Reinforcement Learning Project

1. **Introduction**

   Báo cáo này sử dụng **Double Deep Q Network** như là phương pháp chính để giải bài toán **adversarial - cooperative multi-agent reinforcement learning** trong môi trường battle_v4 từ thư viện magent2. Phương pháp này sử dụng hai neural network: Q-network và target network, trong đó Q-network là một mạng tích chập nhận đầu vào là quan sát tác tử từ đó cho ra giá trị của từng hành động, target network là mạng có cấu trúc giống hệt Q-network với vai trò ổn định quá trình học bằng cách cập nhật tham số từ tham số của Q-network sau một khoảng thời gian. Kết hợp với việc sử dụng Replay Buffer - là bộ nhớ để lưu trữ các transition của từng tác tử, phục vụ cho quá trình huấn luyện. Phương pháp này đã đạt hiệu quả tốt chỉ sau 4 episode với thời lượng huấn luyện khoảng 1h30p. Kết quả evaluate trên 30 episode với max step của mỗi agent là 300:

Model	Win	Draw	Lose
Random	30 +- 0	0 +- 0	0 +- 0
Pretrain-0	30 +- 0	0 +- 0	0 +- 0
New Pretrain	30 +- 0	0 +- 0	0 +- 0

2. **Methods**

   a. **Model architecture**

      Sử dụng mô hình neural network với kiến trúc tích chập (CNN) đảm bảo tác tử có thể xử lý và trích xuất đầy đủ các chi tiết từ quan sát của môi trường (quan sát có kích thước 13*13*5).

      - Hai lớp tích chập đầu tiên đều sử dụng kích thước bộ lọc là 3*3 và đầu ra của mỗi lớp đều đi qua hàm kích hoạt ReLU để tăng tính phi tuyến và khả năng học các đặc trưng phức tạp.
      - Đầu ra của sau đó được duỗi (flatten) để đưa vào lớp fully connected.
      - Lớp fully connected đầu tiên đưa đầu vào đến không gian ẩn với số chiều là 120. Lớp fully connected thứ hai thu gọn đầu ra của lớp trước

đó về số chiều của hành động. Giữa các lớp này cũng dùng hàm kích hoạt ReLU để duy trì tính phi tuyến.

b. **Training approach**

Huấn luyện dựa trên phương pháp Q learning được cải tiến bằng cách đưa mạng học sâu vào để ước lượng hàm giá trị (Deep Q learning). Quá trình huấn luyện diễn ra theo các bước sau:

1. **Chiến lược chọn hành động (Epsilon-Greedy Policy)**

   Tác tử chọn hành động dựa trên chiến lược epsilon-greedy:

   - Chọn hành động ngẫu nhiên để khám phá với xác suất ϵ.
   - Chọn hành động tối ưu dựa trên q values được ước lượng từ mạng chính với xác suất $1 - \epsilon$.

   ϵ giảm dần theo số tập với công thức:

$$\epsilon = max(\epsilon_{end}, \epsilon_{start} - (\epsilon_{start} - \epsilon_{end}) \cdot \frac{steps}{\epsilon_{decay}})$$

2. **Replay Buffer:** Bộ nhớ được thiết kế để lưu trữ kinh nghiệm của từng tác tử gồm quan sát, hành động, phần thưởng, quan sát tiếp theo và trạng thái kết thúc của tác tử đó$(s, a, r, s', done)$.

   a. Dữ liệu được lưu trữ theo dạng FIFO đảm bảo kinh nghiệm mới nhất được lưu trữ và kinh nghiệm cũ nhất bị loại bỏ nếu đầy bộ nhớ.

   b. Lấy mẫu dữ liệu: các lô dữ liệu được lấy ngẫu nhiên từ bộ nhớ phục vụ cho quá trình huấn luyện, việc lấy mẫu là ngẫu nhiên trong bộ nhớ để đảm bảo phân bố độc lập, giống nhau của dữ liệu huấn luyện (i.i.d), tránh việc dữ liệu bị bias, cải thiện hiệu suất huấn luyện.

3. **Hàm loss**

   Sử dụng hàm loss Huber để tính sai số giữa q-values của mạng chính policy network và mạng mục tiêu (target network):

$$L = \text{SmoothL1Loss}\left(Q(s, a), r + \gamma \max_{a'} Q'(s', a')\right)$$

Trong đó:

$Q(s, a)$ là q values hiện tại.

$Q'(s', a')$ là q values mục tiêu.

γ là hệ số chiết khấu của phần thưởng tương lai.

r là phần thưởng sau hành động.

a là hành động của tác tử.

4. **Hàm tối ưu**

   Trọng số của mạng chính được cập nhật bằng hàm tối ưu AdamW kết hợp với gradient clipping để tránh gradient quá lớn dẫn đến mất ổn định quá trình huấn luyện.

5. **Soft update cho mạng mục tiêu**

   Sau mỗi bước di chuyển của tác tử mạng lại được cập nhật trọng số, do đó sử dụng phương pháp soft update mạng mục tiêu với hệ số τ theo công thức sau:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

   Trong đó:

   θ là trọng số mạng mục tiêu .

   θ' là trọng số mạng chính.

   τ là hệ số soft update

3. **Implementation**

   Q Network

```
class QNetwork(nn.Module):
    def __init__(self, observation_shape, action_shape):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(observation_shape[-1], observation_shap
            nn.ReLU(),
            nn.Conv2d(observation_shape[-1], observation_shap
```

```
            nn.ReLU(),
        )
        dummy_input = torch.randn(observation_shape).permute(
        dummy_output = self.cnn(dummy_input)
        flatten_dim = dummy_output.view(-1).shape[0]
        self.network = nn.Sequential(
            nn.Linear(flatten_dim, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, action_shape),
        )

    def forward(self, x):
        assert len(x.shape) >= 3, "only support magent input
        if len(x.shape) == 3:
            batchsize = 1
            x = x.unsqueeze(0)
        else:
            batchsize = x.shape[0]
        x = torch.fliplr(x).permute(0,3,1,2) # flip left-righ
        x = self.cnn(x)
        x = x.reshape(batchsize, -1)
        return self.network(x)
```

Replay Buffer

```
class MultiAgentReplayBuffer:
    def __init__(self, capacity, observation_shape, action_sh
        self.capacity = capacity
        self.observation_shape = observation_shape
        self.action_shape = action_shape

        # Use a defaultdict to automatically create deques fo
        self.buffers = defaultdict(lambda: {
            'obs': deque(maxlen=capacity),
```

```
            'action': deque(maxlen=capacity),
            'reward': deque(maxlen=capacity),
            'next_obs': deque(maxlen=capacity),
            'done': deque(maxlen=capacity),
        })

    def push(self, agent_id, obs, action, reward, next_obs, d
        self.buffers[agent_id]['obs'].append(obs)
        self.buffers[agent_id]['action'].append(action)
        self.buffers[agent_id]['reward'].append(reward)
        self.buffers[agent_id]['next_obs'].append(next_obs)
        self.buffers[agent_id]['done'].append(done)

    def sample(self, batch_size):
        all_agent_ids = list(self.buffers.keys())
        if not all_agent_ids:
            return None  # No agents in the buffer

        # Check if we have enough data to sample
        total_transitions = sum(len(self.buffers[agent_id]['d
        if total_transitions < batch_size:
            return None

        # Collect transitions from all agents into a single l
        all_transitions = []
        for agent_id in all_agent_ids:
            agent_buffer = self.buffers[agent_id]
            for i in range(len(agent_buffer['obs'])):
                all_transitions.append({
                    'obs': agent_buffer['obs'][i],
                    'action': agent_buffer['action'][i],
                    'reward': agent_buffer['reward'][i],
                    'next_obs': agent_buffer['next_obs'][i],
                    'done': agent_buffer['done'][i]
                })
```

```
        # Sample indices from the combined transitions
        indices = np.random.choice(len(all_transitions), batc

        # Extract the sampled transitions
        obs_batch = np.array([all_transitions[i]['obs'] for i
        action_batch = np.array([all_transitions[i]['action']
        reward_batch = np.array([all_transitions[i]['reward']
        next_obs_batch = np.array([all_transitions[i]['next_o
        done_batch = np.array([all_transitions[i]['done'] for

        return {
            'obs': obs_batch,
            'action': action_batch,
            'reward': reward_batch,
            'next_obs': next_obs_batch,
            'done': done_batch
        }

    def update_last_reward(self, agent_id, new_reward):
        if agent_id not in self.buffers:
            return
        self.buffers[agent_id]['reward'][-1] = new_reward

    def __len__(self):
        return sum(len(self.buffers[agent_id]['obs']) for age

    def clear(self, agent_id=None):
        if agent_id:
            self.buffers[agent_id]['obs'].clear()
            self.buffers[agent_id]['action'].clear()
            self.buffers[agent_id]['reward'].clear()
            self.buffers[agent_id]['next_obs'].clear()
            self.buffers[agent_id]['done'].clear()
        else:
```

```
            for agent_id in self.buffers:
                self.clear(agent_id)
```

Epsilon-Greedy Policy

```
def linear_epsilon(steps_done):
    return max(EPS_END, EPS_START - (EPS_START - EPS_END) * (

def policy(observation, q_network):
    global steps_done
    sample = random.random()
    if sample < linear_epsilon(steps_done):
        return env.action_space("red_0").sample()
    else:
        observation = (
            torch.Tensor(observation).to(device)
        )
        with torch.no_grad():
            q_values = q_network(observation)
        return torch.argmax(q_values, dim=1).cpu().numpy()[0]
```

Optimize

```
def optimize_model():
    global running_loss

    batch = buffer.sample(BATCH_SIZE)

    # Handle cases where the buffer doesn't have enough sampl
    if batch is None:
        return

    # Unpack the batch
    state_batch = torch.from_numpy(batch['obs']).float().to(d
    action_batch = torch.from_numpy(batch['action']).long().t
    reward_batch = torch.from_numpy(batch['reward']).float()
```

```
    next_state_batch = torch.from_numpy(batch['next_obs']).fl
    done_batch = torch.from_numpy(batch['done']).float().to(d

    # Reshape action_batch to (BATCH_SIZE, 1) for gather()
    action_batch = action_batch.unsqueeze(1)
    state_action_values = policy_net(state_batch).gather(1, a
    next_state_values = torch.zeros(BATCH_SIZE, device=device
    non_final_mask = (done_batch == 0).squeeze()  # Create a

    # Only compute for non-terminal states
    if non_final_mask.any():
        next_state_values[non_final_mask] = target_net(next_s

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_acti

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(policy_net.parameters(),
    optimizer.step()

    running_loss += loss.item()

    return loss.item()
```

Training Loop

```
for i_episode in range(num_episodes):
    env.reset()
    steps_done += 1
```

```
    for agent in env.agent_iter():

        observation, reward, termination, truncation, info =
        done = termination or truncation


        if done:
            action = None  # Agent is dead
            env.step(action)
        else:
            agent_handle = agent.split("_")
            agent_id = agent_handle[1]
            agent_team = agent_handle[0]
            if agent_team == "blue":
                episode_reward += reward

                buffer.update_last_reward(agent_id, reward) #

                action = policy(observation, policy_net)
                env.step(action)

                try:
                    next_observation = env.observe(agent)
                    agent_done = False
                except:
                    next_observation = None
                    agent_done = True

                reward = 0 # Wait for next time to be selecte

                # Store the transition in buffer
                buffer.push(agent_id, observation, action, re

                # Perform one step of the optimization (on th
                optimize_model()
```

```
                # Soft update of the target network's weights
                # θ' ← τ θ + (1 −τ )θ'
                target_net_state_dict = target_net.state_dict
                policy_net_state_dict = policy_net.state_dict
                for key in policy_net_state_dict:
                    target_net_state_dict[key] = policy_net_s
                target_net.load_state_dict(target_net_state_d

            else:
                # red agent
                action = np.random.randint(0, 21)
                env.step(action)
```

Validation

Tất cả mạng cho quá trình đánh giá

```
class QNetwork(nn.Module):
    def __init__(self, observation_shape, action_shape):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
        )
        dummy_input = torch.randn(observation_shape).permu
        dummy_output = self.cnn(dummy_input)
        flatten_dim = dummy_output.view(-1).shape[0]
        self.network = nn.Sequential(
            nn.Linear(flatten_dim, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, action_shape),
```

```
        )

    def forward(self, x):
        assert len(x.shape) >= 3, "only support magent inp
        x = self.cnn(x)
        if len(x.shape) == 3:
            batchsize = 1
        else:
            batchsize = x.shape[0]
        x = x.reshape(batchsize, -1)
        return self.network(x)

class FinalQNetwork(nn.Module):
    def __init__(self, observation_shape, action_shape):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
        )
        dummy_input = torch.randn(observation_shape).permu
        dummy_output = self.cnn(dummy_input)
        flatten_dim = dummy_output.view(-1).shape[0]
        self.network = nn.Sequential(
            nn.Linear(flatten_dim, 120),
            # nn.LayerNorm(120),
            nn.ReLU(),
            nn.Linear(120, 84),
            # nn.LayerNorm(84),
            nn.Tanh(),
        )
        self.last_layer = nn.Linear(84, action_shape)

    def forward(self, x):
        assert len(x.shape) >= 3, "only support magent inp
```

```
        x = self.cnn(x)
        if len(x.shape) == 3:
            batchsize = 1
        else:
            batchsize = x.shape[0]
        x = x.reshape(batchsize, -1)
        x = self.network(x)
        self.last_latent = x
        return self.last_layer(x)

class DQNNetwork(nn.Module):
    def __init__(self, observation_shape, action_shape):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
            nn.Conv2d(observation_shape[-1], observation_s
            nn.ReLU(),
        )
        dummy_input = torch.randn(observation_shape).permu
        dummy_output = self.cnn(dummy_input)
        flatten_dim = dummy_output.view(-1).shape[0]
        self.network = nn.Sequential(
            nn.Linear(flatten_dim, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, action_shape),
        )

    def forward(self, x):
        assert len(x.shape) >= 3, "only support magent inp
        if len(x.shape) == 3:
            batchsize = 1
            x = x.unsqueeze(0)
        else:
```

```
            batchsize = x.shape[0]
        x = torch.fliplr(x).permute(0,3,1,2) # flip left-r
        x = x.reshape(1, 5, 13, 13)
        x = self.cnn(x)
        x = x.reshape(batchsize, -1)
        return self.network(x)

def one_hot_encode(agent_id, num_classes=81, device="cpu")
  if not isinstance(agent_id, torch.Tensor):
      agent_id = torch.tensor(agent_id, device=device)
  else:
      agent_id = agent_id.to(device)

  return F.one_hot(agent_id.long(), num_classes=num_classe

class AgentNetwork(nn.Module):
    def __init__(self, observation_shape, action_dim, n_ag
        super(AgentNetwork, self).__init__()
        # observation_shape is (H, W, C)
        self.conv1 = nn.Conv2d(observation_shape[2], 16, k
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stri
        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stri

        # Calculate the flattened size after convolutions
        flat_size = 32 * observation_shape[0] * observatio

        # Add a linear layer to process the agent ID
        self.fc_agent_id = nn.Linear(n_agents, 32)

        self.fc1 = nn.Linear(flat_size + 32, 128)  # Conca
        self.fc2 = nn.Linear(128, action_dim)

    def forward(self, obs, agent_id):
        agent_id = one_hot_encode(agent_id, device=agent_i
        # Add a batch dimension
        if len(obs.shape) == 3:
```

```
            obs = obs.unsqueeze(0)
        if len(agent_id.shape) == 1:
            agent_id = agent_id.unsqueeze(0)
        x = torch.fliplr(obs).permute(0,3,1,2) # flip left
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.flatten(start_dim=1)  # Flatten all dimensio

        # Process agent ID
        agent_id_embedding = F.relu(self.fc_agent_id(agent

        # Concatenate the flattened convolutional output w
        x = torch.cat((x, agent_id_embedding), dim=1)

        x = F.relu(self.fc1(x))
        q_values = self.fc2(x)
        return q_values
```

Cài đặt quá trình đánh giá

```
def eval(algo):
    max_cycles = 300
    env = battle_v4.env(map_size=45, max_cycles=max_cycles
    device = "cuda" if torch.cuda.is_available() else "cpu

    if algo == "dqn":
        blue_network = DQNNetwork(
            env.observation_space("red_0").shape, env.acti
        )

        blue_network.load_state_dict(
            torch.load("models/blue_dqn_final.pt", weights
        )
        blue_network.to(device)
    elif algo == "qmix":
```

```
        blue_network = AgentNetwork(
            env.observation_space("red_0").shape, env.acti
        )

        blue_network.load_state_dict(
            torch.load("models/blue_qmix_final.pt", weight
        )
        blue_network.to(device)

    q_network = QNetwork(
        env.observation_space("red_0").shape, env.action_s
    )
    q_network.load_state_dict(
        torch.load("models/red.pt", weights_only=True, map
    )
    q_network.to(device)

    final_q_network = FinalQNetwork(
        env.observation_space("red_0").shape, env.action_s
    )
    final_q_network.load_state_dict(
        torch.load("models/red_final.pt", weights_only=Tru
    )
    final_q_network.to(device)

    def random_policy(env, agent, obs):
        return env.action_space(agent).sample()

    def dqn_policy(env, agent, obs):
        observation = (
            torch.Tensor(obs).float().to(device)
        )
        with torch.no_grad():
            q_values = blue_network(observation)
        return torch.argmax(q_values, dim=1).cpu().numpy()
```

```
    def qmix_policy(env, agent, obs):
        observation = (
            torch.Tensor(obs).float().to(device)
        )
        id = int(agent.split("_")[1])
        with torch.no_grad():
            q_values = blue_network(observation, torch.ten
        return torch.argmax(q_values, dim=1).cpu().numpy()

    def pretrain_policy(env, agent, obs):
        observation = (
            torch.Tensor(obs).float().permute([2, 0, 1]).u
        )
        with torch.no_grad():
            q_values = q_network(observation)
        return torch.argmax(q_values, dim=1).cpu().numpy()

    def final_pretrain_policy(env, agent, obs):
        observation = (
            torch.Tensor(obs).float().permute([2, 0, 1]).u
        )
        with torch.no_grad():
            q_values = final_q_network(observation)
        return torch.argmax(q_values, dim=1).cpu().numpy()

    def run_eval(env, red_policy, blue_policy, n_episode:
        red_win, blue_win = [], []
        red_tot_rw, blue_tot_rw = [], []
        n_agent_each_team = len(env.env.action_spaces) //

        for _ in tqdm(range(n_episode)):
            env.reset()
            n_kill = {"red": 0, "blue": 0}
            red_reward, blue_reward = 0, 0

            for agent in env.agent_iter():
```

```
                observation, reward, termination, truncati
                agent_team = agent.split("_")[0]

                n_kill[agent_team] += (
                    reward > 4.5
                )  # This assumes default reward settups
                if agent_team == "red":
                    red_reward += reward
                else:
                    blue_reward += reward

                if termination or truncation:
                    action = None  # this agent has died
                else:
                    if agent_team == "red":
                        action = red_policy(env, agent, ob
                    else:
                        action = blue_policy(env, agent, o

                env.step(action)

            who_wins = "red" if n_kill["red"] >= n_kill["b
            who_wins = "blue" if n_kill["red"] + 5 <= n_ki
            red_win.append(who_wins == "red")
            blue_win.append(who_wins == "blue")

            red_tot_rw.append(red_reward / n_agent_each_te
            blue_tot_rw.append(blue_reward / n_agent_each_

        return {
            "winrate_red": np.mean(red_win),
            "winrate_blue": np.mean(blue_win),
            "average_rewards_red": np.mean(red_tot_rw),
            "average_rewards_blue": np.mean(blue_tot_rw),
        }
```

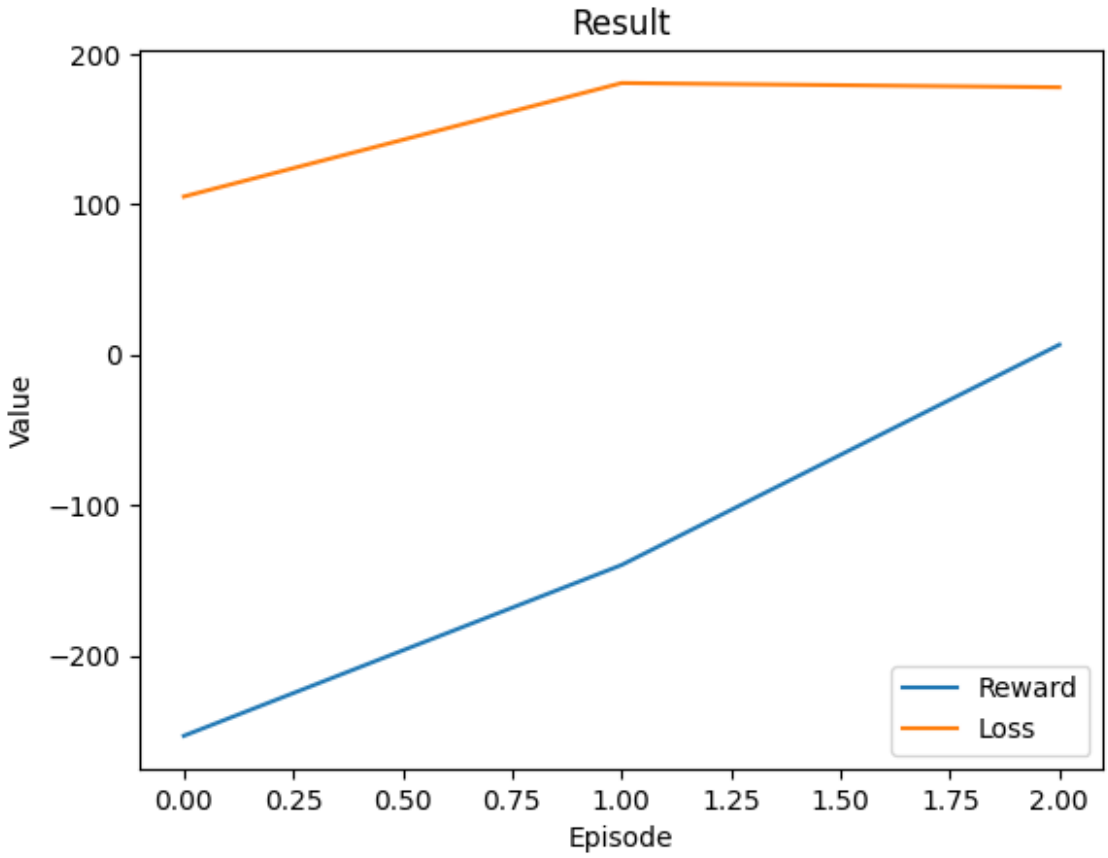
```
    if algo == "dqn":
        my_policy = dqn_policy
    elif algo == "qmix":
        my_policy = qmix_policy

    print("=" * 20)
    print("Eval with random policy")
    print(
        run_eval(
            env=env, red_policy=random_policy, blue_policy
        )
    )
    print("=" * 20)

    print("Eval with trained policy")
    print(
        run_eval(
            env=env, red_policy=pretrain_policy, blue_poli
        )
    )
    print("=" * 20)

    print("Eval with final trained policy")
    print(
        run_eval(
            env=env, red_policy=final_pretrain_policy, blu
        )
    )
    print("=" * 20)
```

4. Experimental Results

    a. Performance metrics

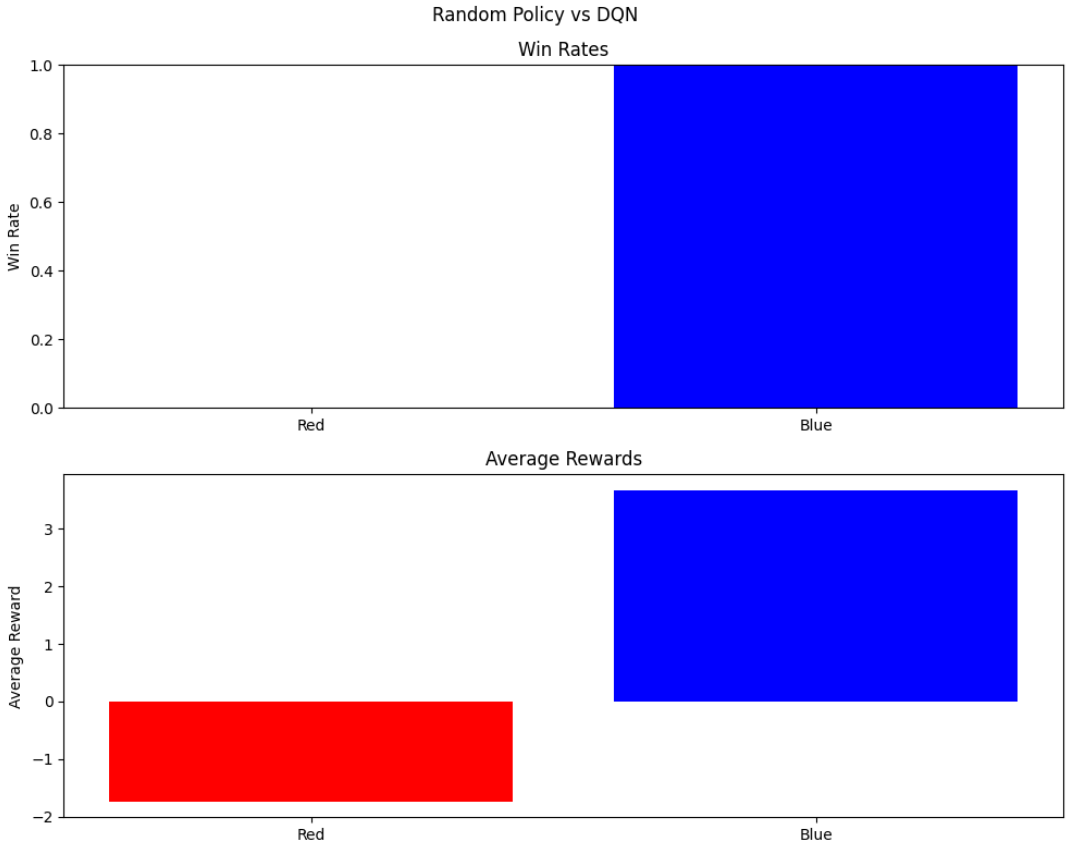
    Đánh giá kết quả với tác tử được cho trước bên đỏ (red), tác tử được huấn luyện trong báo cáo ở bên xanh (blue)

Kết quả của quá trình huấn luyện: Losses và điểm thưởng theo từng episode:

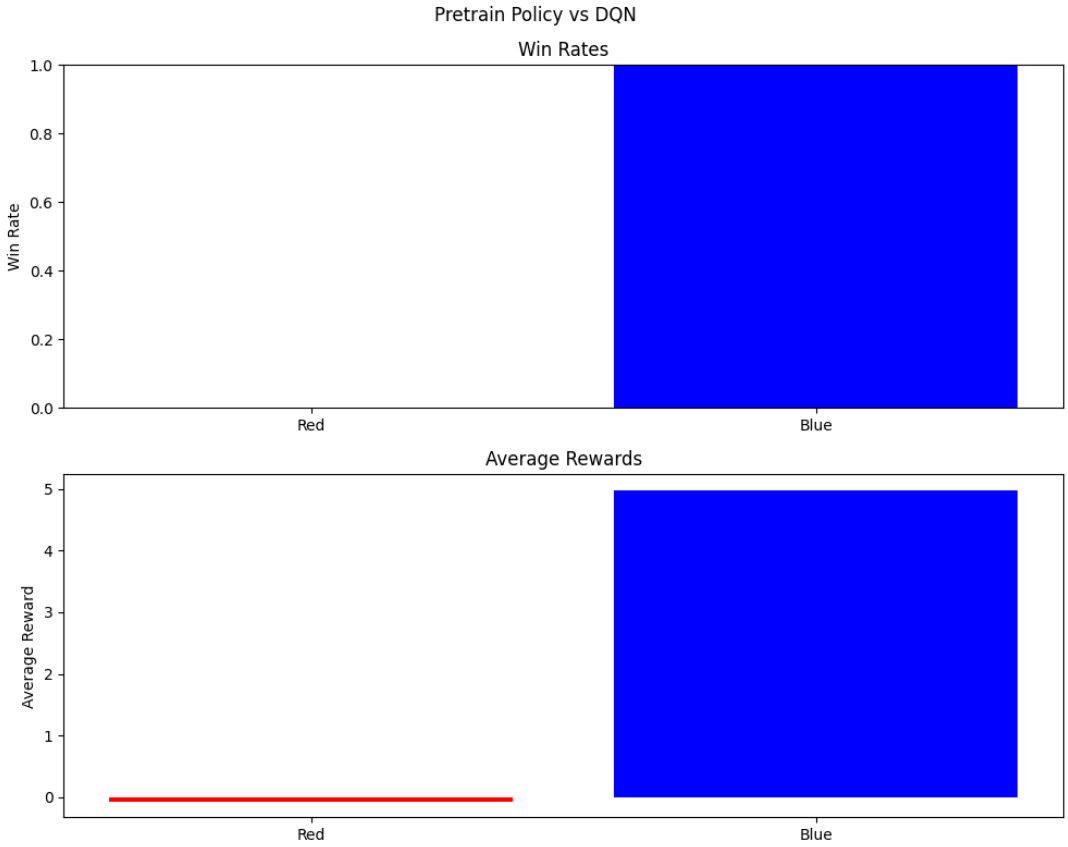


Kết quả của quá trình đánh giá:

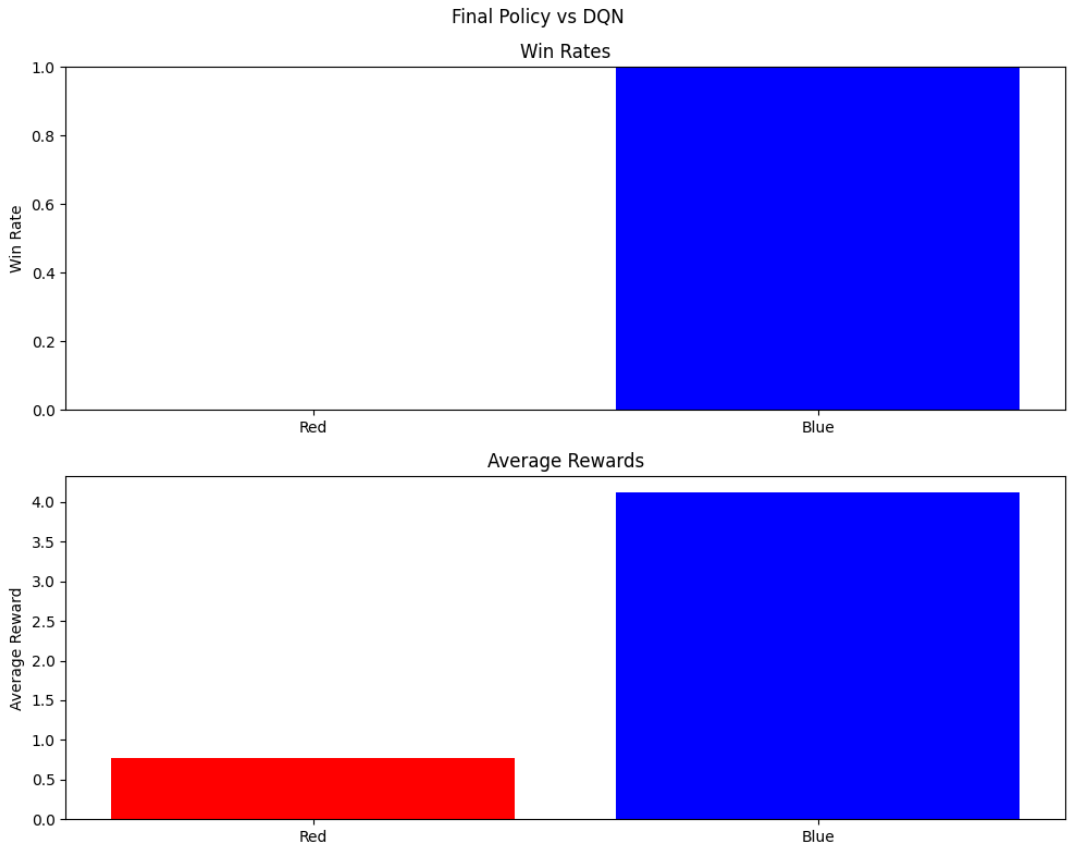
- Đánh giá với tác tử hành động ngẫu nhiên



- Đánh giá với tác tử pretrain-0



- Đánh giá với tác tử new pretrain



b. Parameter tuning discussion

- Epsilon: ϵ_{start} = 1.0, ϵ_{end} = 0.1, ϵ_{decay} = 50: Khuyến khích khám phá mạnh ở giai đoạn đầu, giảm dần và chuyển sang khai thác ở giai đoạn sau.
- **Kích thước batch**: BATCH_SIZE = 128 đảm bảo dữ liệu huấn luyện đủ, đa dạng, cân bằng giữa tốc độ tính toán và bộ nhớ.
- **Hệ số chiết khấu:** γ = 0.9 cân bằng giữa phần thưởng ngắn hạn và dài hạn, vì môi trường khá đơn giản nên không có nhiều mục tiêu dài hạn.
- **Tốc độ học:** α = 1e-4 giúp cập nhật mạng ổn định, tránh dao động hơn so với 1e-3, tuy nhiên tốc độ huấn luyện sẽ chậm hơn.
- **Hệ số cập nhật mạng mục tiêu**: τ = 0.005 cập nhật mạng mục tiêu mượt mà, tránh thay đổi đột ngột vì quá trình cập nhật diễn ra liên tục sau mỗi hành động của tác tử.
- **Bộ nhớ kinh nghiệm**: Bộ nhớ với dung lượng lớn 10000 transitions, đảm bảo lưu trữ đủ mẫu từ nhiều trạng thái.

- **Số tập huấn luyện**: Chỉ cần 4 tập là đủ để mô hình cho ra kết quả tốt vì quá trình cập nhật mạng diễn ra liên tục và nhiều lần trong một tập, cần tăng khi môi trường phức tạp hơn.

c. Thời gian huấn luyện

   Mất khoảng 1h30p để huấn luyện xong 4 tập, vừa đủ để cho ra kết quả tốt trong bài toán này, nếu huấn luyện thêm có thể dẫn tới overfit và học ra những chiến thuật không hiệu quả.

5. Conclusion