs)
        return obs
```

- Resize

  從(240, 256)降至(84, 84)，並做 Normalize。

```python
class DownSample(gym.ObservationWrapper):
    def __init__(self, env, shape=(84, 84)):
        super().__init__(env)
        self._shape = shape + self.observation_space.shape[2:] # Add the third axis
        self.observation_space = Box(low=0, high=255, shape=self._shape, dtype=np.uint8)

    def observation(self, obs):
        # Normalize because there would be FrameStack later on
        transforms = T.Compose(
            [T.Resize(self._shape, antialias=True), T.Normalize(0, 255)]
        )
        obs = transforms(obs.float()).squeeze(0)
        return obs
```

- Frame Stack

  直接使用了 Gym 當中的 FrameStack，會疊 4 個 frame。

NoisyNet 設計

根據 NoisyNet 的原始論文，像是 Dueling 和 Double 這種 DQN 變體由於只會在 Single-Thread 上訓練，因此產生 Random Number 建議採用 Factorised 的版本降低消耗的時間，因此這裡實作的是這個版本。

整體上 NoisyNet 仍然是一個 Linear Layer，只是在訓練期間 Weight 以及 Bias 會經過 sample noise，因此 Forward 的部分如下，單純的 Linear Layer。

```python
class NoisyNetLayer(nn.Module):
    def forward(self, x, sample_noise=True):
        """
        Forward pass the layer. If training, sample noise depends on sample_noise.
        Otherwise, use the default weight and bias.
        """
        if self.training:
            if sample_noise:
                self.sample_noise()
            return nn.functional.linear(x, weight=self.weight, bias=self.bias)
        else:
            return nn.functional.linear(x, weight=self.mu_weight, bias=self.mu_bias)
```

Factorised NoisyNet 產生隨機的方式是將 random number 拆分

$$\epsilon_{i,j}^{w} = f(\epsilon_i) f(\epsilon_j), \qquad \epsilon_j^{b} = f(\epsilon_j)$$

$$\text{where } f(x) = \text{sgn}(x)\sqrt{|x|}$$

我先將$f(\epsilon_i), f(\epsilon_j)$都 register 到 NoisyNet 當中，保留這個參數的同時也避免參與到 Backpropagation 等參數更新的行為。

```
def register_noise_buffers(self):
    """
    Register noise f(epsilon_in) and f(epsilon_out)
    """
    self.register_buffer(name='epsilon_input', tensor=torch.empty(self.in_features))
    self.register_buffer(name='epsilon_output', tensor=torch.empty(self.out_features))
```

接著就可以定義 sample noise 的方法。也就是上面函數 f 的實作。

```
def sample_noise(self):
    """
    Sample factorised noise
    f(x) = sgn(x)\sqrt{|x|}
    """
    with torch.no_grad():
        epsilon_input = torch.randn(self.in_features, device=self.epsilon_input.device)
        epsilon_output = torch.randn(self.out_features, device=self.epsilon_output.device)
        self.epsilon_input = (epsilon_input.sign() * torch.sqrt(torch.abs(epsilon_input))).clone()
        self.epsilon_output = (epsilon_output.sign() * torch.sqrt(torch.abs(epsilon_output))).clone()
    self.cached_weight = None
    self.cached_bias = None
```

如此一來 weight 以及 bias 也就可以計算出包含 noise 的版本。

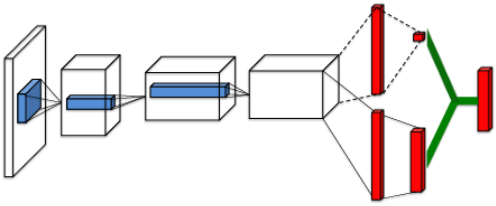
```
@property
def weight(self):
    """
    w = sigma \circ epsilon + mu
    epsilon = f(epsilon_in)f(epsilon_out)
    """
    if self.cached_weight is None:
        self.cached_weight = self.sigma_weight * torch.ger(self.epsilon_output, self.epsilon_input) + self.mu_weight
    return self.cached_weight
```

```
@property
def bias(self):
    """
    b = sigma \circ epsilon + mu
    """
    if self.cached_bias is None:
        self.cached_bias = self.sigma_bias * self.epsilon_output + self.mu_bias
    return self.cached_bias
```

參數的初始化如下。

```
def parameter_initialization(self):
    """
    Initialize with normal distribution
    """
    bound = self._calculate_bound()
    self.sigma_bias.data.fill_(value=self.sigma * bound)
    self.sigma_weight.data.fill_(value=self.sigma * bound)
    self.mu_bias.data.uniform_(-bound, bound)
    self.mu_weight.data.uniform_(-bound, bound)
```

Dueling DQN 設計



Dueling 的架構如上，包含了 CNN、Value、Advantage 三個部分。

CNN 的設計上由於模型有大小的限制，因此起初設計架構時期待用比較多的 Convolution 來避免 Linear Layer 的大量參數。這裡包含了 3 層的 Convolution，輸出經過 Flatten 後會經過兩層的 Linear Layer。NoisyNet 的部分只有加在 Value 以及 Advantage 上，因此初始化如下。

```
def __init__(self, input_dim, output_dim=256, n_actions=12, lr=0.0001, name='DDQN.ckpt'):
    super(DDQN, self).__init__()
    self.device = "cuda" if torch.cuda.is_available() else "cpu"
    self.input_dim = input_dim
    self.output_dim = output_dim
    self.ckpt_name = name

    self.cnn = self.build_cnn(input_dim[0])
    self.value = NoisyNetLayer(output_dim, 1)
    self.advantage = NoisyNetLayer(output_dim, n_actions)

    self.optimizer = torch.optim.Adam(self.parameters(), lr=lr)
    self.to(self.device)
```

首先是 CNN 的部分。初始化都是採用 kaiming normal。除了第三層的輸出是經過 Max Pooling 以外，其他都是經過 ReLU。

```
def build_cnn(self, c):
    cnn = torch.nn.Sequential()
    # Convolution 1
    conv1 = nn.Conv2d(in_channels=c, out_channels=32, kernel_size=4, stride=4)
    nn.init.kaiming_normal_(conv1.weight, mode='fan_out', nonlinearity='relu')
    cnn.add_module("conv_1", conv1)
    cnn.add_module("relu_1", nn.ReLU())

    # 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='relu')
    cnn.add_module("conv_2", conv2)
    cnn.add_module("relu_2", nn.ReLU())

    # 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='relu')
    cnn.add_module("conv_3", conv3)
    cnn.add_module("maxpool", nn.MaxPool2d(kernel_size=2))

    # Reshape CNN output
    class ConvReshape(nn.Module): forward = lambda self, x: x.view(x.size()[0], -1)
    cnn.add_module("reshape", ConvReshape())
```

Linear 的部份。同樣透過 kaiming normal 初始化，輸出都是經過 ReLU。

```
# Calculate input size
state = torch.zeros(1, *(self.input_dim))
dims = cnn(state)
line_input_size = int(np.prod(dims.size()))

# Linear 1
line1 = nn.Linear(line_input_size, 512)
nn.init.kaiming_normal_(conv1.weight, mode='fan_out', nonlinearity='relu')
cnn.add_module("line_1", line1)
cnn.add_module("relu_4", nn.ReLU())

# Linear 2
line2 = nn.Linear(512, self.output_dim)
nn.init.kaiming_normal_(conv1.weight, mode='fan_out', nonlinearity='relu')
cnn.add_module("line_2", line2)
cnn.add_module("relu_5", nn.ReLU())

return cnn
```

最後 Summarize 如下。

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Conv2d-1          [-1, 32, 21, 21]           2,080
             ReLU-2          [-1, 32, 21, 21]               0
           Conv2d-3          [-1, 64, 10, 10]           8,256
             ReLU-4          [-1, 64, 10, 10]               0
           Conv2d-5            [-1, 64, 9, 9]          16,448
        MaxPool2d-6            [-1, 64, 4, 4]               0
      ConvReshape-7                [-1, 1024]               0
           Linear-8                 [-1, 512]         524,800
             ReLU-9                 [-1, 512]               0
          Linear-10                 [-1, 256]         131,328
            ReLU-11                 [-1, 256]               0
================================================================
Total params: 682,912
Trainable params: 682,912
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.38
Params size (MB): 2.61
```

Double DDQN 設計

再來是加上 Double 的部份，也就是會有 Online 與 Target 的 DDQN。更新 Target 的部分我實作在 Agent 當中，每經過指定的 step 會直接 Hard Update。

```
def __init__(self, lr, input_dim, output_dim=256, n_actions=12, ckpt_dir='./ckeckpoints/', name='DDDQN.ckpt'):
    super(DDDQN, self).__init__()
    self.ckpt_dir = ckpt_dir
    self.ckpt_filepath = os.path.join(self.ckpt_dir, name)
    self.device = "cuda" if torch.cuda.is_available() else "cpu"
    self.input_dim = input_dim
    self.output_dim = output_dim

    # Double Networks
    self.online_net = DDQN(input_dim=input_dim, output_dim=output_dim, n_actions=n_actions, name='DDQN_online_{}.ckpt', lr=lr)
    self.target_net = DDQN(input_dim=input_dim, output_dim=output_dim, n_actions=n_actions, name='DDQN_target_{}.ckpt', lr=lr)
    self.target_net.load_state_dict(self.online_net.state_dict())
    for p in self.target_net.parameters():
        p.requires_grad = False
    self.to(self.device)
```

```
def sync_target(self):
    """..."""
    if self.learn_step_count % self.replace_step ==0:
        self.net.target_net.load_state_dict(self.net.online_net.state_dict())
```

Replay Buffer 設計

這裡採用的是 Prioritize Experience Replay，因此除了 Memory 以外也包含了 Priority。

```
def __init__(self, max_size, input_dim, n_actions, save_dir):
    self.mem_size = max_size
    self.mem_cntr = 0
    self.priorities = deque(maxlen=max_size)
    self.save_dir = save_dir

    self.state_memory = np.zeros((self.mem_size,*input_dim), dtype=np.float32)
    self.new_state_memory =np.zeros((self.mem_size,*input_dim), dtype=np.float32)
    self.action_memory = np.zeros(self.mem_size,dtype=np.int64)
    self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
    self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool_)
```

Priority 的設計採用 Proportional Prioritization，$p_i = |\delta_i| + \epsilon$。

```
def set_priorities(self, idx, delta, epsilon=1.1, alpha = 0.7):
    """...
    self.priorities[idx] = (np.abs(delta) + epsilon)** alpha
```

Sample 上採用 Importance-Sampling，$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}$。不過有看到網路上實作這一塊的時候有額外 normalize importance，藉此來讓他更加穩定，底下是取得 importance 的實作。

```
def get_importance(self, probabilities, beta):
    """...
    self.beta = beta
    importance =  np.power(1/self.mem_size * 1/probabilities, -self.beta)
    importance = importance / max(importance)
    return importance
```

Sample 時每個 experience 被 sample 到的機率依照 priority 決定。

```
def get_probabilities(self):
    """...
    scaled_priorities = np.array(self.priorities)
    scaled_priorities = scaled_priorities/ scaled_priorities.sum()
    return scaled_priorities
```

最後 Sample 會透過 probability 決定每個 experience 被 sample 到的機率，接下來取出 batch 並回傳 importance。

```
def sample(self, batch_size, beta):
    max_mem = min(self.mem_cntr, self.mem_size)
    sample_probs = self.get_probabilities()
    batch = np.random.choice(max_mem, batch_size, replace=False, p= sample_probs)

    states = self.state_memory[batch]
    actions = self.action_memory[batch]
    rewards = self.reward_memory[batch]
    next_states = self.new_state_memory[batch]
    dones = self.terminal_memory[batch]
    importance = self.get_importance(sample_probs[batch], beta)

    return states, actions, rewards, next_states, dones, importance, batch
```

Agent 設計

Agent 主要要做的事情就是「選擇 action」、「更新並取得 loss」。

在選擇 action 的部分這次不是用ϵ-greedy，而是依據 Advantage 決定。Advantage 越高就有越高的被選擇機率。

```
def select_action(self, state):
    """...
    state = state[0].__array__() if isinstance(state, tuple) else state.__array__()
    state = torch.tensor(state, device=self.device).unsqueeze(0)
    _, advantage = self.net.online_net.forward(state)
    prob = nn.Softmax(dim=-1)(advantage/self.temperature)
    prob = prob.cpu().detach().numpy()[0]
    action = np.random.choice(self.action_space, p=prob)
    self.total_step_count += 1
    return action
```

更新並取得 loss 首先會「每經過指定 step 更新 Target Network」，並且「從 Replay Buffer 中 Sample experience」。接下來就可以從 Online Network 以及 Target Network 取得對 state 以及 next_state 的預測 Value 與 Advantage。

```
def learn(self):
    if self.memory.mem_cntr < self.batch_size:
        return (-1, -1)

    self.net.online_net.optimizer.zero_grad()
    self.sync_target()

    states, actions, rewards, next_states, dones, importance, batch = self.sample()
    indices = np.arange(self.batch_size)

    # Value shape (batch,)
    # Advantage shape (batch,action_space.n)
    value_s, adv_s = self.net.online_net.forward(states)
    value_next_s, adv_next_s = self.net.target_net.forward(next_states)
    value_next_s_eval, adv_next_s_eval = self.net.online_net(next_states)
```

接著就可以計算出對應的 TD estimation 以及 TD Target。

```
"""
TD Estimate (in dueling)
    TD_e = Q_online^*(s, a) = V(s) + (A(s, a) - 1/|A| * \sum_a' A(s, a'))
"""
q_pred = torch.add(value_s, (adv_s - adv_s.mean(dim=1, keepdim=True)))[indices, actions]
q_next = torch.add(value_next_s,(adv_next_s - adv_next_s.mean(dim=1, keepdim=True)))
q_eval = torch.add(value_next_s_eval, (adv_next_s_eval-adv_next_s_eval.mean(dim=1, keepdim=True)))
max_actions = torch.argmax(q_eval, dim=1)
q_next[dones] = 0.0
"""
TD target
    a' = argmax_a Q_online(s', a)
    TD_t = r + gamma * Q_target^*(s', a')
"""
q_target = rewards + self.gamma * q_next[indices, max_actions]
```

Temporal-difference 計算的結果就可以交給前面設計好的 set_priorities 處理。

```
# Temporal-difference Error term for prioritized experience replay
diff = torch.abs(q_pred - q_target)
for i in range(self.batch_size):
    idx = batch[i]
    self.memory.set_priorities(idx, diff[i].cpu().detach().numpy())
```

最後就剩下更新與儲存模型。

```
self.net.online_net.optimizer.step()
self.learn_step_count +=1
self.decrement_temperature()

# Save model
if self.learn_step_count % self.save_every == 0:
    self.save()

return (q_pred.mean().item(), loss.item())
```

主程式設計

需要的東西都已經準備好了。首先處理環境，套上 Wrapper。

```
# Create Environment
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, COMPLEX_MOVEMENT)
env = SkipFrame(env)
env = GrapScale(env)
env = DownSample(env)
env = FrameStack(env, num_stack=4)
```

初始化 Agent, Wandb 與 Logger。

```
agent = Agent(state_dim=(env.observation_space.shape), action_num=env.action_space.n,
              replace_step=replace_step, beta=beta, gamma=gamma, temperature=temperature, temp_min=temp_min,
              temp_dec=temp_dec, lr=lr, save_every=save_every, batch_size=batch_size,
              save_dir=save_dir, load_dir=load_dir, last_code=load_code)

run = wandb.init(
    project="DRL-HW2",
    # Track hyperparameters and run metadata
    config={
        "replace_step": replace_step,
        "beta": beta,
        "gamma": gamma,
        "temperature": temperature,
        "temp_min": temp_min,
        "temp_dec": temp_dec,
        "learning_rate": lr,
        "save_every": save_every,
        "batch_size": batch_size
    },
)

logger = MetricLogger(save_dir, run)
```

接下來就可以進到 Training Loop。每個 episode 都是不斷重複「根據 state 選擇 action」、「執行 action」、「更新 Replay Buffer」、「更新模型」直到結束。

```
# Training Loop
for episode in trange(n_episodes):
    done = False
    state = env.reset()

    while not done:
        action = agent.select_action(state)
        next_state, reward, done, info = env.step(action)

        agent.store(state, action, reward, next_state, done)
        q, loss = agent.learn()
        logger.log_step(reward, loss, q)

        state = next_state

    logger.log_episode()
    if episode % 20 == 0:
        logger.record(episode, agent.total_step_count)
```

參數細節

底下列出此模型訓練的相關 Hyperparameters。

- 更新 Target Network 所需 Step：10000
- PER $\beta = 0.4$
- Double DQN $\gamma = 0.9$
- Action Softmax $\tau = 0.2$
  - 遞減值 $1e - 7$
  - 最小值 0.0004
- Adam Learning Rate 0.0001
- Batch Size 64
- Replay Buffer Size 100000
- NoisyNet $\sigma = 0.5$
- Skip Frame 4
- Frame Stack 4

Testing Methodology

在 Testing 階段概念與 Training 雷同，不過由於現在 Agent 在選擇 action 上僅能獲取 state 資訊，我們也不能直接對 environment 操作，我們也無法得知一個 episode 結束的時間點，因此有些部分需要做修改。底下分別說明「Wrapper 改寫」、「模型最佳成績篩選」。

Wrapper 改寫

首先最重要的問題是 agent 接收到的 state 不再是已經經過 Wrapper 處理的結果，因此這一塊需要移動到 action selection (act)當中處理。我首先將 Gray Scale 以及 Resize 改寫成兩個 Transform。

```
# Grayscale, resize, normalize
self.transforms1 = T.Compose(
    [T.ToTensor(), T.Grayscale()]
)
self.transforms2 = T.Compose(
    [T.Resize((84, 84), antialias=True), T.Normalize(0, 255)]
)
```

Skip Frame 的部分只需要用一個 counter 去維護現在接收的是第幾個 Frame，留下第 4 個 Frame 處理即可。Stack Frame 的部分則是新增一個大小為 4 的 deque，每次要處理的 Frame 都會存在其中，需要注意的只有對於第一個丟進去的 Frame 需要直接把 deque 填滿，才能維持輸入的 dimension。

```
# Skip Frame
if self.frame_skip % 4 == 0:
    # Gray Scale and Resize
    observation = self.transforms1(observation.astype('int64').copy())
    observation = self.transforms2(observation.float()).squeeze(0)

    # Frame Stack
    while len(self.frames) < 4:
        self.frames.append(observation)
    self.frames.append(observation)
    # To specific input format
    observation = gym.wrappers.frame_stack.LazyFrames(list(self.frames))
    observation = observation[0].__array__() if isinstance(observation, tuple) else observation.__array__()
    observation = torch.tensor(observation, device=self.device).unsqueeze(0)
```

接下來就可以透過 Advantage 決定 Action 了，跟 Training 相同。

```
    # Select Action
    _, advantage = self.net.target_net.forward(observation)
    prob = nn.Softmax(dim=-1)(advantage/0.3)
    prob = prob.cpu().detach().numpy()[0]
    self.last_action = np.random.choice(self.action_space, p=prob)
self.frame_skip += 1
self.timestamp += 1
```

剩下需要注意的僅有 NoisyNety 在 Forward 不需要 Sample Noise。

```
def forward(self, x, sample_noise=True):
    return nn.functional.linear(x, weight=self.mu_weight, bias=self.mu_bias)
```

模型最佳成績篩選

在 Training 階段平均的 Reward 大約都落在 6000 上下，不過詳細觀察可以發現每一輪的成績都不太相同，大多落在 5000~8000 的範圍。

```
Game #0: 5025.0
Game #1: 7861.0
Game #2: 7729.0
Game #3: 6967.0
Game #4: 6955.0
Game #5: 5084.0
Game #6: 7735.0
Game #7: 6967.0
Game #8: 6955.0
```

但是理想上一個模型如果能夠跑出夠高的成績，那麼每一個 episode 也應該能夠取得同樣好的成績。不過現在之所以會出現這樣的不同主要來自兩個因素。1. Action 選擇具有隨機性。2. 不知道結束時間點，導致 Frame Stack 沒有被初始化。

對於 Action 的選擇如果單純找 argmax 作為答案，會發現到表現通常不太好，在部分的位置會出現停在水管/方塊前不動，如果加上隨機性則有機會跳出這個困境，因此可以有更好的成績。

而對於第二個問題，我這裡選擇比較作弊的作法，透過固定 random seed，找出在第一個 episode 就有相當好成績的 seed，針對這個 episode 檢查他實際上經過多少 step，如此一來我就可以透過這個 step 決定什麼時候要初始化。

透過這個方法成功找到在 seed 1116 經過 3448 個 step 可以完成一個 8225 分的 episode，因此對 Agent 做一點小修改，讓他固定 random seed 以及在第 3448 個 step 更新。

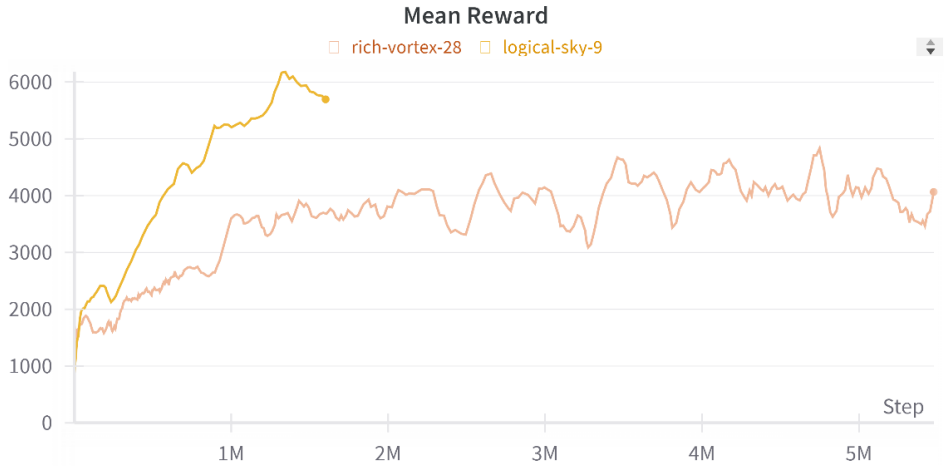
```python
def reset(self):
    np.random.seed(1116)
    self.timestamp = 0
    self.frame_skip = 0
    self.last_action = None
    self.frames = deque(maxLen=4)

def act(self, observation):
    """
    Choose action according to Advantage

    Inputs:
        observation (numpy array) : An observation of the current state
    Outputs:
        action_idx (int) : An integer representing the selected action
    """
    # print(self.timestamp)
    if self.timestamp == 3448:
        self.reset()
```

在不使用這樣的技巧下大概平均得分會落在 6500 分上下。

```
Game #46: 7735.0
Game #47: 6967.0
Game #48: 6006.0
Game #49: 6974.0
mean_reward: 6545.86
```

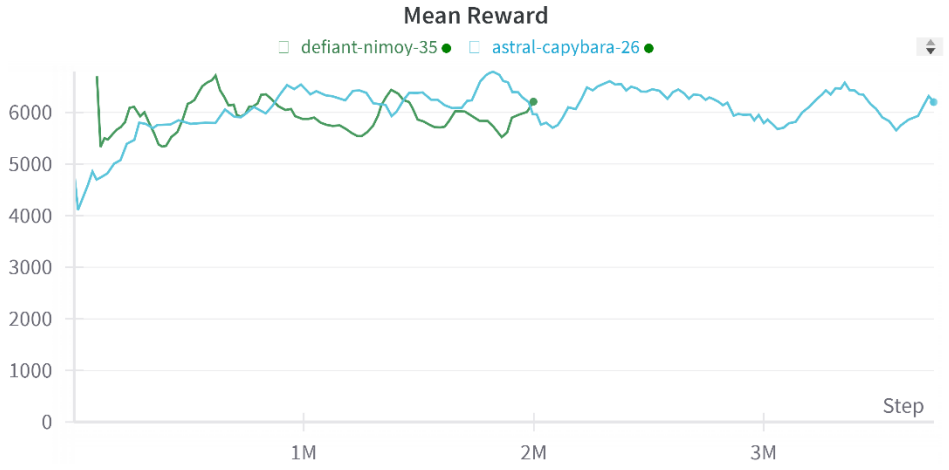
實驗與相關數據

在模型當中採用的 NoisyNet 只有運用在 Value 與 Advantage 上，如果把 Backbone 的 Linear 也換成 NoisyNet 的話會觀察到更新會更花時間，並且後續會停滯在平均 Reward 4000 上下。



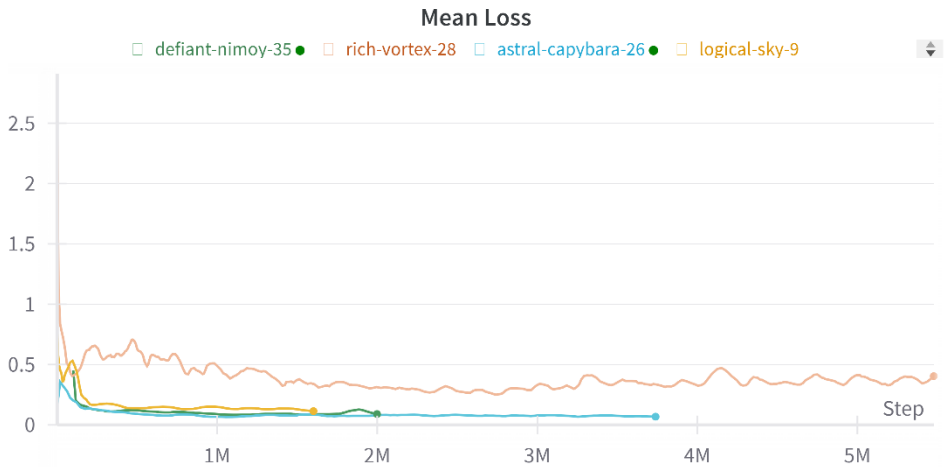
圖一、全 NoisyNet（橘線 rich-vortex-28）與部分 NoisyNet（黃線 logical-sky-9）比較。橫軸為更新 step，縱軸為平均 Reward。部分 NoisyNet 除了可以訓練更快以外，最後的 Mean Reward 也更高。

在轉移已經訓練好的模型繼續訓練時對於 Replay Buffer 有兩種處理方式。由於 Replay Buffer 並不會儲存起來（一個 Replay Buffer 需要約略 22GB 的記憶體空間，有些吃不消），因此在這裡我選擇嘗試 1. 從沒有任何 Experience 開始訓練。2. 先填滿 Replay Buffer 再開始訓練。比較的結果如下。



圖二、直接繼續訓練（藍線 astral-capybara-26）與填滿後再訓練（綠線 defiant-nimony-35）比較。橫軸為更新 step，縱軸為平均 Reward。除了開始訓練時 Mean Reward 有差異外，後續效果沒太大差異。

在 Loss 的部分，無論是哪一種測試，很快就可以降低並收斂，除了全 NoisyNet 的版本收斂的位置比起其他還要高。



圖三、Loss Curve 比較。可以觀察到很快就會收斂，但全 NoisyNet 的版本收斂的數值略高。

其他改善方向

目前 8225 的分數是透過比較作弊的方式取得的，實際上直接取平均會得到的分數只會落在 6500 分上下，進一步觀察模型的表現後發現到雖然模型已經幾乎能夠穩定進入第二關，但是往往對於水管當中的花沒辦法好好閃躲，導致分數停滯在 7000 多的狀況相當常見。

目前認為可行的一個方向是增加模型參數量。目前只有 2.63MB 的模型大小，我認為還有很多可以發揮的空間，像是辨認花的狀態與預測接下來移動方向。

另一個方向是想要嘗試 Curiosity Driven Exploration，不知道跟 NoisyNet 相比甚至是一起運用會不會有更好的成效。

最後是不進行灰階化處理。顏色的資訊也許對於物體的判斷會有更大的幫助，比起單一灰階來說可以更好判斷，因此不進行灰階化處理也許是一個可行的方向。

使用套件與說明

請參閱 HackMD 說明。