

# マルチエージェントサッカーゲーム実装ドキュメント

## プロジェクト概要

### 目標

深層強化学習におけるマルチエージェント環境として、簡易サッカーゲームを実装し、複数の学習アルゴリズムを比較・分析する。

### 成果物

- 2対2のサッカー環境（PettingZoo準拠）
- 複数のマルチエージェント学習アルゴリズムの実装
- 学習性能の比較分析と可視化
- 創発行動の観察と分析

## 技術仕様

### 環境要件

- Python: 3.8+
- 実行環境: Google Colab (A100/T4)
- 主要ライブラリ: PyTorch, PettingZoo, Gymnasium, Stable-Baselines3
- RAM: 12-25GB, GPU Memory: 16-40GB

### 環境仕様

### ゲーム環境詳細

```
python
```

```
# 環境パラメータ
FIELD_SIZE = (800, 600) # フィールドサイズ
GOAL_SIZE = (20, 200) # ゴールサイズ
BALL_RADIUS = 10 # ボール半径
PLAYER_RADIUS = 20 # プレイヤー半径
MAX_STEPS = 1000 # 最大ステップ数

# プレイヤー設定
NUM_PLAYERS_PER_TEAM = 2
TEAM_COLORS = ['blue', 'red']
PLAYER_SPEED = 5.0
BALL_SPEED_MULTIPLIER = 1.5

# 物理パラメータ
FRICTION = 0.95 # 摩擦係数
BALL_DECAY = 0.98 # ボール減衰
COLLISION_THRESHOLD = 30 # 衝突判定距離
```

## 状態空間設計

python

# 各エージェントの観測 (28次元)

```
observation_space = {  
    # 自分の状態 (4次元)  
    'self_pos': (x, y),      # 自分の位置  
    'self_vel': (vx, vy),    # 自分の速度  
  
    # ボールの状態 (4次元)  
    'ball_pos': (x, y),      # ボール位置  
    'ball_vel': (vx, vy),    # ボール速度  
  
    # チームメイトの状態 (4次元)  
    'teammate_pos': (x, y),   # チームメイト位置  
    'teammate_vel': (vx, vy), # チームメイト速度  
  
    # 相手チームの状態 (8次元)  
    'opponent1_pos': (x, y),  # 相手1位置  
    'opponent1_vel': (vx, vy), # 相手1速度  
    'opponent2_pos': (x, y),  # 相手2位置  
    'opponent2_vel': (vx, vy), # 相手2速度  
  
    # ゴール情報 (4次元)  
    'own_goal_dist': float,    # 自陣ゴールまでの距離  
    'enemy_goal_dist': float,  # 敵陣ゴールまでの距離  
    'goal_angle_own': float,   # 自陣ゴールへの角度  
    'goal_angle_enemy': float, # 敵陣ゴールへの角度  
  
    # コンテキスト情報 (4次元)  
    'ball_possession': int,     # ボール保持者ID (-1: なし)  
    'time_remaining': float,    # 残り時間 (正規化)  
    'score_diff': int,          # スコア差  
    'last_touch': int,          # 最後にボールに触れたプレイヤー  
}
```

## 行動空間設計

python

# 連続行動空間 (5次元)

```
action_space = {  
    'move_x': float,    # x方向移動 [-1, 1]  
    'move_y': float,    # y方向移動 [-1, 1]  
    'kick_power': float, # キック力 [0, 1]  
    'kick_dir_x': float, # キック方向x [-1, 1]  
    'kick_dir_y': float, # キック方向y [-1, 1]  
}
```

# 離散行動空間 (代替案、9行動)

```
discrete_actions = [  
    'NOOP',    # 何もしない  
    'UP',      # 上移動  
    'DOWN',    # 下移動  
    'LEFT',    # 左移動  
    'RIGHT',   # 右移動  
    'KICK_UP', # 上にキック  
    'KICK_DOWN', # 下にキック  
    'KICK_LEFT', # 左にキック  
    'KICK_RIGHT' # 右にキック  
]
```

## 報酬設計

### 基本報酬構造

python

```

def calculate_reward(self, agent_id, action, prev_state, current_state):
    """
    多目的報酬関数
    """
    reward = 0.0

    # 1. ゴール報酬 (最重要)
    if self.goal_scored:
        if self.goal_scorer_team == self.get_team(agent_id):
            reward += 100.0 # ゴール成功
        else:
            reward -= 100.0 # 失点

    # 2. ボール接触報酬
    if self.ball_touched_by == agent_id:
        reward += 5.0

    # 3. ゴールに向かう報酬
    goal_approach_reward = self.calculate_goal_approach_reward(
        agent_id, prev_state, current_state
    )
    reward += goal_approach_reward * 0.1

    # 4. ボールに向かう報酬
    ball_approach_reward = self.calculate_ball_approach_reward(
        agent_id, prev_state, current_state
    )
    reward += ball_approach_reward * 0.05

    # 5. チームワーク報酬
    team_reward = self.calculate_team_reward(agent_id, current_state)
    reward += team_reward * 0.02

    # 6. ペナルティ
    # - フィールド外ペナルティ
    if self.out_of_bounds(agent_id):
        reward -= 10.0

    # - 膠着状態ペナルティ
    if self.is_stalemate():
        reward -= 0.1

    return reward

def calculate_goal_approach_reward(self, agent_id, prev_state, current_state):
    """ゴールへのアプローチ報酬"""

```

```

enemy_goal_pos = self.get_enemy_goal_position(agent_id)

prev_dist = np.linalg.norm(prev_state['pos'] - enemy_goal_pos)
current_dist = np.linalg.norm(current_state['pos'] - enemy_goal_pos)

return prev_dist - current_dist # 近づく と 正の報酬

def calculate_team_reward(self, agent_id, current_state):
    """チームワーク報酬"""
    teammate_id = self.get_teammate(agent_id)

    # チームメイトとの距離を適度に保つ報酬
    teammate_dist = np.linalg.norm(
        current_state[agent_id]['pos'] - current_state[teammate_id]['pos']
    )

    # 適切な距離（100-200ピクセル）を維持する報酬
    optimal_dist = 150
    dist_penalty = abs(teammate_dist - optimal_dist) / optimal_dist

    return 1.0 - dist_penalty

```

## 実装アーキテクチャ

### ディレクトリ構造

```

multi_agent_soccer/
├── README.md
├── requirements.txt
├── config/
│   ├── __init__.py
│   ├── env_config.py    # 環境設定
│   ├── training_config.py # 学習設定
│   └── experiment_config.py # 実験設定
├── environment/
│   ├── __init__.py
│   ├── soccer_env.py    # メイン環境
│   ├── soccer_env_v2.py # PettingZoo準拠版
│   ├── physics.py       # 物理エンジン
│   ├── renderer.py      # 可視化
│   └── utils.py         # ユーティリティ
├── agents/
│   ├── __init__.py
│   ├── base_agent.py    # 基底エージェント
│   ├── random_agent.py  # ランダムエージェント
│   └── dqn_agent.py     # DQNエージェント

```

```
| ├── ppo_agent.py    # PPOエージェント
| ├── maddpg_agent.py # MADDPGエージェント
| └── qmix_agent.py   # QMIXエージェント
└── training/
    ├── __init__.py
    ├── independent_trainer.py # 独立学習
    ├── self_play_trainer.py  # セルフプレイ
    ├── cooperative_trainer.py # 協調学習
    └── evaluation.py         # 評価
└── utils/
    ├── __init__.py
    ├── logging_utils.py    # ログ管理
    ├── visualization.py    # 可視化ツール
    ├── metrics.py          # 評価指標
    └── replay_buffer.py    # リプレイバッファ
└── experiments/
    ├── __init__.py
    ├── run_baseline.py    # ベースライン実験
    ├── run_independent.py # 独立学習実験
    ├── run_cooperative.py # 協調学習実験
    └── compare_methods.py # 手法比較
└── notebooks/
    ├── environment_test.ipynb # 環境テスト
    ├── agent_visualization.ipynb # エージェント可視化
    └── result_analysis.ipynb  # 結果分析
└── models/
    └── saved_models/      # 保存されたモデル
```

## 核心クラス設計

### 環境クラス（PettingZoo準拠）

```
python
```

```
class SoccerEnvironment(AECEnv):
    """
    PettingZoo準拠のマルチエージェントサッカー環境
    """

    def __init__(self, config: SoccerConfig):
        super().__init__()

        # エージェント設定
        self.possible_agents = [f"player_{i}" for i in range(4)]
        self.agents = self.possible_agents[:]

        # 観測・行動空間
        self._init_spaces()

        # 物理エンジン
        self.physics = PhysicsEngine(config)

        # レンダラー
        self.renderer = SoccerRenderer(config)

        # ゲーム状態
        self.reset()

    def reset(self):
        """環境リセット"""
        self.agents = self.possible_agents[:]
        self._agent_selector = agent_selector(self.agents)
        self.agent_selection = self._agent_selector.next()

        # 初期位置設定
        self._init_player_positions()
        self._init_ball_position()

        # 状態初期化
        self.state = self._get_full_state()
        self.rewards = {agent: 0 for agent in self.agents}
        self.dones = {agent: False for agent in self.agents}
        self.infos = {agent: {} for agent in self.agents}

        return self._get_observations()

    def step(self, action):
        """環境ステップ実行"""
        if self.dones[self.agent_selection]:
            return self._was_done_step(action)
```



```
# 行動実行
self._execute_action(self.agent_selection, action)

# 物理シミュレーション
self.physics.step()

# 報酬計算
self._calculate_rewards()

# 終了判定
self._check_done()

# 次のエージェント選択
self.agent_selection = self._agent_selector.next()

# 累積報酬更新
self._accumulate_rewards()

return self.observe(self.agent_selection)
```

## MADDPGエージェント実装

python

```
class MADDPGAgent:
```

```
    """
```

```
Multi-Agent Deep Deterministic Policy Gradient実装
```

```
    """
```

```
def __init__(self, config: MADDPGConfig):
```

```
    self.config = config
```

```
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    # ネットワーク初期化
```

```
    self.actor = Actor(config.obs_dim, config.action_dim, config.hidden_dims)
```

```
    self.critic = Critic(config.global_obs_dim, config.global_action_dim, config.hidden_dims)
```

```
    self.target_actor = copy.deepcopy(self.actor)
```

```
    self.target_critic = copy.deepcopy(self.critic)
```

```
    # オプティマイザー
```

```
    self.actor_optimizer = torch.optim.Adam(self.actor.parameters(), lr=config.actor_lr)
```

```
    self.critic_optimizer = torch.optim.Adam(self.critic.parameters(), lr=config.critic_lr)
```

```
    # リプレイバッファ
```

```
    self.replay_buffer = ReplayBuffer(config.buffer_size)
```

```
    # ノイズ
```

```
    self.noise = OUNoise(config.action_dim, config.noise_scale)
```

```
def select_action(self, obs, add_noise=True):
```

```
    """行動選択"""
```

```
    obs_tensor = torch.FloatTensor(obs).unsqueeze(0).to(self.device)
```

```
    action = self.actor(obs_tensor).cpu().data.numpy().flatten()
```

```
    if add_noise:
```

```
        action += self.noise.sample()
```

```
        action = np.clip(action, -1, 1)
```

```
    return action
```

```
def update(self, experiences, agent_id, global_obs, global_actions):
```

```
    """エージェント更新"""
```

```
    states, actions, rewards, next_states, dones = experiences
```

```
    # Criticの更新
```

```
    with torch.no_grad():
```

```
        next_actions = self.target_actor(next_states)
```

```
        target_q = self.target_critic(global_obs, global_actions)
```

```
        target_q = rewards + (self.config.gamma * target_q * (1 - dones))
```

```
current_q = self.critic(global_obs, global_actions)
critic_loss = F.mse_loss(current_q, target_q)

self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# Actorの更新
predicted_actions = self.actor(states)
actor_loss = -self.critic(global_obs, global_actions).mean()

self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

# ターゲットネットワーク更新
self._soft_update(self.actor, self.target_actor)
self._soft_update(self.critic, self.target_critic)

return {'critic_loss': critic_loss.item(), 'actor_loss': actor_loss.item()}
```

## 参考論文と理論的背景

### 主要論文

#### 1. Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

論文: "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" (Lowe et al., 2017)

#### 核心アイデア:

- 各エージェントが独自のActor-Criticを持つ
- Criticは全エージェントの観測・行動を利用（中央集権的学習）
- Actorは自身の観測のみ利用（分散実行）

#### 実装のポイント:

python

```
# 中央集権的Critic
critic_input = torch.cat([
    global_observations, # 全エージェントの観測
    global_actions       # 全エージェントの行動
], dim=1)

# 分散Actor
actor_input = local_observation # 自身の観測のみ
```

## 2. QMIX (Monotonic Value Function Factorisation)

論文: "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning" (Rashid et al., 2018)

### 核心アイデア:

- 個別のQ値を混合ネットワークで結合
- 単調性制約により分散実行を保証

### 実装での活用:

```
python
```

```

class QMixNetwork(nn.Module):
    def __init__(self, num_agents, state_dim, hidden_dim):
        super().__init__()
        self.num_agents = num_agents

        # 混合ネットワーク (重み生成)
        self.hyper_w1 = nn.Linear(state_dim, num_agents * hidden_dim)
        self.hyper_w2 = nn.Linear(state_dim, hidden_dim)

        # バイアス生成
        self.hyper_b1 = nn.Linear(state_dim, hidden_dim)
        self.hyper_b2 = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, agent_qs, states):
        batch_size = agent_qs.size(0)

        # 重み生成 (非負制約)
        w1 = torch.abs(self.hyper_w1(states))
        w2 = torch.abs(self.hyper_w2(states))

        # バイアス生成
        b1 = self.hyper_b1(states)
        b2 = self.hyper_b2(states)

        # 混合計算
        hidden = F.elu(torch.bmm(agent_qs.unsqueeze(1), w1.view(batch_size, self.num_agents, -1)).squeeze(1))
        q_tot = torch.bmm(hidden.unsqueeze(1), w2.unsqueeze(2)).squeeze() + b2.squeeze()

        return q_tot

```

### 3. Self-Play and Population-Based Training

論文: "Emergent Complexity via Multi-Agent Competition" (Bansal et al., 2017)

実装アプローチ:

python

```
class SelfPlayTrainer:
    def __init__(self, config):
        self.population = [] # エージェント集団
        self.matchmaking = EloMatchmaking() # Eloレーティングシステム

    def train_step(self):
        # 対戦相手選択
        agent1, agent2 = self.matchmaking.select_opponents()

        # 対戦実行
        results = self.play_match(agent1, agent2)

        # レーティング更新
        self.matchmaking.update_ratings(agent1, agent2, results)

        # 学習実行
        agent1.learn_from_experience(results.experience1)
        agent2.learn_from_experience(results.experience2)
```

## 理論的考慮事項

### 1. Non-Stationarity Problem

**問題:** 複数エージェントが同時学習することで環境が非定常となる

**対策:**

- Experience Replay with Importance Sampling
- Target Network の慎重な更新
- 相手の過去のポリシーとの対戦

### 2. Credit Assignment Problem

**問題:** チーム報酬をどう個々のエージェントに割り当てるか

**対策:**

- Difference Rewards
- Counterfactual Multi-Agent Policy Gradients
- Attention-based Credit Assignment

### 3. Exploration vs Exploitation in Multi-Agent Settings

**課題:** 協調的探索の実現

**アプローチ:**

python

```
class MultiAgentExploration:
    def __init__(self, num_agents, exploration_config):
        # 各エージェント用のノイズ
        self.noise_processes = [
            OUNoise(action_dim, exploration_config.noise_scale)
            for _ in range(num_agents)
        ]

        # 協調的探索フラグ
        self.coordinated_exploration = exploration_config.coordinated

    def get_exploration_actions(self, base_actions, step):
        if self.coordinated_exploration:
            # チーム全体で探索方向を決定
            team_exploration_direction = self._sample_team_direction()
            noisy_actions = [
                action + team_exploration_direction * noise.sample()
                for action, noise in zip(base_actions, self.noise_processes)
            ]
        else:
            # 独立探索
            noisy_actions = [
                action + noise.sample()
                for action, noise in zip(base_actions, self.noise_processes)
            ]

        return noisy_actions
```

## 段階的実装計画

### Phase 1: 基礎環境実装（1週間）

#### Day 1-2: 環境コア実装

python

## # 優先実装項目

1. SoccerEnvironment基底クラス
2. 基本的な物理シミュレーション
3. 単純な可視化機能
4. ランダムエージェントでの動作確認

## # 成果物

- environment/soccer\_env.py
- environment/physics.py
- environment/renderer.py
- experiments/test\_random\_agents.py

## Day 3-4: 観測・行動空間の実装

python

### # 実装内容

1. 観測空間の詳細設計
2. 行動空間の実装（連続・離散両対応）
3. 報酬関数の基本実装
4. PettingZoo準拠への変換

### # 成果物

- environment/soccer\_env\_v2.py (PettingZoo版)
- config/env\_config.py
- notebooks/environment\_test.ipynb

## Day 5-7: 単一エージェント学習

python

### # 実装内容

1. DQNエージェントの実装
2. PPOエージェントの実装
3. 基本的な学習ループ
4. 学習曲線の可視化

### # 成果物

- agents/dqn\_agent.py
- agents/ppo\_agent.py
- training/single\_agent\_trainer.py
- experiments/run\_single\_agent.py

## Phase 2: マルチエージェント実装（1週間）

### Day 8-10: Independent Learning



python

#### # 実装内容

1. 複数エージェントの独立学習
2. 環境の同期実行機能
3. エージェント間の相互作用実装
4. 基本的な性能評価

#### # 成果物

- training/independent\_trainer.py
- experiments/run\_independent.py
- utils/metrics.py

## Day 11-12: Self-Play Implementation

python

#### # 実装内容

1. セルフプレイ機能の実装
2. モデルの定期的保存・読み込み
3. 対戦履歴の管理
4. Eloレーティングシステム（オプション）

#### # 成果物

- training/self\_play\_trainer.py
- utils/matchmaking.py
- experiments/run\_self\_play.py

## Day 13-14: デバッグと調整

python

#### # 実装内容

1. 学習の安定性確認
2. ハイパーパラメータ調整
3. 可視化機能の充実
4. ログ機能の実装

#### # 成果物

- utils/logging\_utils.py
- utils/visualization.py
- config/training\_config.py

## Phase 3: 発展的手法実装（1週間）

### Day 15-17: MADDPG実装

python

## # 実装内容

1. MADDPGアルゴリズムの実装
2. 中央集権的Criticの実装
3. Experience Replayの実装
4. ソフトアップデート機構

## # 核心実装ポイント

```
class MADDPGTrainer:
    def __init__(self, config):
        self.agents = [MADDPGAgent(config) for _ in range(4)]
        self.replay_buffer = SharedReplayBuffer(config.buffer_size)

    def train_step(self):
        # 全エージェントの経験を収集
        experiences = self.replay_buffer.sample(config.batch_size)

        # 各エージェントを更新
        for i, agent in enumerate(self.agents):
            # グローバル情報を構築
            global_obs = self.build_global_obs(experiences)
            global_actions = self.build_global_actions(experiences)

            # エージェント更新
            agent.update(experiences, i, global_obs, global_actions)

# 成果物
- agents/maddpg_agent.py
- training/maddpg_trainer.py
- experiments/run_maddpg.py
```

## Day 18-19: 協調学習手法

python

## # 実装内容

1. QMIXアルゴリズムの実装
2. チーム報酬の分配機構
3. 協調的探索の実装
4. 創発行動の観察機能

## # 成果物

- agents/qmix\_agent.py
- training/cooperative\_trainer.py
- utils/emergent\_behavior\_analyzer.py

## Day 20-21: 実験設計と実行

python

## # 実験項目

```
experiments = {  
    'baseline': 'Random vs Random',  
    'independent_dqn': 'Independent DQN Learning',  
    'independent_ppo': 'Independent PPO Learning',  
    'self_play': 'Self-Play Training',  
    'maddpg': 'MADDPG Cooperative Learning',  
    'qmix': 'QMIX Team Learning',  
    'reward_ablation': 'Different Reward Designs',  
    'observation_ablation': 'Different Observation Spaces'  
}
```

## # 成果物

- experiments/run\_all\_experiments.py
- experiments/compare\_methods.py
- config/experiment\_config.py

## Phase 4: 分析・レポート（1週間）

### Day 22-24: 詳細分析

python

## # 分析項目

1. 学習曲線の比較
2. 創発行動の質的分析
3. チームワークの定量評価
4. 戦略の多様性分析

## # 評価指標

```
metrics = {  
    'performance': ['win_rate', 'goal_difference', 'episode_length'],  
    'teamwork': ['pass_success_rate', 'team_coordination_index'],  
    'exploration': ['state_coverage', 'action_diversity'],  
    'stability': ['learning_variance', 'policy_drift']  
}
```

## # 成果物

- notebooks/result\_analysis.ipynb
- utils/advanced\_metrics.py
- visualization/behavior\_analysis.py

## Day 25-26: 可視化・動画作成

python

## # 可視化内容

1. 学習プロセスの動画
2. 戦略進化の可視化
3. チーム連携の分析図
4. 創発行動のハイライト

## # 成果物

- utils/video\_generator.py
- visualization/strategy\_evolution.py
- notebooks/behavior\_visualization.ipynb

## Day 27-28: レポート作成

markdown

# レポート構成

1. Introduction

- Problem Statement
- Related Work

2. Methodology

- Environment Design
- Algorithm Implementation
- Experimental Setup

3. Results

- Quantitative Analysis
- Qualitative Behavior Analysis
- Ablation Studies

4. Discussion

- Emergent Behaviors
- Cooperation vs Competition
- Future Directions

5. Conclusion

重要な注意点とベストプラクティス

1. 学習安定性の確保

Hyperparameter Sensitivity

python

```

# 推奨ハイパーパラメータ
MADDPG_CONFIG = {
    'actor_lr': 1e-4,      # Actorの学習率 (低めに設定)
    'critic_lr': 1e-3,     # Criticの学習率
    'gamma': 0.95,        # 割引率
    'tau': 0.01,          # ソフトアップデート率
    'batch_size': 256,     # バッチサイズ
    'buffer_size': 1e6,    # リプレイバッファサイズ
    'noise_scale': 0.1,    # 探索ノイズ
    'noise_decay': 0.9999, # ノイズ減衰率
}

# 学習率スケジューリング
class LearningRateScheduler:
    def __init__(self, optimizer, config):
        self.optimizer = optimizer
        self.initial_lr = config.learning_rate
        self.decay_rate = config.lr_decay
        self.step_count = 0

    def step(self):
        self.step_count += 1
        new_lr = self.initial_lr * (self.decay_rate ** self.step_count)
        for param_group in self.optimizer.param_groups:
            param_group['lr'] = new_lr

```

## Gradient Clipping

```

python

def update_agent(self, experiences):
    # ... 損失計算 ...

    # 勾配クリッピング (重要)
    torch.nn.utils.clip_grad_norm_(self.actor.parameters(), max_norm=0.5)
    torch.nn.utils.clip_grad_norm_(self.critic.parameters(), max_norm=0.5)

    self.actor_optimizer.step()
    self.critic_optimizer.step()

```

## 2. Non-stationarity対策

### Experience Replay Strategy

```

python

```

```
class PrioritizedReplayBuffer:
```

```
    def __init__(self, capacity, alpha=0.6):  
        self.capacity = capacity  
        self.alpha = alpha # 優先度の重み  
        self.beta = 0.4 # Importance Sampling重み  
        self.beta_increment = 0.001
```

```
    def sample(self, batch_size):  
        # TD誤差に基づく優先サンプリング  
        priorities = np.array([abs(td_error) + 1e-6 for td_error in self.td_errors])  
        probabilities = priorities ** self.alpha  
        probabilities /= probabilities.sum()  
  
        indices = np.random.choice(len(self.buffer), batch_size, p=probabilities)  
  
        # Importance Sampling重み計算  
        weights = (len(self.buffer) * probabilities[indices]) ** (-self.beta)  
        weights /= weights.max()  
  
        self.beta = min(1.0, self.beta + self.beta_increment)  
  
        return indices, weights, [self.buffer[i] for i in indices]
```

## Opponent Modeling

python

```
class OpponentModel:
    """相手の行動パターンを学習"""

    def __init__(self, obs_dim, action_dim):
        self.model = MLPNetwork(obs_dim, action_dim)
        self.optimizer = torch.optim.Adam(self.model.parameters())

    def update(self, opponent_obs, opponent_actions):
        """相手の行動を予測するモデルを更新"""
        predicted_actions = self.model(opponent_obs)
        loss = F.mse_loss(predicted_actions, opponent_actions)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def predict_action(self, opponent_obs):
        """相手の次の行動を予測"""
        with torch.no_grad():
            return self.model(opponent_obs)
```

### 3. デバッグ・監視機能

#### 詳細ログ機能

```
python
```



```
class DetailedLogger:
    def __init__(self, log_dir):
        self.writer = SummaryWriter(log_dir)
        self.metrics_history = defaultdict(list)

    def log_training_step(self, step, metrics):
        """学習ステップの詳細ログ"""
        for key, value in metrics.items():
            self.writer.add_scalar(f'Training/{key}', value, step)
            self.metrics_history[key].append(value)

        # 異常値検出
        if self.detect_anomaly(metrics):
            logging.warning(f'Anomaly detected at step {step}: {metrics}')

    def log_episode(self, episode, episode_metrics):
        """エピソード単位のログ"""
        # ゲーム統計
        self.writer.add_scalar('Episode/Length', episode_metrics['length'], episode)
        self.writer.add_scalar('Episode/TotalReward', episode_metrics['total_reward'], episode)
        self.writer.add_scalar('Episode/Goals', episode_metrics['goals'], episode)

        # エージェント別統計
        for agent_id, agent_metrics in episode_metrics['agents'].items():
            self.writer.add_scalar(f'Agent_{agent_id}/Reward', agent_metrics['reward'], episode)
            self.writer.add_scalar(f'Agent_{agent_id}/BallTouches', agent_metrics['ball_touches'], episode)

    def detect_anomaly(self, metrics):
        """異常値検出"""
        for key, value in metrics.items():
            if len(self.metrics_history[key]) > 100:
                recent_mean = np.mean(self.metrics_history[key][-100:])
                recent_std = np.std(self.metrics_history[key][-100:])

                if abs(value - recent_mean) > 3 * recent_std:
                    return True
        return False
```

## 可視化機能

python

```

class BehaviorVisualizer:
    def __init__(self, env):
        self.env = env

    def visualize_heatmap(self, agent_positions, save_path):
        """エージェントの位置ヒートマップ"""
        plt.figure(figsize=(12, 8))

        for agent_id, positions in agent_positions.items():
            x_coords = [pos[0] for pos in positions]
            y_coords = [pos[1] for pos in positions]

            plt.hexbin(x_coords, y_coords, gridsize=30, alpha=0.7, label=f'Agent {agent_id}')

        plt.xlabel('X Position')
        plt.ylabel('Y Position')
        plt.title('Agent Position Heatmap')
        plt.legend()
        plt.savefig(save_path)
        plt.close()

    def analyze_team_coordination(self, team_data):
        """チーム連携の分析"""
        # チームメイト間距離の時系列
        distances = []

        for step_data in team_data:
            dist = np.linalg.norm(
                np.array(step_data['agent_0']['pos']) -
                np.array(step_data['agent_1']['pos'])
            )
            distances.append(dist)

        # 統計量計算
        coordination_metrics = {
            'mean_distance': np.mean(distances),
            'distance_variance': np.var(distances),
            'coordination_index': self._calculate_coordination_index(team_data)
        }

        return coordination_metrics

```

## 4. パフォーマンス最適化

### 効率的な実装

python

# GPU使用の最適化

class EfficientAgent:

def \_\_init\_\_(self, config):

self.device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# モデルをGPUに移動

self.actor.to(self.device)

self.critic.to(self.device)

# 混合精度学習 (メモリ節約)

self.scaler = GradScaler()

def train\_step(self, experiences):

with autocast(): # 自動混合精度

# ... 前向き計算 ...

loss = self.calculate\_loss(experiences)

# スケール付き逆伝播

self.scaler.scale(loss).backward()

self.scaler.step(self.optimizer)

self.scaler.update()

# バッチ処理の効率化

def batch\_process\_environments(envs, actions):

"""複数環境の並列処理"""

with ThreadPoolExecutor(max\_workers=len(envs)) as executor:

futures = [

executor.submit(env.step, action)

for env, action in zip(envs, actions)

]

results = [future.result() for future in futures]

return results

## 5. 実験再現性の確保

### 再現性設定

python

```
def set_reproducibility(seed=42):
    """実験の再現性を確保"""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # 決定的アルゴリズムの使用
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # 環境変数設定
    os.environ['PYTHONHASHSEED'] = str(seed)

# 設定の保存
import json
def save_config(config, path):
    """実験設定を保存"""
    config_dict = {
        'environment': config.env_config.__dict__,
        'training': config.training_config.__dict__,
        'agents': config.agent_config.__dict__,
        'experiment': config.experiment_config.__dict__
    }

    with open(path, 'w') as f:
        json.dump(config_dict, f, indent=2)
```

## 評価指標と分析手法

### 定量的評価指標

#### 1. パフォーマンス指標

python

```
class PerformanceMetrics:
    def __init__(self):
        self.metrics = {
            'win_rate': [], # 勝率
            'goal_difference': [], # 得失点差
            'episode_length': [], # エピソード長
            'goals_per_episode': [], # エピソード当たりゴール数
            'ball_possession': [], # ボール保持率
            'pass_success_rate': [], # パス成功率
        }

    def calculate_team_performance(self, episode_data):
        """チームパフォーマンス計算"""
        team_a_goals = episode_data['goals']['team_a']
        team_b_goals = episode_data['goals']['team_b']

        # 勝利判定
        if team_a_goals > team_b_goals:
            winner = 'team_a'
        elif team_b_goals > team_a_goals:
            winner = 'team_b'
        else:
            winner = 'draw'

        # 各種指標計算
        metrics = {
            'winner': winner,
            'goal_difference': team_a_goals - team_b_goals,
            'episode_length': len(episode_data['steps']),
            'ball_possession_a': self._calculate_possession(episode_data, 'team_a'),
            'ball_possession_b': self._calculate_possession(episode_data, 'team_b'),
        }

        return metrics

    def _calculate_possession(self, episode_data, team):
        """ボール保持率計算"""
        possession_steps = 0
        total_steps = len(episode_data['steps'])

        for step in episode_data['steps']:
            if step['ball_holder'] in self._get_team_agents(team):
                possession_steps += 1

        return possession_steps / total_steps if total_steps > 0 else 0
```

## 2. 学習効率指標

python

```
class LearningEfficiencyMetrics:
    def calculate_sample_efficiency(self, performance_history, target_performance=0.8):
        """サンプル効率性の計算"""
        for i, performance in enumerate(performance_history):
            if performance >= target_performance:
                return i # 目標性能到達までのサンプル数
        return len(performance_history) # 到達しなかった場合

    def calculate_learning_stability(self, performance_history, window_size=100):
        """学習安定性の計算"""
        if len(performance_history) < window_size:
            return 0

        # 移動平均の分散を計算
        moving_averages = []
        for i in range(len(performance_history) - window_size + 1):
            avg = np.mean(performance_history[i:i + window_size])
            moving_averages.append(avg)

        return 1 / (1 + np.var(moving_averages)) # 分散が小さいほど安定
```

## 質的分析手法

### 1. 創発行動分析

python

```
class EmergentBehaviorAnalyzer:
    def __init__(self):
        self.behavior_patterns = {}

    def identify_formation(self, agent_positions, time_window=50):
        """フォーメーション識別"""
        formations = []

        for t in range(0, len(agent_positions), time_window):
            window_positions = agent_positions[t:t + time_window]

            # チーム別の平均位置計算
            team_positions = self._group_by_team(window_positions)

            # フォーメーション分類
            formation = self._classify_formation(team_positions)
            formations.append(formation)

        return formations

    def analyze_cooperation_patterns(self, episode_data):
        """協調パターン分析"""
        cooperation_events = []

        for step in episode_data['steps']:
            # パス判定
            if self._is_pass(step):
                cooperation_events.append({
                    'type': 'pass',
                    'from_agent': step['ball_holder'],
                    'to_agent': step['ball_receiver'],
                    'success': step['pass_success'],
                    'timestamp': step['timestamp']
                })

            # 連携プレイ判定
            if self._is_coordinated_play(step):
                cooperation_events.append({
                    'type': 'coordination',
                    'agents': step['involved_agents'],
                    'effectiveness': step['coordination_score'],
                    'timestamp': step['timestamp']
                })

        return self._extract_cooperation_patterns(cooperation_events)
```

## 2. 戦略進化分析

python

```
class StrategyEvolutionAnalyzer:
    def track_strategy_evolution(self, training_episodes):
        """戦略進化の追跡"""
        strategy_timeline = []

        # 一定間隔で戦略を抽出
        for episode_batch in self._batch_episodes(training_episodes, batch_size=1000):
            strategy = self._extract_strategy_features(episode_batch)
            strategy_timeline.append(strategy)

        return strategy_timeline

    def _extract_strategy_features(self, episodes):
        """戦略特徴の抽出"""
        features = {
            'aggression_level': self._calculate_aggression(episodes),
            'formation_preference': self._analyze_formations(episodes),
            'cooperation_frequency': self._measure_cooperation(episodes),
            'risk_taking': self._measure_risk_taking(episodes),
        }
        return features

    def visualize_strategy_evolution(self, strategy_timeline, save_path):
        """戦略進化の可視化"""
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # 各戦略特徴の時系列プロット
        features = ['aggression_level', 'formation_preference',
                    'cooperation_frequency', 'risk_taking']

        for i, feature in enumerate(features):
            ax = axes[i // 2, i % 2]
            values = [strategy[feature] for strategy in strategy_timeline]
            ax.plot(values)
            ax.set_title(f'{feature.replace("_", " ").title()} Evolution')
            ax.set_xlabel('Training Phase')
            ax.set_ylabel('Feature Value')

        plt.tight_layout()
        plt.savefig(save_path)
        plt.close()
```



# 最終成果物チェックリスト

## コード実装

- ☐ PettingZoo準拠のサッカー環境
- ☐ ランダムエージェント
- ☐ DQNエージェント
- ☐ PPOエージェント
- ☐ MADDPGエージェント
- ☐ QMIXエージェント（オプション）
- ☐ 独立学習フレームワーク
- ☐ セルフプレイフレームワーク
- ☐ 協調学習フレームワーク

## 実験・分析

- ☐ ベースライン実験（ランダムエージェント）
- ☐ 独立学習実験（DQN, PPO）
- ☐ セルフプレイ実験
- ☐ 協調学習実験（MADDPG）
- ☐ アブレーション研究（報酬設計、観測空間）
- ☐ 学習曲線の比較分析
- ☐ 創発行動の質的分析
- ☐ チームワーク指標の定量分析

## 可視化・文書

- ☐ 学習プロセスの動画
- ☐ 戦略進化の可視化
- ☐ 性能比較グラフ
- ☐ 詳細な実装ドキュメント
- ☐ 実験結果レポート
- ☐ コードのドキュメント化

この実装ドキュメントに従って、Claude Codeで段階的に実装を進めてください。各フェーズで必要に応じて詳細な仕様や実装指針を追加できます。