🏗️ CƠ CHẾ AGENT & ENVIRONMENT TRONG KIẾN TRÚC DUAL-AGENT RL IDS

Ngày: 27/10/2025
Kiến trúc: Dual-Agent Reinforcement Learning cho Intrusion Detection System

📋 MỤC LỤC

1. TỔNG QUAN KIẾN TRÚC

Diagram Tổng Quát:

```
┌─────────────────────────────────────────────────────────────┐
│                    DUAL-AGENT RL SYSTEM                       │
├─────────────────────────────────────────────────────────────┤
│                                                               │
│  ┌─────────────────┐         ┌─────────────────┐            │
│  │ Classifier Agent│ ◄─────► │Environment Agent│             │
│  │      (Q_c)      │ Compete │      (Q_e)      │             │
│  └────────┬────────┘         └────────┬────────┘             │
│           │                           │                       │
│           │ Select                    │ Attack                │
│           │ Model+Agg                 │ Type                  │
│           │                           │                       │
│          ▼                            ▼                 │
│  ┌──────────────────────────────────────────────┐        │
│  │     MTDAdversarialIDSEnv (Environment)       │        │
│  │  ┌─────────────────────────────────────────┐ │        │
│  │  │ Model Pool (MLP/CNN/TCN/Transformer)     │ │        │
│  │  │ Aggregators (mean/median/majority)        │  │       │
│  │  │ Autoencoder Denoiser (defense)            │  │       │
│  │  │ Drift Embedding (monitoring)              │  │       │
│  │  │ ADI Scheduler (drift injection)           │  │       │
│  │  └─────────────────────────────────────────┘ │        │
│  │                                                │       │
│  │  Execute actions → Return rewards & states    │       │
│  └──────────────────────────────────────────────┘        │
│          │                            │                  │
│          │ Reward_c (+Acc)           │ Reward_e (-Acc)  │
│          │ State_next                 │ State_next       │
│          │                            │                  │
│          ▼                            ▼                  │
│  ┌─────────────────┐         ┌─────────────────┐        │
│  │ Replay Buffer C │         │ Replay Buffer E │        │
│  │  (s,a,r,s',d)   │         │  (s,a,r,s',d)   │        │
│  └─────────────────┘         └─────────────────┘        │
│          │                            │                  │
│          │ Sample batches             │ Sample batches   │
│          │                            │                  │
│          ▼                            ▼                  │
│  ┌─────────────────┐         ┌─────────────────┐        │
│  │  Q-Learning     │         │  Q-Learning     │        │
│  │  Update Q_c     │         │  Update Q_e     │        │
│  └─────────────────┘         └─────────────────┘        │
│                                                           │
└───────────────────────────────────────────────────────────┘
```

---

2. PHÂN BIỆT CÁC KHÁI NIỆM

❗ QUAN TRỌNG: Có 3 thực thể khác nhau!

Thực thể	Vai trò	Mục tiêu	Code class
Classifier Agent	RL Agent #1	Maximize accuracy	`ClassifierAgent`

Thực thể	Vai trò	Mục tiêu	Code class
Environment Agent	RL Agent #2	Minimize accuracy	`EnvironmentAgent`
Environment (MTD)	Môi trường RL	Execute actions	`MTDAdversarialIDSEnv`

🔑 Phân biệt:

1. **Classifier Agent vs Environment Agent:**
    - Đây là 2 RL agents **đối nghịch** (adversarial)
    - Cùng observe state từ environment
    - Có action spaces **RIÊNG BIỆT**
    - Có rewards **ĐỐI NGHỊCH**
2. **Agent vs Environment:**
    - Agent **quyết định** action
    - Environment **thực thi** action và **trả về** (reward, next_state)
    - Relationship: **Actor-Observer** pattern

---

3. CLASSIFIER AGENT (Q_c)

3.1. Định nghĩa:

```
class ClassifierAgent(nn.Module):
    """
    Q_c network: selects (anchor_model, aggregator) pair.
    Action space = |models| × |aggregators|
    """
    def __init__(self, state_dim: int, n_models: int, n_aggregators: int,
        super().__init__()
        self.n_models = n_models
        self.n_aggregators = n_aggregators
        self.n_actions = n_models * n_aggregators  # 4 models × 3 aggs =

        # Q-network: State → Q-values for each action
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, self.n_actions),  # Output: Q-value for eac
        )
```

3.2. Action Space:

```
# Example: n_models=4, n_aggregators=3 → 12 actions total
# Action encoding: a_c = anchor_idx + agg_idx * n_models

Actions:
  0: (MLP, mean)       1: (CNN, mean)       2: (TCN, mean)       3: (Trar
  4: (MLP, median)     5: (CNN, median)     6: (TCN, median)     7: (Trar
  8: (MLP, majority)   9: (CNN, majority)  10: (TCN, majority)  11: (Trar
```

3.3. Decoding Action:

```
def decode_action(self, a_c: int) -> Tuple[int, int]:
    """Decode flat action into (anchor_idx, agg_idx)."""
    # Example: a_c = 7
    # anchor_idx = 7 % 4 = 3 (Transformer)
    # agg_idx = 7 // 4 = 1 (median)
    mid = a_c % self.n_models
    aid = a_c // self.n_models
    return mid, aid
```

3.4. Reward Function:

```
# GOAL: MAXIMIZE accuracy
reward_c = w1·ΔAcc - w2·Cost - w3·Switch - w4·Drift

Components:
  - ΔAcc: Change in accuracy (higher is better)
  - Cost: Computational cost of selected models
  - Switch: Penalty for changing models frequently
  - Drift: Penalty for high drift magnitude
```

3.5. State Input:

```
State dimension = 32:
  - Recent accuracy history: [10]
  - Recent reward history: [10]
  - Last actions (normalized): [3] → [model_idx/n_models, agg_idx/n_aggs,
  - Running average reward: [1]
  - Drift embedding vector: [11] → KS/PSI/MMD/CUSUM statistics
```

3.6. Learning Process:

```
# Q-Learning Update:
Q(s, a) ← Q(s, a) + α[r + γ·max_a' Q(s', a') - Q(s, a)]

Where:
  - s: current state
  - a: action taken (model+aggregator selection)
  - r: shaped reward (accuracy-based)
  - s': next state
  - γ: discount factor (0.95)
```

---

4. ENVIRONMENT AGENT (Q_e)

4.1. Định nghĩa:

```
class EnvironmentAgent(nn.Module):
    """
    Q_e network: selects adversarial attack (class × eps × attack_type).
    For simplicity, we use a flat action space.
    """
    def __init__(self, state_dim: int, n_attack_actions: int = 12, hidder
        super().__init__()
        self.n_actions = n_attack_actions

        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, self.n_actions),  # 12 attack types
        )
```

4.2. Action Space:

```
# 12 attack actions:
# Action encoding: a_e = attack_type * 4 + eps_level

Attack types:
  - 0: Random noise
  - 1: Sign-based (FGSM-like)
  - 2: Uniform noise

Epsilon levels (perturbation strength):
  - Level 0: eps = 0.01
  - Level 1: eps = 0.02
  - Level 2: eps = 0.03
  - Level 3: eps = 0.04

Total actions:
  a_e=0:  Random, eps=0.01    a_e=4:  Sign, eps=0.01      a_e=8:  Uniforn
  a_e=1:  Random, eps=0.02    a_e=5:  Sign, eps=0.02      a_e=9:  Uniforn
  a_e=2:  Random, eps=0.03    a_e=6:  Sign, eps=0.03      a_e=10: Uniforn
  a_e=3:  Random, eps=0.04    a_e=7:  Sign, eps=0.04      a_e=11: Uniforn
```

4.3. Attack Implementation:

```
# In MTDAdversarialIDSEnv.step():
if a_e is not None and a_e > 0:
    eps = 0.01 * (1 + (a_e % 4))  # Extract epsilon level
    attack_type = a_e // 4         # Extract attack type

    if attack_type == 0:
        # Random perturbation
        noise = torch.randn_like(x_t) * eps
    elif attack_type == 1:
        # Sign-based (FGSM-like without gradient)
        noise = torch.sign(torch.randn_like(x_t)) * eps
    else:
        # Uniform noise
        noise = (torch.rand_like(x_t) * 2 - 1) * eps

    x_attacked = x_t + noise
```

4.4. Reward Function:

```
# GOAL: MINIMIZE accuracy (maximize confusion)
reward_e = -accuracy_raw

Where:
  - accuracy_raw = 1.0 if prediction correct, 0.0 if wrong
  - Environment Agent wants to create adversarial examples that fool the
```

4.5. State Input:

```
# SAME state as Classifier Agent!
# Both agents observe the same environment state

State dimension = 32 (identical to Q_c)
```

4.6. Learning Process:

```
# Q-Learning Update (same algorithm, different reward):
Q_e(s, a) ← Q_e(s, a) + α[r_e + γ·max_a' Q_e(s', a') - Q_e(s, a)]

Where:
  - r_e = -accuracy_raw (negative reward for correct predictions)
  - Environment Agent learns to maximize misclassification
```

---

5. MTDAdversarialIDSEnv (MÔI TRƯỜNG)

5.1. Định nghĩa:

```
class MTDAdversarialIDSEnv:
    """
    Environment with:
    - Model pool (ensemble)
    - Autoencoder denoiser for adversarial robustness
    - Adversarial perturbation capability
    - Drift-aware state augmentation
    """
```

5.2. Components:

```
┌──────────────────────────────────────────────────────────┐
│             MTDAdversarialIDSEnv                         │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  1. MODEL POOL (Ensemble):                               │
│     ├─ MLPDropout       (512→256→128→2)                  │
│     ├─ DeepCNN          (Conv1D layers)                  │
│     ├─ DeepTCN          (Temporal Conv)                  │
│     └─ Transformer      (Self-attention)                 │
│                                                          │
│  2. AGGREGATORS:                                         │
│     ├─ mean_logits      (Average logits)                 │
│     ├─ median_logits    (Median logits)                  │
│     └─ majority_vote    (Vote counting)                  │
│                                                          │
│  3. AUTOENCODER DENOISER:                                │
│     ├─ Encoder: input → bottleneck (32-dim)              │
│     ├─ Decoder: bottleneck → reconstructed input         │
│     └─ Purpose: Remove adversarial perturbations         │
│                                                          │
│  4. DRIFT EMBEDDING:                                     │
│     ├─ KS/PSI statistics (per-feature drift)             │
│     ├─ MMD (Maximum Mean Discrepancy)                    │
│     └─ CUSUM (Cumulative Sum detector)                   │
│                                                          │
│  5. ADI SCHEDULER:                                       │
│     ├─ Covariate drift (feature shift)                   │
│     ├─ Prior drift (label flip)                          │
│     └─ Conditional drift (region-based flip)             │
│                                                          │
│  6. REWARD SHAPER:                                       │
│     └─ r' = w1·ΔAcc - w2·Cost - w3·Switch - w4·Drift     │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

5.3. Key Methods:

a) reset():

```
def reset(self, seed: Optional[int] = None) -> np.ndarray:
    """Reset environment to initial state."""
    self.i = 0  # Reset stream pointer
    self.hist = []  # Clear history
    self.adversarial_buffer = []  # Clear attack buffer

    # Initialize state
    return self._get_state()  # Return initial state (32-dim vector)
```

b) step():

```
def step(self, a_c: int, a_e: Optional[int] = None) -> Tuple[np.ndarray,
    """
    Execute one step in environment.

    Args:
        a_c: Classifier Agent action (model+aggregator selection)
        a_e: Environment Agent action (attack type, optional)

    Returns:
        next_state: Next state observation (32-dim)
        reward_c: Reward for Classifier Agent (shaped)
        reward_e: Reward for Environment Agent (-accuracy)
        done: Episode termination flag
        info: Additional information dict
    """
    # 1. Get current data sample
    x_current = self.X[self.i]
    y_current = self.y[self.i]

    # 2. Apply ADI drift injection (if active)
    x_current, y_current, adi_info = self.adi_scheduler.apply(self.i, x_c

    # 3. Apply adversarial attack (if Environment Agent acts)
    if a_e is not None and a_e > 0:
        x_attacked = self._apply_attack(x_current, a_e)
    else:
        x_attacked = x_current
    # 4. Apply denoiser defense
    if self.denoiser is not None:
        x_t = self.denoiser(x_attacked)
    else:
        x_t = x_attacked

    # 5. Decode Classifier Agent action
    anchor_idx = a_c % n_models
    agg_idx = a_c // n_models

    # 6. Select ensemble subset
    selected_models = self._select_ensemble_subset(anchor_idx, model_name

    # 7. Make prediction with ensemble
    pred, probs = self._ensemble_predict(x_t, selected_models, agg_methoc

    # 8. Compute rewards
    current_acc = float(pred == y_current)
    delta_acc = current_acc - self.prev_accuracy

    # Shaped reward for Classifier Agent
    reward_c_shaped = self.reward_shaper.total_reward(delta_acc, selectec

    # Raw accuracy for Environment Agent (negative)
    reward_e = -current_acc

    # 9. Update drift embedding
    self.drift_embedding.push(x_current, score=probs.max(), y=y_current)

    # 10. Move to next sample
    self.i += 1
    done = (self.i >= len(self.X))

    # 11. Get next state
    next_state = self._get_state() if not done else np.zeros(self.obs_dim

    return next_state, reward_c_shaped, reward_e, done, info
```

c) _get_state():

```
  def _get_state(self) -> np.ndarray:
    """Construct state vector (32-dim)."""
    state = np.concatenate([
        np.array(list(self.acc_history), dtype=np.float32),    # [10] rec
        np.array(list(self.reward_history), dtype=np.float32), # [10] rec
        self.last_actions,                                      # [3] last
        np.array([self.avg_reward], dtype=np.float32),         # [1] runni
        self.drift_embedding.vector() if self.drift_embedding else np.zer
    ])
    return state
```

5.4. Action Decoding (CRITICAL):

```
# MUST match ClassifierAgent.decode_action()!

# In Environment:
anchor_idx = a_c_clamped % n_models  # Same as ClassifierAgent
agg_idx = a_c_clamped // n_models    # Same as ClassifierAgent

# CONSISTENCY CHECK:
agent_mid, agent_aid = q_c.decode_action(a_c)
env_mid = a_c % n_models
env_aid = a_c // n_models

assert agent_mid == env_mid  # MUST be equal!
assert agent_aid == env_aid  # MUST be equal!
```

---

6. LUỒNG TƯƠNG TÁC

6.1. Single Step Flow:

```
┌─────────────────────────────────────────────────────────────┐
│                     ONE TRAINING STEP                       │
└─────────────────────────────────────────────────────────────┘

1. OBSERVE STATE:
   ┌─────────────────┐
   │ Environment     │
   │ state = [32]    │──┐
   └─────────────────┘  │
                        ├──► Classifier Agent observes state
```

```
                         │
                         └──▶ Environment Agent observes state

2. SELECT ACTIONS:

   ┌─────────────────┐
   │Classifier Agent │
   │  ε-greedy       │──▶ a_c = 7 (Transformer + median)
   └─────────────────┘

   ┌─────────────────┐
   │Environment Agent│
   │  ε-greedy       │──▶ a_e = 5 (Sign attack, eps=0.02)
   └─────────────────┘

3. EXECUTE IN ENVIRONMENT:

   ┌────────────────────────────────────────┐
   │ MTDAdversarialIDSEnv.step(a_c=7, a_e=5) │
   │                                        │
   │ 1. Get data: x, y                      │
   │ 2. Apply drift: x' = ADI(x)            │
   │ 3. Apply attack: x'' = Attack(x', a_e) │
   │ 4. Denoise: x''' = Denoiser(x'')       │
   │ 5. Predict: pred = Ensemble(x''', a_c) │
   │ 6. Compute rewards:                    │
   │    - reward_c = shaped_reward(...)     │
   │    - reward_e = -accuracy              │
   │ 7. Get next state: s'                  │
   └────────────────────────────────────────┘
                  │
                  ▼
   ┌────────────────────────────────────────┐
   │  RETURNS: (s', r_c, r_e, done, info)   │
   └────────────────────────────────────────┘

4. STORE TRANSITIONS:

   ┌─────────────────┐
   │Replay Buffer C  │──▶ (s, a_c, r_c, s', done)
   └─────────────────┘

   ┌─────────────────┐
   │Replay Buffer E  │──▶ (s, a_e, r_e, s', done)
   └──────────────────┘


5. LEARN FROM EXPERIENCE:
   ┌──────────────────────────────────────┐
   │ Sample batch from Replay Buffer C    │
   │ Update Q_c network (Q-learning)      │
   └──────────────────────────────────────┘


   ┌──────────────────────────────────────┐
   │ Sample batch from Replay Buffer E    │
   │ Update Q_e network (Q-learning)      │
   └──────────────────────────────────────┘


6. REPEAT for next sample...
```

6.2. Episode Flow:

```
Episode = Processing entire dataset sequentially


┌─────────────────────────────────────────────────────────┐
│ EPISODE START (i=0)                                     │
└─────────────────────────────────────────────────────────┘
   │
   │ 1. s = env.reset()
   │
   ▼
┌─────────────────────────────────────────────────────────┐
│ STEP LOOP (i=0 to N-1)                                  │
├─────────────────────────────────────────────────────────┤
│  while not done:                                        │
│    1. Select actions:                                   │
│       a_c = ε-greedy(Q_c(s))                            │
│       a_e = ε-greedy(Q_e(s))                            │
│                                                         │
│    2. Execute:                                          │
│       s', r_c, r_e, done, info = env.step(a_c, a_e)    │
│                                                         │
│    3. Store:                                            │
│       replay_c.push(s, a_c, r_c, s', done)             │
│       replay_e.push(s, a_e, r_e, s', done)             │
│                                                         │
│    4. Learn (every few steps):                          │
│       if len(replay_c) >= batch_size:                   │
│         update_network(Q_c, replay_c, ...)              │
│         update_network(Q_e, replay_e, ...)              │
│                                                          │
│    5. Update state:                                      │
│       s = s'                                             │
│       i += 1                                             │
└──────────────────────────────────────────────────────────┘
   │
   │ done = True (reached end of dataset)
   │
   ▼
┌──────────────────────────────────────────────────────────┐
│ EPISODE END                                              │
│ - Decay ε (exploration rate)                             │
│ - Save checkpoint                                         │
│ - Log metrics                                             │
└──────────────────────────────────────────────────────────┘
```

---

7. MARKOV DECISION PROCESS (MDP)

7.1. MDP Framework cho Dual-Agent System:

Hệ thống dual-agent RL của chúng ta có thể được mô hình hóa như một **Multi-Agent Markov Decision Process (MAMDP)** với các thành phần:

```
MDP = (S, A_c, A_e, P, R_c, R_e, γ)

Where:
  S      : State space (32-dimensional continuous space)
  A_c    : Classifier Agent action space (12 discrete actions)
  A_e    : Environment Agent action space (12 discrete actions)
  P      : State transition probability P(s'|s, a_c, a_e)
  R_c    : Reward function for Classifier Agent
  R_e    : Reward function for Environment Agent
  γ      : Discount factor (0.95)
```

7.2. Sơ Đồ MDP Tổng Quan:

```
┌──────────────────────────────────────────────────────────────────────────
┐
│                    DUAL-AGENT MARKOV DECISION PROCESS
│
└──────────────────────────────────────────────────────────────────────────
┘

TIME STEP: t                          TIME STEP: t+1
═══════════════════════════════════════════════════════════════════════════
═

┌──────────────┐                                       ┌──────────────┐
│   State s_t  │                                       │  State s_t+1 │
│   (32-dim)   │◄──────────────────────────────────────│   (32-dim)   │
│              │  Transition P(s'|s,a_c,a_e)           │              │
│ - Acc hist   │                                       │ - Acc hist   │
│ - Reward hist│                                       │ - Reward hist│
│ - Last action│                                       │ - Last action│
│ - Avg reward │                                       │ - Avg reward │
│ - Drift embed│                                       │ - Drift embed│
└──────────────┘                                       └──────▲───────┘
       │                                                      │
       │ Observe                                              │
       │                                                      │
       ├────────────────┬──────────────────┐                 │
       │                │                  │                 │
       ▼                ▼                  ▼                 │
                                                           │
┌─────────────┐  ┌─────────────┐  ┌─────────────┐          │
│ Classifier  │  │ Environment │  │ Environment │          │
│   Agent     │  │   Agent     │  │(MTD System) │          │
│             │  │             │  │             │          │
│ Policy π_c  │  │ Policy π_e  │  │  Execute    │          │
│  (Q-net)    │  │  (Q-net)    │  │  Actions    │          │
└──────┬──────┘  └──────┬──────┘  └──────┬──────┘          │
       │                │                 │                 │
       │ Select         │ Select          │                 │
       │ Action         │ Action          │                 │
       │                │                 │                 │
       ▼                ▼                 │                 │
  a_c ∈ A_c       a_e ∈ A_e              │                 │
  (model+agg)     (attack type)          │                 │
```

```
        │                │                 │              │
        │                │                 │              │
        └───────┬────────┘                 │              │
                │                          │              │
                │ Joint Action             │              │
                │ (a_c, a_e)               │              │
                └──────────────────────────▶              │
                                           │              │
                              ┌────────────┴──────────┐   │
                              │   EXECUTION PHASE     │   │
                              │                       │   │
                              │ 1. Get data (x,y)     │   │
                              │ 2. ADI drift          │   │
                              │ 3. Attack (a_e)       │   │
                              │ 4. Denoise            │   │
                              │ 5. Predict (a_c)      │   │
                              │ 6. Compute rewards    │   │
                              │ 7. Update state       │   │
                              └────────────┬──────────┘   │
                                           │              │
                                           ▼              │
                              ┌──────────────────────┐    │
                              │   REWARD SIGNALS     │    │
                              ├──────────────────────┤    │
                              │ r_c = shaped_reward  │    │
                              │     = w1·ΔAcc -      │    │
                              │       w2·Cost -      │    │
                              │       w3·Switch -    │    │
                              │       w4·Drift       │    │
                              │                      │    │
                              │ r_e = -accuracy_raw  │    │
                              │     = -1.0 (correct) │    │
                              │     =  0.0 (wrong)   │    │
                              └──────────┬───────────┘    │
                                         │                │
                                         └────────────────┘
```

═══
=

LEARNING UPDATES (Off-policy Q-learning):

```
Classifier Agent:
  Q_c(s_t, a_c) ← Q_c(s_t, a_c) + α[r_c + γ·max_a' Q_c(s_t+1, a') -
Q_c(s_t, a_c)]

Environment Agent:
  Q_e(s_t, a_e) ← Q_e(s_t, a_e) + α[r_e + γ·max_a' Q_e(s_t+1, a') -
Q_e(s_t, a_e)]
```

7.3. State Space (S):

```
┌─────────────────────────────────────────────────────────────────┐
│                    STATE SPACE S (32-dim)                       │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  Dimensions [0-9]: Recent Accuracy History                    │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │ [acc_t-9, acc_t-8, ..., acc_t-1]                          │  │
│  │ Range: [0.0, 1.0] per dimension                           │  │
│  │ Purpose: Track performance trend                          │  │
│  └──────────────────────────────────────────────────────────┘  │
│                                                                 │
│  Dimensions [10-19]: Recent Reward History                    │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │ [r_t-9, r_t-8, ..., r_t-1]                                │  │
│  │ Range: [-1.0, +1.0] (typically)                           │  │
│  │ Purpose: Track reward trajectory                          │  │
│  └──────────────────────────────────────────────────────────┘  │
│                                                                 │
│  Dimensions [20-22]: Last Actions (Normalized)                │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │ [model_idx/n_models, agg_idx/n_aggs, attack_idx/12]      │  │
│  │ Range: [0.0, 1.0] per dimension                           │  │
│  │ Purpose: Action history for temporal patterns            │  │
│  └──────────────────────────────────────────────────────────┘  │
│                                                                 │
│  Dimension [23]: Running Average Reward                       │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │ avg_reward = EMA(rewards)                                 │  │
│  │ Range: [-1.0, +1.0]                                       │  │
│  │ Purpose: Long-term performance indicator                  │  │
│   └───────────────────────────────────────────────────────────┘   │
│                                                                    │
│  Dimensions [24-31]: Drift Embedding (8-dim actual)                │
│   ┌───────────────────────────────────────────────────────────┐   │
│   │ - KS/PSI statistics (per-feature drift)                    │   │
│   │ - MMD (Maximum Mean Discrepancy)                           │   │
│   │ - CUSUM (Cumulative Sum control chart)                     │   │
│   │ Range: Normalized [0.0, 1.0]                               │   │
│   │ Purpose: Detect and quantify concept drift                │   │
│   └───────────────────────────────────────────────────────────┘   │
│                                                                    │
└────────────────────────────────────────────────────────────────────┘

State Evolution:
  s_t → s_t+1 through environment execution

  Update rules:
    - acc_history: shift left, append new accuracy
    - reward_history: shift left, append new reward
    - last_actions: update with (a_c, a_e) normalized
    - avg_reward: exponential moving average
    - drift_embed: update with latest drift statistics
```

7.4. Action Spaces (A_c, A_e):

```
┌─────────────────────────────────────────────────────────────────┐
│              CLASSIFIER AGENT ACTION SPACE (A_c)                 │
├─────────────────────────────────────────────────────────────────┤
│                                                                  │
│  |A_c| = 12 discrete actions                                    │
│                                                                  │
│  Action encoding: a_c = anchor_idx + agg_idx × n_models         │
│                                                                  │
│   ┌──────────────────────────────────────────────────────────┐  │
│   │ Anchor Model        │ Aggregator │ Action ID │ Effect      │  │
│   ├─────────────────────┼────────────┼───────────┼─────────────┤  │
│   │ MLP (0)             │ Mean (0)   │    0      │ Fast        │  │
│   │ CNN (1)             │ Mean (0)   │    1      │ Balanced    │  │
│   │ TCN (2)             │ Mean (0)   │    2      │ Temporal    │  │
│   │ Transformer (3)     │ Mean (0)   │    3      │ Accurate    │  │
│   │ MLP (0)             │ Median (1) │    4      │ Robust      │  │
│  │ CNN (1)           │ Median (1) │    5      │ Robust     │ │
│  │ TCN (2)           │ Median (1) │    6      │ Robust     │ │
│  │ Transformer (3)   │ Median (1) │    7      │ Robust     │ │
│  │ MLP (0)           │ Majority(2)│    8      │ Vote-based │ │
│  │ CNN (1)           │ Majority(2)│    9      │ Vote-based │ │
│  │ TCN (2)           │ Majority(2)│   10      │ Vote-based │ │
│  │ Transformer (3)   │ Majority(2)│   11      │ Vote-based │ │
│  └──────────────────────────────────────────────────────────┘ │
│                                                                │
│  Policy: π_c(a_c|s) = ε-greedy over Q_c(s, ·)                 │
│                                                                │
└────────────────────────────────────────────────────────────────┘


┌────────────────────────────────────────────────────────────────┐
│            ENVIRONMENT AGENT ACTION SPACE (A_e)                │
├────────────────────────────────────────────────────────────────┤
│                                                                │
│  |A_e| = 12 discrete actions                                  │
│                                                                │
│  Action encoding: a_e = attack_type × 4 + eps_level           │
│                                                                │
│  ┌──────────────────────────────────────────────────────────┐ │
│  │ Attack Type       │ Epsilon │ Action ID │ Strength        │ │
│  ├───────────────────┼─────────┼───────────┼─────────────────┤ │
│  │ Random (0)        │ 0.01    │    0      │ Very Weak       │ │
│  │ Random (0)        │ 0.02    │    1      │ Weak            │ │
│  │ Random (0)        │ 0.03    │    2      │ Medium          │ │
│  │ Random (0)        │ 0.04    │    3      │ Strong          │ │
│  │ Sign-based (1)    │ 0.01    │    4      │ Weak FGSM       │ │
│  │ Sign-based (1)    │ 0.02    │    5      │ Medium FGSM     │ │
│  │ Sign-based (1)    │ 0.03    │    6      │ Strong FGSM     │ │
│  │ Sign-based (1)    │ 0.04    │    7      │ Very Strong     │ │
│  │ Uniform (2)       │ 0.01    │    8      │ Weak Uniform    │ │
│  │ Uniform (2)       │ 0.02    │    9      │ Medium Uniform  │ │
│  │ Uniform (2)       │ 0.03    │   10      │ Strong Uniform  │ │
│  │ Uniform (2)       │ 0.04    │   11      │ Max Uniform     │ │
│  └──────────────────────────────────────────────────────────┘ │
│                                                                │
│  Policy: π_e(a_e|s) = ε-greedy over Q_e(s, ·)                 │
│                                                                │
└────────────────────────────────────────────────────────────────┘
```

7.5. State Transition Function P(s'|s, a_c, a_e):

```
┌─────────────────────────────────────────────────────────────────────┐
│                STATE TRANSITION DYNAMICS                            │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  P(s_{t+1} | s_t, a_c, a_e) = Pr[next state | current context]     │
│                                                                     │
│  Transition Process:                                                │
│  ┌─────────────────────────────────────────────────────────────┐   │
│  │                                                               │ │
│  │  1. DETERMINISTIC COMPONENTS:                               │ │
│  │     • Data stream: x_{t+1}, y_{t+1} ← Dataset[i+1]          │ │
│  │     • History update: shift and append                      │ │
│  │     • Action recording: last_actions ← (a_c, a_e)           │ │
│  │                                                               │ │
│  │  2. STOCHASTIC COMPONENTS:                                  │ │
│  │     • ADI drift: x' ~ DriftInjection(x, schedule)           │ │
│  │     • Attack noise: x'' ~ Attack(x', a_e, ε)                │ │
│  │     • Model prediction: pred ~ Ensemble(x'', a_c)           │ │
│  │     • Accuracy: acc =  [pred == y]  (stochastic via model)  │ │
│  │                                                               │ │
│  │  3. STATE CONSTRUCTION:                                     │ │
│  │     s_{t+1} = [                                               │ │
│  │       acc_history[1:] + [acc_new],     # Shift & append     │ │
│  │       reward_history[1:] + [r_new],    # Shift & append     │ │
│  │       normalize(a_c, a_e),             # Action encoding    │ │
│  │       EMA(rewards),                     # Running average    │ │
│  │       DriftEmbedding.vector()          # Drift stats        │ │
│  │     ]                                                         │ │
│  │                                                               │ │
│  └─────────────────────────────────────────────────────────────┘   │
│                                                                     │
│  Markov Property:                                                   │
│    P(s_{t+1}|s_t,a_c,a_e) = P(s_{t+1}|s_t,a_c,a_e,s_{t-1},...)│
│                                                                     │
│  → State s_t contains sufficient statistics (history buffer)      │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

7.6. Reward Functions (R_c, R_e):

REWARD FUNCTIONS

$R\_c: S \times A\_c \times A\_e \to \mathbb{R}$ (Classifier Agent reward)

$r\_c(s,a\_c,a\_e) = w_1 \cdot \Delta Acc - w_2 \cdot Cost - w_3 \cdot Switch - w_4 \cdot Drift$

Component	Formula	Weight	Range
Accuracy Δ	acc_t - acc_{t-1}	$w_1=1.0$	[-1,+1]
Cost Penalty	Σ model_complexity / max	$w_2=0.2$	[0, 1]
Switch Cost	[action changed]	$w_3=0.1$	{0, 1}
Drift Impact	||drift_vector||	$w_4=0.3$	[0, 1]

Design Goal: Maximize long-term cumulative return

$G\_t\hat{}c = \Sigma\_\{k=0\}\hat{}\infty\ \gamma\hat{}k \cdot r\_c(s\_\{t+k\}, a\_c, a\_e)$

Optimal Policy:

$\pi*\_c = argmax\_\{\pi\_c\}\ [G\_t\hat{}c \mid \pi\_c, \pi\_e]$

$R\_e: S \times A\_c \times A\_e \to \mathbb{R}$ (Environment Agent reward)

r_e(s,a_c,a_e) = -accuracy_raw

Outcome	Accuracy	Reward r_e	Interpretation
Correct predict	1.0	-1.0	BAD (defense)
Wrong predict	0.0	0.0	GOOD (attack)

Design Goal: Minimize classifier accuracy (adversarial)

$G\_t\hat{}e = \Sigma\_\{k=0\}\hat{}\infty\ \gamma\hat{}k \cdot r\_e(s\_\{t+k\}, a\_c, a\_e)$

$= -\Sigma\_\{k=0\}\hat{}\infty\ \gamma\hat{}k \cdot accuracy\_\{t+k\}$

```
|                                                               |
|  Optimal Policy:                                              |
|    π*_e = argmax_{π_e}  [G_t^e | π_c, π_e]                    |
|          = argmin_{π_e}  [Σ accuracy | π_c, π_e]               |
|                                                               |
└───────────────────────────────────────────────────────────────┘
```

```
ZERO-SUM GAME PROPERTY:
  While r_c + r_e ≠ 0 (not strictly zero-sum due to shaped reward),
  the agents have OPPOSING objectives:
    - Classifier: maximize accuracy
    - Attacker: minimize accuracy

  This creates adversarial pressure for robustness.
```

7.7. Bellman Equations:

```
┌───────────────────────────────────────────────────────────────┐
│                   BELLMAN OPTIMALITY EQUATIONS                │
├───────────────────────────────────────────────────────────────┤
│                                                               │
│  For Classifier Agent (Q_c):                                  │
│  ──────────────────────────                                   │
│                                                               │
│  Q*_c(s, a_c) =  [r_c(s,a_c,a_e) + γ·max_{a'_c} Q*_c(s',a'_c)] │
│                                                               │
│  Iterative Update (Q-learning):                               │
│  ┌───────────────────────────────────────────────────────────┐ │
│  │                                                           │ │
│  │  Q_c(s_t, a_c) ← Q_c(s_t, a_c) +                          │ │
│  │                  α[r_c + γ·max_{a'} Q_c(s_{t+1}, a')      │ │
│  │                     - Q_c(s_t, a_c)]                      │ │
│  │                  └────────────┬────────────┘              │ │
│  │                         TD Error δ_c                      │ │
│  │                                                           │ │
│  │  Where:                                                   │ │
│  │    α = learning rate (e.g., 0.001 via Adam optimizer)     │ │
│  │    γ = discount factor (0.95)                             │ │
│  │    δ_c = temporal difference error                        │ │
│  │                                                           │ │
│  └───────────────────────────────────────────────────────────┘ │
│                                                                  │
│  Convergence Guarantee:                                          │
│    If all (s,a) pairs visited infinitely often and              │
│    learning rate satisfies Robbins-Monro conditions,            │
│    then Q_c → Q*_c as t → ∞                                     │
│                                                                  │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  For Environment Agent (Q_e):                                    │
│  ────────────────────────────                                    │
│                                                                  │
│  Q*_e(s, a_e) =  [r_e(s,a_c,a_e) + γ·max_{a'_e} Q*_e(s',a'_e)] │
│                                                                  │
│  Iteration Update (Q-learning):                                  │
│   ┌────────────────────────────────────────────────────────────┐ │
│   │                                                             │ │
│   │  Q_e(s_t, a_e) ← Q_e(s_t, a_e) +                            │ │
│   │                  α[r_e + γ·max_{a'} Q_e(s_{t+1}, a')       │ │
│   │                     - Q_e(s_t, a_e)]                        │ │
│   │                  └────────────┬────────────┘                │ │
│   │                           TD Error δ_e                      │ │
│   │                                                             │ │
│   └────────────────────────────────────────────────────────────┘ │
│                                                                  │
│  Independent Learning:                                           │
│    - Both agents learn simultaneously                            │
│    - Non-stationary environment (each agent's policy changes)    │
│    - No Nash equilibrium guarantee in general                    │
│    - Empirically: adversarial training improves robustness       │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

7.8. Trajectory và Episode:

```
┌──────────────────────────────────────────────────────────────────┐
│                     TRAJECTORY STRUCTURE                         │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  Episode = One complete pass through dataset                     │
│  Length = N samples in streaming data                            │
│                                                                  │
│  Trajectory τ:                                                    │
│  ┌──────────────────────────────────────────────────────────────┐ │
│  │                                                              │ │
│  │  τ = (s₀, a_c₀, a_e₀, r_c₀, r_e₀,                          │ │
│  │       s₁, a_c₁, a_e₁, r_c₁, r_e₁,                          │ │
│  │       ...,                                                   │ │
│  │       s_N, a_c_N, a_e_N, r_c_N, r_e_N)                       │ │
│  │                                                              │ │
│  │  Cumulative Returns:                                         │ │
│  │    G₀^c = Σ_{t=0}^{N-1} γ^t · r_c_t  (Classifier)          │ │
│  │    G₀^e = Σ_{t=0}^{N-1} γ^t · r_e_t  (Environment)         │ │
│  │                                                              │ │
│  └──────────────────────────────────────────────────────────────┘ │
│                                                                   │
│  Example Episode Flow:                                            │
│  ┌──────────────────────────────────────────────────────────────┐ │
│  │ t=0:  s₀ → (a_c=3, a_e=5) → r_c=+0.12, r_e=-1.0 → s₁       │ │
│  │ t=1:  s₁ → (a_c=7, a_e=8) → r_c=+0.05, r_e=-1.0 → s₂       │ │
│  │ t=2:  s₂ → (a_c=7, a_e=11)→ r_c=-0.20, r_e= 0.0 → s₃       │ │
│  │ ...                                                          │ │
│  │ t=N:  s_N → TERMINAL STATE                                   │ │
│  └──────────────────────────────────────────────────────────────┘ │
│                                                                   │
│  Properties:                                                      │
│    • Finite horizon: T = N (dataset size)                        │
│    • Deterministic termination: done = (i >= N)                  │
│    • No early stopping (process all samples)                     │
│    • Episodic task (clear start and end)                         │
│                                                                   │
└───────────────────────────────────────────────────────────────────┘
```

7.9. Nash Equilibrium và Convergence:

```
┌───────────────────────────────────────────────────────────────────┐
│              GAME-THEORETIC ANALYSIS                              │
├───────────────────────────────────────────────────────────────────┤
│                                                                   │
│  Game Type: Two-Player General-Sum Markov Game                    │
│                                                                   │
│  Players:                                                         │
│    • Player 1: Classifier Agent (maximizes r_c)                  │
│    • Player 2: Environment Agent (maximizes r_e = -accuracy)    │
│                                                                  │
│  Strategy Space:                                                 │
│    • Π_c = {π_c : S → Δ(A_c)} (policies for Classifier)         │
│    • Π_e = {π_e : S → Δ(A_e)} (policies for Environment)        │
│                                                                  │
│  Nash Equilibrium (π*_c, π*_e):                                  │
│  ┌────────────────────────────────────────────────────────────┐ │
│  │                                                              │ │
│  │  V_c(s; π*_c, π*_e) ≥ V_c(s; π_c, π*_e)  ∀π_c, ∀s         │ │
│  │  V_e(s; π*_c, π*_e) ≥ V_e(s; π*_c, π_e)  ∀π_e, ∀s         │ │
│  │                                                              │ │
│  │  Interpretation:                                             │ │
│  │    Neither agent can improve by unilaterally changing        │ │
│  │    its policy given the other agent's fixed policy.         │ │
│  │                                                              │ │
│  └────────────────────────────────────────────────────────────┘ │
│                                                                  │
│  Convergence Challenges:                                         │
│  ┌────────────────────────────────────────────────────────────┐ │
│  │    Non-stationary Environment:                                │ │
│  │   - Each agent's policy changes during training              │ │
│  │   - Violates MDP stationarity assumption                     │ │
│  │   - Can lead to oscillations or cycles                       │ │
│  │                                                              │ │
│  │    No Guaranteed Convergence:                                 │ │
│  │   - Independent Q-learning may not converge to Nash          │ │
│  │   - Can converge to suboptimal equilibria                    │ │
│  │                                                              │ │
│  │ ✓ Empirical Success:                                          │ │
│  │   - Adversarial training improves robustness in practice     │ │
│  │   - Experience replay stabilizes learning                    │ │
│  │   - Slow policy updates reduce non-stationarity              │ │
│  │                                                              │ │
│  └────────────────────────────────────────────────────────────┘ │
│                                                                  │
│  Practical Goal:                                                 │
│    Find approximate equilibrium (π^_c, π^_e) where:               │
│      • Classifier achieves high accuracy under attack           │
│      • Attacker forces classifier to be robust                  │
│      • Co-evolution leads to better generalization              │
```

8. TRAINING LOOP

7.1. Main Training Function:

```
def train_dual_agent_rl(
    env: MTDAdversarialIDSEnv,
    q_c: ClassifierAgent,
    q_e: EnvironmentAgent,
    replay_c: ReplayBuffer,
    replay_e: ReplayBuffer,
    optimizer_c: torch.optim.Optimizer,
    optimizer_e: torch.optim.Optimizer,
    prequential: Optional[PrequentialMetrics] = None,
    start_step: int = 0,
    epochs: int = 3,
    batch_size: int = 64,
    gamma: float = 0.95,
    epsilon: float = 0.2,
) -> Tuple[int, float]:
    """Train dual-agent RL system."""

    device = env.device
    global_step = start_step

    for epoch in range(epochs):
        # Reset environment for new episode
        state = env.reset(seed=epoch)
        done = False

        while not done:
            # 1. SELECT ACTIONS (ε-greedy)
            if random.random() < epsilon:
                a_c = random.randint(0, q_c.n_actions - 1)
                a_e = random.randint(0, q_e.n_actions - 1)
            else:
                with torch.no_grad():
                    s_t = torch.from_numpy(state).unsqueeze(0).to(device)
                    a_c = q_c(s_t).argmax().item()
                    a_e = q_e(s_t).argmax().item()
            # 2. EXECUTE in environment
            next_state, reward_c, reward_e, done, info = env.step(a_c, a_

            # 3. STORE transitions
            replay_c.push(state, a_c, reward_c, next_state, done)
            replay_e.push(state, a_e, reward_e, next_state, done)

            # 4. LEARN from experience
            if len(replay_c) >= batch_size and global_step % 4 == 0:
                _update_network(q_c, replay_c, optimizer_c, batch_size, g
                _update_network(q_e, replay_e, optimizer_e, batch_size, g

            # 5. UPDATE state
            state = next_state
            global_step += 1

            # 6. CHECKPOINT & LOGGING
            if global_step % SAVE_EVERY_STEPS == 0:
                save_checkpoint(q_c, q_e, optimizer_c, optimizer_e, globa

        # Decay exploration
        epsilon = max(0.01, epsilon * 0.95)

    return global_step, epsilon
```

7.2. Q-Learning Update:

```
def _update_network(
    q_net: nn.Module,
    replay: ReplayBuffer,
    optimizer: torch.optim.Optimizer,
    batch_size: int,
    gamma: float,
    device: torch.device,
):
    """Update Q-network using experience replay."""

    # 1. SAMPLE batch from replay buffer
    s, a, r, ns, done = replay.sample(batch_size)

    # 2. Convert to tensors
    s_t = torch.from_numpy(s).to(device)
    a_t = torch.from_numpy(a).to(device)
    r_t = torch.from_numpy(r).to(device)
    ns_t = torch.from_numpy(ns).to(device)
    done_t = torch.from_numpy(done).to(device)

    # 3. COMPUTE current Q-values
    q_pred = q_net(s_t).gather(1, a_t.view(-1, 1)).squeeze(1)

    # 4. COMPUTE target Q-values (Bellman equation)
    with torch.no_grad():
        q_next = q_net(ns_t).max(dim=1)[0]  # Max Q-value for next state
        target = r_t + gamma * q_next * (1.0 - done_t)  # TD target

    # 5. COMPUTE loss
    loss = F.smooth_l1_loss(q_pred, target)

    # 6. BACKPROPAGATION
    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(q_net.parameters(), max_norm=1.0)
    optimizer.step()
```

---

9. SO SÁNH AGENT VS ENVIRONMENT

8.1. Bảng So Sánh Chi Tiết:

Khía cạnh	Classifier Agent (Q_c)	Environment Agent (Q_e)	MTDAdversarialIDSEnv
Vai trò	RL Agent (Defense)	RL Agent (Attack)	Môi trường RL
Mục tiêu	Maximize accuracy	Minimize accuracy	Execute & return results
Action Space	12 actions (4×3)	12 actions (attack types)	N/A
Action Type	Model+Aggregator selection	Attack type selection	Execute actions
Reward	+Shaped (ΔAcc-Cost-Switch-Drift)	-Accuracy (raw)	Compute & return rewards
State	Observe 32-dim vector	Observe 32-dim vector	Provide state vector

Khía cạnh	Classifier Agent (Q_c)	Environment Agent (Q_e)	MTDAdversarialIDSEnv
Learning	Q-learning (maximize reward)	Q-learning (maximize -acc)	No learning
Network	MLP (state→Q-values)	MLP (state→Q-values)	Ensemble models
Replay Buffer	Separate (capacity=100k)	Separate (capacity=100k)	N/A
Exploration	ε-greedy	ε-greedy	N/A
Update Frequency	Every 4 steps	Every 4 steps	Every step

8.2. Relationship Diagram:

```
┌──────────────────────────────────────────────────────────┐
│                   RELATIONSHIPS                          │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  1. ADVERSARIAL (Agent vs Agent):                       │
│                                                          │
│     Classifier Agent (Q_c)                               │
│            ↕ ↕ ↕ ↕ ↕                                     │
│      COMPETE FOR REWARDS                                 │
│            ↕ ↕ ↕ ↕ ↕                                     │
│     Environment Agent (Q_e)                              │
│                                                          │
│     - Same state observation                             │
│     - Different action spaces                            │
│     - OPPOSITE rewards (r_c = +acc, r_e = -acc)          │
│     - Independent learning (separate replay buffers)     │
│                                                          │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  2. COOPERATIVE (Agent vs Environment):                 │
│                                                          │
│     Classifier Agent ──▶ MTDAdversarialIDSEnv            │
│                           │                              │
│                           ├─▶ Execute a_c action         │
│                           │   (model+agg selection)      │
│                           │                              │
│                           └─▶ Return (s', r_c, done)     │
│                                                           │
│     Environment Agent ──▶ MTDAdversarialIDSEnv            │
│                           │                               │
│                           ├──▶ Execute a_e action         │
│                           │    (attack type)              │
│                           │                               │
│                           └──▶ Return (s', r_e, done)     │
│                                                           │
│     - Environment MUST correctly decode actions           │
│     - Consistent action space definition                  │
│     - Environment provides state observations             │
│                                                           │
└───────────────────────────────────────────────────────────┘
```

8.3. Information Flow:

```
TIME: t

┌─────────────────┐
│   State s_t     │ (32-dim vector)
└────────┬────────┘
         │
         ├──────────────────┬──────────────────┐
         │                  │                  │
         ▼                  ▼                  ▼
┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐
│ Classifier Agent│  │Environment Agent│  │  Environment    │
│    Observe      │  │    Observe      │  │   (Context)     │
└────────┬────────┘  └────────┬────────┘  └─────────────────┘
         │                    │
         │ Decide             │ Decide
         │                    │
         ▼                    ▼
    a_c = 7              a_e = 5
  (Transformer+        (Sign attack
   median)             eps=0.02)
         │                    │
         └──────────┬─────────┘
                    │
                    ▼
         ┌──────────────────────┐
```

```
          │  MTDAdversarialIDSEnv│
          │   .step(a_c, a_e)   │
          └──────────┬───────────┘
                     │
                     │ Execute both actions
                     │ simultaneously
                     │
                     ▼
          ┌──────────────────────┐
          │  1. Get data: x, y   │
          │  2. Drift: ADI       │
          │  3. Attack: a_e      │
          │  4. Denoise          │
          │  5. Predict: a_c     │
          │  6. Rewards          │
          └──────────┬───────────┘
                     │
          ┌──────────┴───────────┐
          │                      │
          ▼                      ▼
    s'_t+1, r_c, done      s'_t+1, r_e, done
          │                      │
          ▼                      ▼
  ┌─────────────────┐    ┌─────────────────┐
  │ Replay Buffer C │    │ Replay Buffer E │
  │ (s,a_c,r_c,s',d)│    │ (s,a_e,r_e,s',d)│
  └─────────────────┘    └─────────────────┘
```

---

9. CÁC ĐIỂM QUAN TRỌNG

9.1. Action Space Consistency:

```
# ✅ CORRECT (Current implementation):
# ClassifierAgent.decode_action():
mid = a_c % n_models
aid = a_c // n_models

# MTDAdversarialIDSEnv.step():
anchor_idx = a_c % n_models   # SAME formula
agg_idx = a_c // n_models     # SAME formula
```

```
# Result: Agent learns correct mapping between actions and outcomes


# ❌ WRONG (Before fix):
# ClassifierAgent.decode_action():
mid = a_c % n_models
aid = a_c // n_models

# MTDAdversarialIDSEnv.step():
anchor_idx = a_c % n_models
agg_idx = (a_c // n_models) % n_aggs  # DIFFERENT formula (extra modulo)

# Result: Agent learns wrong mapping! Action 11 decoded differently!
```

9.2. Reward Signals:

```
# Classifier Agent: Maximize shaped reward
reward_c = w1·ΔAcc - w2·Cost - w3·Switch - w4·Drift
         = 1.0·(+0.1) - 0.2·(0.3) - 0.1·(0.2) - 0.3·(0.05)
         = +0.1 - 0.06 - 0.02 - 0.015
         = +0.005  # Small positive reward

# Environment Agent: Maximize negative accuracy
reward_e = -accuracy_raw
         = -1.0 (if correct prediction)
         = -0.0 (if wrong prediction)
# Environment Agent wants prediction to be WRONG!
```

9.3. Separate Replay Buffers:

```
# WHY separate buffers?
# 1. Different action spaces (model+agg vs attack type)
# 2. Different rewards (shaped vs raw negative)
# 3. Independent learning (no interference)
# 4. Different exploration strategies (can have different ε)

replay_c = ReplayBuffer(capacity=100000)  # For Classifier Agent
replay_e = ReplayBuffer(capacity=100000)  # For Environment Agent
```

9.4. Exploration Strategy:

```
# ε-greedy exploration (shared ε for both agents):
if random.random() < epsilon:
    # EXPLORE: Random action
    a_c = random.randint(0, q_c.n_actions - 1)
    a_e = random.randint(0, q_e.n_actions - 1)
else:
    # EXPLOIT: Best known action
    a_c = q_c(state).argmax()
    a_e = q_e(state).argmax()

# Decay schedule:
epsilon = max(0.01, epsilon * 0.95)  # Decay by 5% each epoch
```

---

10. TÓM TẮT

10.1. Key Takeaways:

1. **3 thực thể khác nhau:**
    - Classifier Agent (Q_c): Defense player
    - Environment Agent (Q_e): Attack player
    - MTDAdversarialIDSEnv: Game referee
2. **Adversarial relationship (Agent vs Agent):**
    - Opposite rewards
    - Same state observation
    - Different action spaces
    - Compete to improve each other
3. **Cooperative relationship (Agent vs Environment):**
    - Consistent action decoding
    - Environment executes actions faithfully
    - Returns rewards and next states
    - Provides state observations
4. **Learning mechanism:**
    - Q-learning for both agents

    - Experience replay for stability
    - ε-greedy exploration
    - Separate replay buffers

5. **Architecture goal:**

    - Robust IDS through adversarial training
    - Adaptive model selection (MTD)
    - Drift-aware learning
    - Efficient ensemble management

10.2. Workflow Summary:

```
1. Initialize:
   - Create environment with model pool
   - Create 2 agents (Q_c, Q_e)
   - Create 2 replay buffers

2. Training loop:
   FOR each epoch:
     Reset environment
     FOR each sample in dataset:
       1. Both agents observe state
       2. Both agents select actions (ε-greedy)
       3. Environment executes both actions
       4. Environment returns rewards & next state
       5. Store transitions in separate buffers
       6. Update networks from replay buffers
       7. Move to next state
     Decay exploration rate

3. Evaluation:
   - Test on holdout set
   - Measure accuracy, F1, AUC
   - Test adversarial robustness
   - Test drift detection
```

---

11. REFERENCES

- **Q-Learning:** Watkins & Dayan (1992)
- **Experience Replay:** Lin (1992)

- **Adversarial Training:** Goodfellow et al. (2014)
- **Moving Target Defense:** Zhuang et al. (2019)
- **Drift Detection:** Gama et al. (2014)

📝 **Ghi chú:** Document này mô tả kiến trúc hiện tại AFTER fixes. Trước khi fix, action decoding không consistent giữa Agent và Environment, gây lỗi học sai.