ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

-----***-----

BÁO CÁO FINAL PROJECT REINFORCEMENT LEARNING

ĐỀ TÀI

Train Tác Tử Xanh Sử Dụng Qmix

Học phần: Học Tăng Cường & Lập Kế Hoạch

Giảng Viên Hướng Dẫn: Tạ Việt Cường

Ngành: Trí tuệ nhân tạo - QH-2022-I/CQ-A-AI

Nhóm: 26

Tên thành viên	Mã sinh viên
Chu Thân Nhất	22022578

Link code: Github

I. Tổng quan về bài toán

1. Giới thiệu chung và mục đích của bài toán

Học tăng cường (“Reinforcement Learning” - RL) là một trong những nhánh quan trọng nhất của trí tuệ nhân tạo (AI), nhấn mạnh đến khả năng học tập tự động thông qua việc tương tác với môi trường. Trong bài toán này, tôi tìm hiểu và ứng dụng một thuật toán RL tiên tiến – QMIX, nhằm giải quyết bài toán huấn luyện đội quân xanh trong môi trường chiến đấu đa tác tử.

Mục đích của bài toán là thiết kế một chương trình huấn luyện đội quân xanh sao cho có thể đánh bại đội quân đỏ với tỷ lệ thắng cao nhất, trong khi đội quân đỏ sử dụng chiến thuật ngẫu nhiên đã cài đặt sẵn. Kết quả bài toán này không chỉ minh chứng hiệu quả của thuật toán QMIX trong bài toán RL đa tác tử mà còn giúp định hướng ứng dụng AI vào các bài toán phức tạp trong thực tế.

Cuối cùng mặc dù tôi đã dành nhiều tuần để thực hiện dự án này, tôi vẫn chưa hoàn thiện được và vì vậy không thể tiến hành các bước đánh giá cuối cùng. Trong quá trình thực hiện, tôi đã thử nghiệm nhiều lần cấu hình blue team khác nhau nhưng không thành công

2. Mô tả bài toán

Bài toán được thực hiện trong môi trường chiến đấu đa tác tử (“battle_v4”) thuộc thư viện Magent2. Trong môi trường này, hai đội quân xanh và đỏ được đặt trong một khu vực chiến đấu, mỗi đội có nhiều tác tử (“agents”) di chuyển và tấn công theo chiến lược đã cài đặt.

Trong khuôn khổ huấn luyện, đội quân xanh sử dụng thuật toán QMIX để tối đa hóa chiến lược phối hợp giữa các tác tử trong đội. Trong khi đó, đội quân đỏ hoạt động với chiến thuật ngẫu nhiên, tức không có cơ chế phối hợp hay học tập.

Môi trường được cài đặt với các thông số:

- **Số bước huấn luyện (steps):** 300 bước mỗi trận.
- **Số trận (“games”):** 30 trận.
- **Số mô hình khác nhau phải đối mặt:** 3 mô hình red (random, red.pt, red_final.pt).
- **Thời gian tối đa:** 2 giờ.
- **Kết quả đầu ra:** Một video minh họa tác chiến giữa hai đội.

Nhiệm vụ chính của bài toán là đạt được tỷ lệ thắng cao nhất cho đội quân xanh, đòi hỏi cần tối ưu hóa chiến lược phối hợp nhanh chóng và hiệu quả trong khoảng thời gian giới hạn.

II. Thiết lập và Cấu Hình

Dự án sử dụng các thư viện chính như PyTorch, numpy, OpenCV và magent2. Cụ thể, PyTorch được sử dụng để xây dựng và huấn luyện các mô hình mạng neural. Môi trường phát triển bao gồm việc cài đặt các thư viện cần thiết và thiết lập thiết bị (GPU/CPU) để đảm bảo hiệu suất huấn luyện tối ưu.

1. Thiết lập

Mô hình được chạy trên colab với GPU T4, clone github từ gianbang về để triển khai tiếp.

```
%%capture
!git clone https://github.com/giangbang/RL-final-project-AIT-3007
%cd RL-final-project-AIT-3007
!pip install -r requirements.txt
```

2. Một số thư viện sử dụng

- Trong chương trình main.py, tôi có sử dụng một số thư viện như sau:

```
from magent2.environments import battle_v4
import os
import cv2
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torch_model import QNetwork, QMixer, ReplayBuffer
import time
```

- Trong torch_model.py, tôi sử dụng các thư viện sau:

```
import torch
import torch.nn as nn
import numpy as np
from collections import deque
import random
```

3. Cầu hình môi trường huấn luyện

- Môi trường huấn luyện cần được cấu hình chi tiết để đảm bảo rằng các agent có thể học và tương tác một cách hiệu quả. Tôi đã thiết lập môi trường battle_v4 với các tham số phù hợp để mô phỏng các trận chiến.

```python
if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    env = battle_v4.env(map_size=45, render_mode="rgb_array")
    os.makedirs("video", exist_ok=True)
```

- Giải thích:
  - map_size=45: Kích thước của bản đồ chiến đấu. Kích thước này đủ lớn để các agent có không gian di chuyển và thực hiện các chiến thuật.
  - render_mode="rgb_array": Chế độ hiển thị của môi trường. Tôi sử dụng chế độ này để có thể lưu lại các khung hình dưới dạng mảng RGB và sau đó kết hợp chúng thành video.
  -

4. Khởi tạo các thành phần cần thiết

- Khởi tạo mạng Neural
  - Mỗi agent trong môi trường sẽ sử dụng một mạng neural để quyết định hành động dựa trên trạng thái hiện tại của môi trường. Tôi đã khởi tạo các mạng neural và chuyển chúng sang thiết bị đã được cấu hình (GPU hoặc CPU).

```python
agent_networks = [QNetwork(observation_shape, action_space).to(device) for _ in range(n_agents)]
qmixer = QMixer(n_agents, state_dim).to(device)
```

- Giải thích:
  - QNetwork: Mạng neural để các agent dự đoán Q-values cho các hành động có thể thực hiện.
  - QMixer: Mạng neural để kết hợp các Q-values từ các agent khác nhau.

5. Khởi tạo Replay Buffer

- Replay Buffer được sử dụng để lưu trữ các trải nghiệm của các agent trong quá trình huấn luyện. Các trải nghiệm này sau đó được sử dụng để huấn luyện lại các mạng neural.

```python
replay_buffer = ReplayBuffer(capacity=10000, observation_shape=observation_shape)
```

- capacity=10000: Dung lượng của Replay Buffer, tức là số lượng trải nghiệm tối đa mà nó có thể lưu trữ.

III. Phân tích cách huấn luyện

Tệp main.py là trung tâm điều khiển của dự án, chịu trách nhiệm khởi tạo môi trường, huấn luyện các agent, và lưu trữ kết quả huấn luyện. Trong phần này, chúng ta sẽ đi sâu vào từng đoạn mã và giải thích chi tiết các bước thực hiện.

1. Khởi tạo môi trường và các thư mục cần thiết

Đầu tiên, ta khởi tạo môi trường chiến đấu sử dụng `battle_v4` từ thư viện `magent2`. Môi trường này sẽ mô phỏng các trận chiến giữa các agent. Chúng ta cũng tạo thư mục `video` để lưu trữ các video ghi lại quá trình huấn luyện của các agent.

```python
from magent2.environments import battle_v4
import os
import cv2
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torch_model import QNetwork, QMixer, ReplayBuffer
import time

def train_qmix(env, n_episodes=30, batch_size=32, gamma=0.99, max_steps=300, max_train_time=7200):
    n_agents = 81
    action_space = env.action_space("blue_0").n
    observation_shape = env.observation_space("blue_0").shape
    state_shape = env.state_space.shape
    state_dim = np.prod(state_shape)

    agent_networks = [QNetwork(observation_shape, action_space).to(device) for _ in range(n_agents)]
    qmixer = QMixer(n_agents, state_dim).to(device)

    agent_optimizers = [optim.Adam(network.parameters(), lr=0.001) for network in agent_networks]
    qmixer_optimizer = optim.Adam(qmixer.parameters(), lr=0.001)

    replay_buffer = ReplayBuffer(capacity=10000, observation_shape=observation_shape)

    start_time = time.time()

```

Các mạng neural (QNetwork) cho từng agent và mạng QMixer được khởi tạo và chuyển sang thiết bị đã được xác định (GPU hoặc CPU). Replay Buffer được sử dụng để lưu trữ các trải nghiệm của các agent trong quá trình huấn luyện.

2. Vòng lặp huấn luyện qua các episode

Trong mỗi episode, môi trường được khởi tạo lại và các agent bắt đầu tương tác với môi trường. Trạng thái, hành động và kết quả của các agent được lưu trữ trong Replay Buffer.

```
    start_time = time.time()

    for episode in range(n_episodes):
        env.reset()
        episode_reward = 0
        frames = []
        step_count = 0

        for agent in env.agent_iter():
            observation, reward, termination, truncation, info = env.last()

            if termination or truncation:
                action = None
            else:
                agent_handle = agent.split("_")[0]
                agent_id = int(agent.split("_")[1])

                if agent_handle == "blue":
                    with torch.no_grad():
                        q_values = agent_networks[agent_id](observation)
                        action = torch.argmax(q_values).item()
                else:
                    action = env.action_space(agent).sample()

            env.step(action)
```

```
            if agent == "blue_0":
                frames.append(env.render())
                episode_reward += reward

                if not termination and not truncation:
                    next_observation = env.observe(agent)
                    current_state = env.state()
                    next_state = env.state()

                    replay_buffer.push(
                        current_state,
                        action,
                        reward,
                        next_state,
                        0.0 if termination or truncation else 1.0
                    )

                step_count += 1
                if step_count >= max_steps:
                    break
```

- `env.reset()`: Khởi tạo lại môi trường cho mỗi episode.
- `env.agent_iter()`: Lặp qua từng agent trong môi trường.

- `env.last()`: Lấy thông tin cuối cùng của agent bao gồm quan sát, phần thưởng, trạng thái kết thúc và thông tin khác.
- `env.step(action)`: Thực hiện hành động và cập nhật trạng thái môi trường.
- `replay_buffer.push()`: Lưu trữ trải nghiệm vào Replay Buffer.

3. Huấn luyện mạng neural dựa trên Replay Buffer

Khi Replay Buffer đã chứa đủ các trải nghiệm, chúng được sử dụng để huấn luyện lại các mạng neural. Điều này giúp các agent cải thiện chính sách hành động của mình dựa trên các trải nghiệm đã thu thập được.

```
72
73      if len(replay_buffer) >= batch_size:
74          states, actions, rewards, next_states, dones = replay_buffer.sample(batch_size)
75
76          with torch.no_grad():
77              next_q_values = torch.stack([network(next_states) for network in agent_networks])
78              next_actions = next_q_values.max(2)[1]
79              next_q = qmixer(next_q_values, next_states)
80              target_q = rewards + gamma * next_q * (1 - dones)
81
82          current_q_values = torch.stack([network(states) for network in agent_networks])
83          current_q = qmixer(current_q_values, states)
84
85          loss = nn.MSELoss()(current_q, target_q)
86
87          for optimizer in agent_optimizers:
88              optimizer.zero_grad()
89          qmixer_optimizer.zero_grad()
90
91          loss.backward()
92
93          for optimizer in agent_optimizers:
94              optimizer.step()
95          qmixer_optimizer.step()
```

- `replay_buffer.sample(batch_size)`: Lấy mẫu ngẫu nhiên từ Replay Buffer.
- `torch.no_grad()`: Khối mã không tính toán gradient, giúp tăng tốc tính toán.
- `loss.backward()`: Tính toán gradient của loss.
- `optimizer.step()`: Cập nhật các tham số của mạng neural dựa trên gradient.

4. Quản lý thời gian huấn luyện và lưu kết quả

Quá trình huấn luyện được giám sát chặt chẽ để đảm bảo không vượt quá giới hạn về thời gian và số bước. Kết quả của episode cuối cùng được lưu lại dưới dạng video để phân tích sau này.

```
        if episode == n_episodes - 1:
            height, width, _ = frames[0].shape
            out = cv2.VideoWriter(
                os.path.join("video", f"qmix_trained.mp4"),
                cv2.VideoWriter_fourcc(*"mp4v"),
                35,
                (width, height),
            )
            for frame in frames:
                frame_bgr = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
                out.write(frame_bgr)
            out.release()

        elapsed_time = time.time() - start_time
        if elapsed_time >= max_train_time:
            print(f"Training stopped early due to time limit. Elapsed time: {elapsed_time:.2f} seconds")
            break

```

- `time.time() - start_time`: Tính toán thời gian đã trôi qua từ khi bắt đầu huấn luyện.
- `cv2.VideoWriter`: Lưu lại các khung hình thành video để phân tích sau này.
- `if elapsed_time >= max_train_time`: Kiểm tra xem thời gian huấn luyện có vượt quá giới hạn hay không.

5. Phân tích torch_mode.py

Tệp `torch_model.py` chứa định nghĩa các lớp mạng neural và một lớp Replay Buffer, các thành phần này đóng vai trò quan trọng trong quá trình huấn luyện các agent. Trong phần này, chúng ta sẽ đi sâu vào từng lớp và giải thích chi tiết các bước thực hiện và lý do chọn các kiến trúc này.

A. Lớp QNetwork

Lớp `QNetwork` định nghĩa một mạng neural convolutional (CNN) để xử lý các quan sát từ môi trường và dự đoán Q-values cho các hành động có thể thực hiện. Mạng này bao gồm các lớp convolutional và fully connected để trích xuất đặc trưng và dự đoán Q-values.

- `self.cnn`: Định nghĩa một chuỗi các lớp convolutional để trích xuất đặc trưng từ các quan sát. Các lớp bao gồm:
    - `nn.Conv2d`: Lớp convolutional với 32 và 64 filter, kích thước kernel 3x3, stride 1 và padding 1.
    - `nn.ReLU`: Hàm kích hoạt ReLU để giới thiệu tính phi tuyến tính.
    - `nn.MaxPool2d`: Lớp max pooling để giảm kích thước không gian của các đặc trưng.
    - `nn.AdaptiveAvgPool2d`: Lớp pooling để chuyển đổi kích thước không gian thành kích thước cố định.
- `self.flatten_dim`: Tính toán kích thước đầu vào của mạng fully connected dựa trên đầu ra của mạng CNN.

- `self.network`: Định nghĩa mạng fully connected với một lớp ẩn có 256 nút và một lớp đầu ra với số nút bằng số hành động.

```
class QNetwork(nn.Module):
    def __init__(self, observation_shape, action_shape):
        super().__init__()
        self.input_shape = observation_shape

        self.cnn = nn.Sequential(
            nn.Conv2d(observation_shape[-1], 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1))
        ).to(device)

        with torch.no_grad():
            dummy_input = torch.randn(1, observation_shape[-1], observation_shape[0], observation_shape[1]).to(device)
            dummy_output = self.cnn(dummy_input)
            self.flatten_dim = np.prod(dummy_output.shape[1:])

        self.network = nn.Sequential(
            nn.Linear(self.flatten_dim, 256),
            nn.ReLU(),
            nn.Linear(256, action_shape)
        ).to(device)

```

Hàm `forward`

Hàm `forward` xác định cách dữ liệu di chuyển qua mạng neural. Trong hàm này, chúng ta xử lý đầu vào, chuyển qua các lớp CNN và fully connected để dự đoán Q-values.

```
        ).to(device)

    def forward(self, x):
        if isinstance(x, np.ndarray):
            x = torch.FloatTensor(x)
        x = x.to(device)

        if len(x.shape) == 3:
            x = x.unsqueeze(0)
        elif len(x.shape) == 2:
            x = x.view(-1, self.input_shape[0], self.input_shape[1], self.input_shape[2])

        if x.shape[-1] != self.input_shape[-1]:
            x = x.view(-1, self.input_shape[0], self.input_shape[1], self.input_shape[2])

        x = x.permute(0, 3, 1, 2).contiguous()
        x = self.cnn(x)
        x = x.reshape(x.size(0), -1)
        return self.network(x)
```

- Kiểm tra và chuyển đổi đầu vào `x` sang kiểu `torch.FloatTensor` nếu cần.
- Chuyển đổi đầu vào sang thiết bị (GPU hoặc CPU) đã được xác định.
- Điều chỉnh hình dạng của đầu vào sao cho phù hợp với các lớp CNN.

- Sử dụng `permute` để hoán đổi thứ tự các chiều của tensor đầu vào từ (batch_size, height, width, channels) sang (batch_size, channels, height, width).
- Chuyển đầu vào qua các lớp CNN và sau đó qua mạng fully connected để dự đoán Q-values.

B. Lớp QMixer

Lớp `QMixer` chịu trách nhiệm kết hợp Q-values từ các agent khác nhau dựa trên trạng thái toàn cục của môi trường. Mạng này sử dụng embedding của trạng thái và các lớp fully connected để tính toán trọng số cho từng agent.

```
56 class QMixer(nn.Module):
57     def __init__(self, n_agents, state_dim):
58         super(QMixer, self).__init__()
59         self.n_agents = n_agents
60         self.state_dim = state_dim
61 
62         self.state_embedding = nn.Sequential(
63             nn.Linear(self.state_dim, 256),
64             nn.ReLU(),
65             nn.Linear(256, 128)
66         ).to(device)
67 
68         self.mixing_network = nn.Sequential(
69             nn.Linear(128, 64),
70             nn.ReLU(),
71             nn.Linear(64, self.n_agents)
72         ).to(device)
73 
```

- `self.state_embedding`: Mạng fully connected để lấy embedding của trạng thái:
  - `nn.Linear(self.state_dim, 256)`: Lớp fully connected đầu tiên với 256 nút.
  - `nn.ReLU()`: Hàm kích hoạt ReLU.
  - `nn.Linear(256, 128)`: Lớp fully connected thứ hai với 128 nút.
- `self.mixing_network`: Mạng fully connected để tính toán trọng số cho từng agent:
  - `nn.Linear(128, 64)`: Lớp fully connected với 64 nút.
  - `nn.ReLU()`: Hàm kích hoạt ReLU.
  - `nn.Linear(64, self.n_agents)`: Lớp fully connected cuối cùng với số nút bằng số agent.
- Hàm `forward` xác định cách dữ liệu di chuyển qua mạng neural để tính toán Q-values tổng hợp từ Q-values của từng agent.

```python
def forward(self, agent_qs, states):
    batch_size = states.size(0)
    states = states.to(device).reshape(batch_size, -1).float()
    agent_qs = agent_qs.to(device)

    state_embed = self.state_embedding(states)
    weights = torch.abs(self.mixing_network(state_embed))

    if len(agent_qs.shape) == 3:
        agent_qs = agent_qs.permute(1, 0, 2)
        agent_qs = agent_qs.mean(dim=-1)

    weights = weights.view(batch_size, -1)
    mixed_qvals = torch.sum(agent_qs * weights, dim=1)
    return mixed_qvals
```

C. Lớp RepkayBuffer

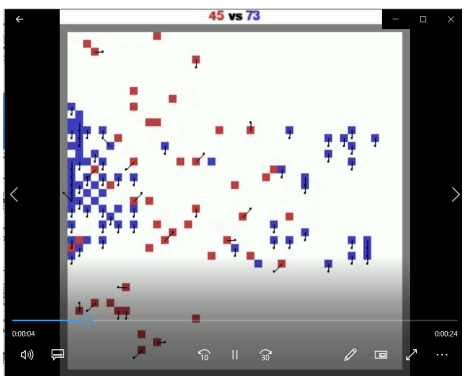
Lớp `ReplayBuffer` được sử dụng để lưu trữ các trải nghiệm (state, action, reward, next_state, done) trong quá trình huấn luyện. Các trải nghiệm này sau đó được lấy mẫu ngẫu nhiên để huấn luyện lại các mạng neural. Điều này giúp làm mịn quá trình huấn luyện và giảm thiểu sự phụ thuộc vào thứ tự của các trải nghiệm.

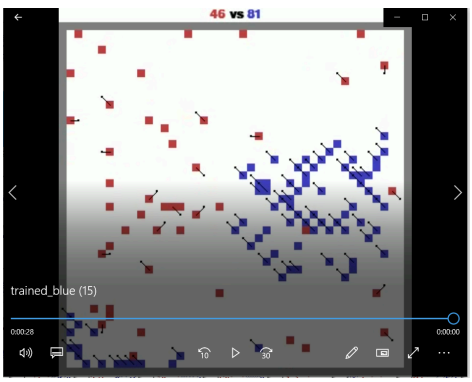
- `__init__`: Khởi tạo Replay Buffer với dung lượng tối đa và hình dạng của các quan sát.
- `push`: Lưu một trải nghiệm vào Replay Buffer.
- `sample`: Lấy mẫu ngẫu nhiên một batch các trải nghiệm từ Replay Buffer để huấn luyện.
- `__len__`: Trả về kích thước hiện tại của Replay Buffer.

```python
class ReplayBuffer:
    def __init__(self, capacity=10000, observation_shape=None):
        self.buffer = deque(maxlen=capacity)
        self.observation_shape = observation_shape

    def push(self, state, action, reward, next_state, done):
        if isinstance(state, np.ndarray):
            state = state.copy()
        if isinstance(next_state, np.ndarray):
            next_state = next_state.copy()

        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)

        state_array = np.array(state)
        next_state_array = np.array(next_state)

        if self.observation_shape is not None:
            if len(state_array.shape) == 2:
                state_array = state_array.reshape(-1, *self.observation_shape)
                next_state_array = next_state_array.reshape(-1, *self.observation_shape)

        return (
            torch.FloatTensor(state_array).to(device),
            torch.LongTensor(np.array(action)).to(device),
            torch.FloatTensor(np.array(reward)).to(device),
            torch.FloatTensor(next_state_array).to(device),
            torch.FloatTensor(np.array(done)).to(device)
        )

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

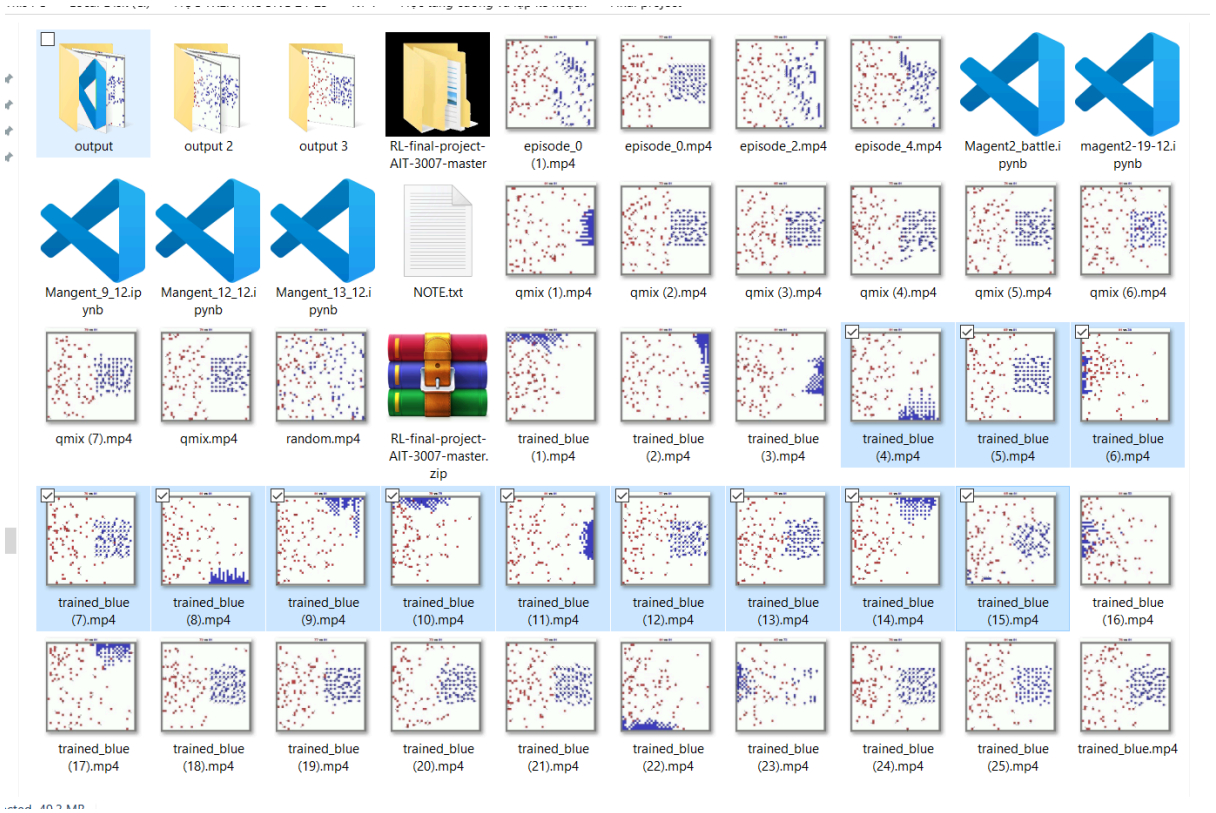
IV. Kết quả và đánh giá

- Dù chưa hoàn thiện nhưng cũng có những kết quả khả quan
-





-



V. Tổng kết

1. Kết luận

- `main.py` cho thấy quá trình khởi tạo môi trường, thiết lập các thành phần cần thiết, và huấn luyện các agent sử dụng thuật toán QMIX. Việc quản lý thời gian huấn luyện và lưu trữ kết quả giúp đảm bảo quá trình huấn luyện diễn ra hiệu quả và có thể phân tích sau này. Mặc dù đã dành nhiều thời gian và công sức, dự án vẫn chưa hoàn thiện và cần thêm nhiều cải tiến để đạt được kết quả mong muốn.
- `torch_model.py` cho thấy sự đa dạng và phức tạp của các lớp mạng neural và Replay Buffer. Các lớp này được thiết kế để tối ưu hóa quá trình huấn luyện các agent trong môi trường chiến đấu. Mặc dù đã dành nhiều thời gian và công sức, dự án vẫn chưa hoàn thiện và cần thêm nhiều cải tiến để đạt được kết quả mong muốn. Việc tích hợp các thuật toán học sâu khác và cải thiện khả năng phối hợp giữa các agent sẽ là hướng phát triển tiếp theo của dự án này.

Khó khăn:

Mặc dù đã dành nhiều tuần để thực hiện dự án này, tôi vẫn chưa hoàn thiện được nó. Một số khó khăn chính gặp phải bao gồm:

1. Hiệu suất và tối ưu hóa: Việc đảm bảo rằng các mạng neural được huấn luyện hiệu quả trên GPU đòi hỏi sự tối ưu hóa cẩn thận. Các vấn đề về bộ nhớ và tốc độ xử lý cũng cần được giải quyết.
2. Quản lý Replay Buffer: Replay Buffer cần được quản lý hiệu quả để lưu trữ và lấy mẫu trải nghiệm một cách ngẫu nhiên, đảm bảo tính ngẫu nhiên và đa dạng của dữ liệu huấn luyện.
3. Phối hợp giữa các agent: Thuật toán QMIX yêu cầu sự phối hợp phức tạp giữa các agent, đòi hỏi sự điều chỉnh và tinh chỉnh cẩn thận các tham số.

Tương lai:

Dự án vẫn còn nhiều tiềm năng để phát triển và cải tiến. Một số hướng đi tương lai có thể bao gồm:

1. Tối ưu hóa thuật toán: Nghiên cứu và áp dụng các thuật toán học sâu tiên tiến hơn để cải thiện hiệu suất và hiệu quả của quá trình huấn luyện.
2. Phát triển môi trường phức tạp hơn: Mở rộng môi trường huấn luyện để bao gồm các tình huống và kịch bản phức tạp hơn, giúp các agent học được các chiến lược phức tạp hơn.
3. Tích hợp học chuyển tiếp (Transfer Learning): Áp dụng học chuyển tiếp để tận dụng các mô hình đã được huấn luyện trước đó, giúp giảm thời gian và tài nguyên cần thiết cho quá trình huấn luyện.

Tổng kết:

- Dự án này nhằm mục đích huấn luyện các agent sử dụng thuật toán QMIX trong môi trường chiến đấu của MAgent2. Tôi đã trải qua nhiều bước từ việc thiết lập môi trường, cấu hình phần cứng, đến việc xây dựng và huấn luyện các mạng neural. Trong quá trình này, các vấn đề như tối ưu hóa hiệu suất, quản lý thời gian huấn luyện và lưu trữ kết quả đều được xem xét kỹ lưỡng.

Mặc dù dự án chưa hoàn thiện, những nỗ lực đã bỏ ra trong quá trình này đã giúp tôi hiểu rõ hơn về các thách thức và cơ hội trong việc huấn luyện các agent sử dụng thuật toán QMIX. Tôi sẽ tiếp tục nỗ lực để hoàn thiện dự án và đạt được các kết quả mong muốn trong tương lai.

Cuối cùng là chút tâm sự: trong quá trình làm thì em nhận thấy đây là một bài toán khó, cần nhiều thời gian hơn nữa, em đã rút ra được nhiều kinh nghiệm sau quá

trình train hơn 3 tuần này. Đã có lúc em nghĩ em không làm được và đúng thật là em không làm được thật, đã có những người bạn bỏ cuộc và em cũng có suy nghĩ như vậy, em thấy hơi thất vọng về bản thân nhưng sau cùng em vẫn nghĩ rằng người bỏ cuộc là người thua cuộc, vậy nên vấn cố làm để viết ra báo cáo này, em sẽ cố gắng hơn trong tương lại.
Cảm ơn thầy.