Hybrid Rule-Based Deep Q-Network Agent for LUX AI

Algorithm 1 Overall Agent Architecture

```
Initialize Components:
Create Q-network (neural network for action evaluation)
Create target network (stable copy of Q-network)
Create replay buffer (stores past experiences)
Set hyperparameters: learning rate, discount factor, exploration rates
Initialize tracking systems: relic memory, productive tiles, unit assignments

for each game do
    Reset game-specific memory
    for each match (3 matches per game) do
        for each time step do
            Observe current state and update relic positions
            for each active unit do
                Generate random number u from [0,1]
                if u < DQN probability then
                    Select action using DQN policy (Algorithm 3)
                else
                    Select action using rule-based policy (Algorithm 4)
                end if
                Augment with attack action if appropriate (Algorithm 5)
            end for
            Execute all unit actions
            Observe rewards and next state
            Update productive tile statistics (Algorithm 6)
            Calculate shaped rewards for each unit (Algorithm 7)
            Store transitions in replay buffer
            if training step then
                Perform network update (Algorithm 8)
            end if
        end for
        Reset match-specific memory
    end for
    Reset all memories for next game
end for
```

Algorithm 2 Neural Network Architecture

Q-Network Structure:
Input layer: Game state features (position, energy, distances, etc.)
Hidden layer 1: 256 neurons with ReLU activation
Hidden layer 2: 256 neurons with ReLU activation
Hidden layer 3: 256 neurons with ReLU activation
Hidden layer 4: 128 neurons with ReLU activation
Output layer: 5 neurons (Q-values for each action)
    Actions: 0=Stay, 1=Up, 2=Right, 3=Down, 4=Left

Target Network:
Same architecture as Q-network
Updated periodically (every N training steps) for learning stability

Algorithm 3 DQN Action Selection (Epsilon-Greedy)

Require: Current state, Q-network, exploration rate

Generate random number u from [0,1]
if $u <$ exploration rate **then**
    return random action   // *Exploration*
else
    Compute Q-values for all actions using Q-network
    return action with highest Q-value   // *Exploitation*
end if

Algorithm 4 Rule-Based Policy

Require: Unit ID, time step, game state, relic positions, productive tiles

```

Extract unit position, energy level, and active status
if unit inactive OR energy below safety threshold then
    return Stay action
end if

Determine number of active units
Calculate unit's rank among active units
Compute exploration ratio based on game phase:
    Early matches: higher exploration (find new relics)
    Late matches: lower exploration (focus on collection)

Assign unit role based on rank and exploration ratio

if unit is COLLECTOR then
    Priority 1: Stay on productive tiles
    if current position is productive tile then
        return Stay action
    end if

    Priority 2: Move to known productive tiles
    if productive tiles exist then
        Find nearest productive tile
        return direction toward nearest productive tile
    end if

    Priority 3: Systematic grid exploration
    if unit has no assignment then
        Assign unit to position in 5×5 grid around relics
    end if
    Calculate target position from assignment
    return direction toward assigned grid position
else
    // Unit is EXPLORER
    Systematic quadrant search
    if unit has no exploration target then
        Assign quadrant based on unit ID (4 quadrants total)
        Generate random target within assigned quadrant
    end if

    if target reached OR stuck at current target then
        Generate new random target in quadrant
    end if
    return direction toward exploration target
end if
```

Algorithm 5 Attack Action Selection

Require: Unit position, game state, unit energy, attack range

Get opponent positions and ally positions
if energy insufficient OR no opponents visible **then**
    return no attack
end if

Initialize best score = $-\infty$ and best target = none
for each possible target location within attack range **do**
    Calculate direct hit score (enemies at exact target)
    Calculate splash score (enemies adjacent to target)
    Calculate friendly fire penalty (allies at or near target)
    Calculate distance penalty (slight preference for closer targets)

    Total score = direct + splash − friendly fire − distance

    if score > best score **then**
        Update best score and best target
    end if
end for

if best score $\geq$ minimum threshold **then**
    return attack toward best target
else
    return no attack
end if

Algorithm 6 Productive Tile Detection

Require: Current state, team points, previous points, position statistics

```
Calculate points gained this step = current points − previous points
Find all unit positions near known relics (within distance 3)

for each position near relics do
    Increment occupancy counter for this position
end for

if points gained > 0 AND units are near relics then
    Distribute gained points among near-relic positions
    for each position that was occupied do
        Update total points earned at this position
        Calculate confidence = total points / times occupied

        if confidence > 0.5 AND occupied at least 3 times then
            Mark position as "productive tile"
        end if
    end for
end if

Periodic cleanup (every 100 steps):
Remove tiles from productive set if confidence drops below 0.7
```

Algorithm 7 Reward Shaping Function

Require: State, action, team reward, productive tiles, statistics

```
Initialize shaped reward = 3.0 × team reward
Extract current position and next position from states

Component 1: Productive Tile Bonuses
if currently on productive tile then
    Add large bonus scaled by tile confidence
    if stayed on same tile (didn't move) then
        Add additional staying bonus
    end if
end if
if moved to a productive tile then
    Add bonus for reaching productive tile
end if

Component 2: Distance Improvement
if productive tiles are known then
    Calculate distance improvement toward nearest productive tile
    Add reward proportional to improvement
else
    Calculate distance improvement toward nearest relic
    Add smaller reward proportional to improvement
end if

Component 3: Energy Management
Calculate energy change
if energy increased then
    Add small bonus for energy recovery
end if
if energy below threshold AND not on productive tile then
    Add penalty for risky low energy state
end if

Component 4: Movement Encouragement
if unit moved AND not leaving productive tile then
    Add small bonus to encourage exploration
end if

Component 5: Survival Bonus
Add small constant bonus for staying alive

Component 6: Leaving Productive Tile Penalty
if left productive tile without going to another productive tile then
    Add penalty scaled by confidence of abandoned tile
end if

return shaped reward
```

Algorithm 8 Training Procedure (Double DQN)

Require: Replay buffer, batch size, Q-network, target network

if not enough experiences in buffer **then**
    return // *Skip training*
end if

Sample random batch of experiences from replay buffer
Each experience contains: (state, action, reward, next state, done flag)

Step 1: Compute current Q-value predictions
Pass states through Q-network
Extract Q-values for actions that were actually taken

Step 2: Compute target Q-values (Double DQN method)
Use Q-network to SELECT best actions for next states
Use target network to EVALUATE those actions
Calculate targets = reward + discount $\times$ evaluated Q-value
If episode ended, target = reward only (no future value)

Step 3: Calculate loss and update
Compute mean squared error between predictions and targets
Calculate gradients of loss with respect to network parameters
Clip gradients to prevent instability
Update network parameters using Adam optimizer

Step 4: Decay exploration rate
Reduce exploration rate: new rate = current rate $\times$ decay factor
Ensure rate doesn't go below minimum value

Step 5: Periodically update target network
if training step is multiple of update frequency **then**
    Copy Q-network parameters to target network
end if

Key Concepts Explained

Hybrid Policy

The agent combines two decision-making strategies:

- **Rule-based policy:** Hand-crafted rules for reliable behavior
- **DQN policy:** Learned through experience via neural networks
- A probability parameter controls which policy is used

Q-Network

A neural network that estimates the "quality" (Q-value) of each action in a given state. Higher Q-values indicate better expected future rewards.

Experience Replay

Past experiences are stored in a buffer and randomly sampled during training. This breaks correlations between consecutive experiences and improves learning stability.

Target Network

A slowly-updated copy of the Q-network used for computing learning targets. This prevents the "moving target" problem and stabilizes training.

Reward Shaping

The basic team reward is augmented with additional signals to guide learning:

- Bonuses for productive tiles
- Distance improvement rewards
- Energy management signals
- Exploration encouragement

Productive Tile Detection

Statistical tracking identifies which positions near relics actually generate points. Confidence increases as more evidence accumulates.