# Technical Implementation Guide: Hierarchical Relational Reinforcement Learning

## Table of Contents

1. [System Architecture Overview](#system-architecture-overview)

2. [Core Components](#core-components)

3. [Implementation Details](#implementation-details)

4. [Training Procedures](#training-procedures)

5. [Evaluation Methods](#evaluation-methods)

6. [Performance Optimization](#performance-optimization)

7. [Troubleshooting Guide](#troubleshooting-guide)

8. [Extension Guidelines](#extension-guidelines)

## System Architecture Overview

### High-Level Architecture

```

┌─────────────────────────────────────────────────────────────┐

│ Environment Interface │

├─────────────────────────────────────────────────────────────┤

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ Goal Decomposer │ │ Constraint │ │

│ │ │ │ Analyzer │ │

│ └─────────────────┘ └─────────────────┘ │

├─────────────────────────────────────────────────────────────┤

│ Hierarchical State Encoder │

├─────────────────────────────────────────────────────────────┤

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ Phase-Adaptive │ │ Experience │ │

│ │ Q-Network │ │ Replay Buffer │ │

│ └─────────────────┘ └─────────────────┘ │

├─────────────────────────────────────────────────────────────┤

│ Action Selection Layer │

└─────────────────────────────────────────────────────────────┘

```

### Data Flow

1. \*\*Input Processing\*\*: Environment state → Relational feature extraction

2. \*\*Goal Management\*\*: Target decomposition → Subgoal tracking

3. \*\*Network Processing\*\*: Features → Phase classification → Q-value computation

4. \*\*Action Selection\*\*: Q-values + constraints → Valid action selection

5. \*\*Learning\*\*: Experience storage → Batch sampling → Network updates

## Core Components

### 1. HierarchicalState Class

\*\*Purpose\*\*: Convert raw environment states into scale-invariant relational features.

\*\*Key Features\*\*:

- Multi-scale gap analysis

- Constraint proximity computation

- Phase identification

- Efficiency metrics

\*\*Critical Implementation Details\*\*:

```python

class HierarchicalState:

def \_\_init\_\_(self, current, target, step, max\_steps, forbidden\_states=None):

self.current = current

self.target = target

self.step = step

self.max\_steps = max\_steps

self.forbidden\_states = forbidden\_states or set()

def to\_features(self):

"""

Returns 12-dimensional feature vector:

[0] progress\_ratio: current/target (0-1+)

[1] remaining\_ratio: (target-current)/target (0-1)

[2] time\_ratio: step/max\_steps (0-1)

[3] log\_gap: log-scaled distance to target

[4] gap\_magnitude: relative gap size

[5] is\_close: binary flag for proximity

[6] is\_far: binary flag for distance

[7] danger\_proximity: distance to nearest forbidden state

[8] constraint\_pressure: path blockage measure

[9] phase: problem-solving phase (0/1/2)

[10] theoretical\_min\_steps: optimal step count

[11] efficiency\_ratio: time pressure measure

"""

if self.target == 0:

return np.zeros(12)

# Core relational features

progress\_ratio = self.current / self.target

remaining\_ratio = (self.target - self.current) / self.target

time\_ratio = self.step / self.max\_steps

# Multi-scale analysis

gap = abs(self.target - self.current)

log\_gap = math.log(gap + 1) / math.log(self.target + 1)

gap\_magnitude = gap / self.target

# Strategic indicators

is\_close = 1.0 if gap <= 10 else 0.0

is\_far = 1.0 if gap >= self.target \* 0.5 else 0.0

# Constraint analysis

danger\_proximity = self.\_compute\_danger\_proximity()

constraint\_pressure = self.\_compute\_constraint\_pressure()

# Phase and efficiency

phase = self.\_identify\_phase()

theoretical\_min\_steps = self.\_compute\_min\_steps()

efficiency\_ratio = theoretical\_min\_steps / (self.max\_steps - self.step + 1)

return np.array([

progress\_ratio, remaining\_ratio, time\_ratio,

log\_gap, gap\_magnitude, is\_close, is\_far,

danger\_proximity, constraint\_pressure,

phase, theoretical\_min\_steps, efficiency\_ratio

])

```

\*\*Performance Considerations\*\*:

- Feature computation is O(|forbidden\_states|)

- Cache expensive computations when possible

- Normalize all features to [0,1] range for neural network stability

### 2. HierarchicalQNetwork Architecture

\*\*Purpose\*\*: Multi-head neural network with phase-specific specialization.

\*\*Architecture Specifications\*\*:

```python

class HierarchicalQNetwork(nn.Module):

def \_\_init\_\_(self, state\_dim=12, action\_dim=3, hidden\_dim=256):

super().\_\_init\_\_()

# Shared feature extractor (critical for transfer learning)

self.feature\_extractor = nn.Sequential(

nn.Linear(state\_dim, hidden\_dim),

nn.ReLU(),

nn.Dropout(0.1), # Prevent overfitting on small datasets

nn.Linear(hidden\_dim, hidden\_dim),

nn.ReLU(),

nn.Dropout(0.1)

)

# Phase-specific heads (enables specialized strategies)

self.exploration\_head = self.\_build\_head(hidden\_dim, action\_dim)

self.navigation\_head = self.\_build\_head(hidden\_dim, action\_dim)

self.precision\_head = self.\_build\_head(hidden\_dim, action\_dim)

# Phase classifier (learns to identify problem phases)

self.phase\_classifier = nn.Sequential(

nn.Linear(hidden\_dim, 32),

nn.ReLU(),

nn.Linear(32, 3),

nn.Softmax(dim=-1)

)

def \_build\_head(self, input\_dim, output\_dim):

return nn.Sequential(

nn.Linear(input\_dim, input\_dim // 2),

nn.ReLU(),

nn.Linear(input\_dim // 2, output\_dim)

)

def forward(self, state\_features):

# Extract shared representations

features = self.feature\_extractor(state\_features)

# Classify current phase

phase\_probs = self.phase\_classifier(features)

# Compute phase-specific Q-values

exploration\_q = self.exploration\_head(features)

navigation\_q = self.navigation\_head(features)

precision\_q = self.precision\_head(features)

# Weighted combination based on phase probabilities

q\_values = (phase\_probs[:, 0:1] \* exploration\_q +

phase\_probs[:, 1:2] \* navigation\_q +

phase\_probs[:, 2:3] \* precision\_q)

return q\_values, phase\_probs

```

\*\*Design Rationale\*\*:

- \*\*Shared Feature Extractor\*\*: Enables transfer learning across phases

- \*\*Phase-Specific Heads\*\*: Allows specialized strategies for different problem stages

- \*\*Soft Attention\*\*: Phase probabilities enable smooth transitions between strategies

- \*\*Dropout\*\*: Prevents overfitting on small training datasets

### 3. GoalDecompositionAgent

\*\*Purpose\*\*: Main agent class coordinating all components.

\*\*Key Responsibilities\*\*:

- Goal decomposition and subgoal management

- Experience collection and replay

- Network training and target updates

- Action selection with constraint handling

\*\*Critical Methods\*\*:

```python

class GoalDecompositionAgent:

def decompose\_target(self, current, target):

"""

Automatically decompose large targets into manageable subgoals.

Strategy:

- Targets ≤ 50: Direct approach

- Targets > 50: Create intermediate waypoints

- Waypoint spacing: ~75 units (empirically optimal)

"""

gap = abs(target - current)

if gap <= 50:

return [target]

num\_subgoals = max(2, gap // 75)

step\_size = gap // num\_subgoals

direction = 1 if target > current else -1

subgoals = []

for i in range(1, num\_subgoals):

subgoal = current + (step\_size \* i \* direction)

subgoals.append(subgoal)

subgoals.append(target)

return subgoals

def choose\_action(self, current, target, step, max\_steps, forbidden\_states=None, training=True):

"""

Action selection with hierarchical reasoning and constraint handling.

"""

# Update subgoals if needed

if not self.subgoal\_stack:

self.subgoal\_stack = self.decompose\_target(current, target)

self.current\_subgoal = self.subgoal\_stack.pop(0)

# Check subgoal completion

if current == self.current\_subgoal and self.subgoal\_stack:

self.current\_subgoal = self.subgoal\_stack.pop(0)

# Use current subgoal for decision making

working\_target = self.current\_subgoal if self.current\_subgoal else target

# Adaptive exploration based on problem difficulty

difficulty = abs(target - current)

epsilon = self.epsilon\_schedule(self.step\_count, difficulty)

if training and random.random() < epsilon:

return random.choice(self.actions)

# Neural network decision

state\_features = self.get\_state\_features(current, working\_target, step, max\_steps, forbidden\_states)

state\_tensor = torch.FloatTensor(state\_features).unsqueeze(0)

with torch.no\_grad():

q\_values, phase\_probs = self.q\_network(state\_tensor)

# Constraint-aware action selection

valid\_actions = []

for i, action in enumerate(self.actions):

next\_state = current + action

if forbidden\_states and next\_state in forbidden\_states:

continue

valid\_actions.append((i, action))

if not valid\_actions:

# Emergency backtracking

return -self.actions[0] if -self.actions[0] in self.actions else self.actions[0]

# Select highest-value valid action

best\_idx = -1

best\_value = float('-inf')

for idx, action in valid\_actions:

if q\_values[0, idx] > best\_value:

best\_value = q\_values[0, idx]

best\_idx = idx

return self.actions[best\_idx]

```

\*\*Epsilon Schedule Design\*\*:

```python

def \_create\_epsilon\_schedule(self):

"""

Adaptive exploration schedule that increases for harder problems.

"""

def epsilon\_fn(step, target\_difficulty):

base\_epsilon = 0.1

difficulty\_bonus = min(0.2, target\_difficulty / 1000)

decay = max(0.01, base\_epsilon \* (0.995 \*\* (step / 100)))

return decay + difficulty\_bonus

return epsilon\_fn

```

## Implementation Details

### 1. Feature Engineering Considerations

\*\*Scale Invariance\*\*:

- All features normalized to [0,1] range

- Ratios used instead of absolute values

- Logarithmic scaling for gap measures

\*\*Constraint Handling\*\*:

- Danger proximity: `1/(min\_distance + 1)`

- Constraint pressure: Fraction of path blocked

- Emergency backtracking when all actions forbidden

\*\*Phase Identification\*\*:

```python

def \_identify\_phase(self):

"""

Phase classification:

0.0 = Exploration (gap > 70% of target)

1.0 = Navigation (gap 10-70% of target)

2.0 = Precision (gap < 10 units)

"""

gap = abs(self.target - self.current)

if gap > self.target \* 0.7:

return 0.0 # Exploration phase

elif gap > 10:

return 1.0 # Navigation phase

else:

return 2.0 # Precision phase

```

### 2. Neural Network Training Details

\*\*Loss Function\*\*:

```python

# Smooth L1 loss for stability

loss = nn.SmoothL1Loss()(current\_q\_values.squeeze(), target\_q\_values)

```

\*\*Gradient Clipping\*\*:

```python

# Prevent exploding gradients

torch.nn.utils.clip\_grad\_norm\_(self.q\_network.parameters(), 1.0)

```

\*\*Target Network Updates\*\*:

```python

# Soft updates every 200 steps

if self.step\_count % 200 == 0:

self.target\_network.load\_state\_dict(self.q\_network.state\_dict())

```

\*\*Experience Replay Configuration\*\*:

- Buffer size: 10,000 experiences

- Batch size: 64

- Sampling: Uniform random (can be extended to prioritized)

### 3. Reward Engineering

\*\*Reward Function Design\*\*:

```python

def compute\_reward(current, next\_current, target, step, forbidden\_states):

"""

Multi-component reward function for efficient learning.

"""

if next\_current == target:

return 100 - step # Success with time bonus

elif next\_current in forbidden\_states:

return -50 # Constraint violation penalty

elif abs(next\_current - target) < abs(current - target):

return 10 # Progress reward

elif next\_current > target:

return -20 # Overshoot penalty

else:

return -1 # Time penalty

```

\*\*Reward Shaping Principles\*\*:

- Dense rewards for progress tracking

- Heavy penalties for constraint violations

- Time bonuses for efficient solutions

- Balanced exploration vs exploitation

## Training Procedures

### 1. Curriculum Learning Protocol

\*\*Stage 1: Foundation (Episodes 0-1000)\*\*

```python

targets = range(5, 11) # Simple targets

forbidden\_count = 1-2 # Minimal constraints

episode\_length = 20 # Short episodes

```

\*\*Stage 2: Expansion (Episodes 1000-2500)\*\*

```python

targets = range(12, 31) # Medium targets

forbidden\_count = 2-3 # Moderate constraints

episode\_length = 25 # Longer episodes

```

\*\*Stage 3: Generalization (Episodes 2500-4000)\*\*

```python

targets = range(20, 101) # Large targets

forbidden\_count = 3-5 # Complex constraints

episode\_length = 30 # Full episodes

```

\*\*Stage 4: Mixed Training (Episodes 4000-5000)\*\*

```python

targets = range(5, 101) # Full range

forbidden\_count = 1-5 # Variable constraints

episode\_length = 30 # Adaptive length

```

### 2. Training Loop Implementation

```python

def train\_agent(agent, total\_episodes=5000):

"""

Complete training procedure with curriculum learning.

"""

curriculum = [

{'targets': range(5, 11), 'episodes': 1000, 'max\_steps': 20},

{'targets': range(12, 31), 'episodes': 1500, 'max\_steps': 25},

{'targets': range(20, 101), 'episodes': 1500, 'max\_steps': 30},

{'targets': range(5, 101), 'episodes': 1000, 'max\_steps': 30}

]

episode\_count = 0

for stage\_idx, stage in enumerate(curriculum):

print(f"Training Stage {stage\_idx + 1}: Targets {list(stage['targets'])}")

for episode in range(stage['episodes']):

# Sample problem

target = random.choice(list(stage['targets']))

forbidden\_count = random.randint(1, min(5, target // 20))

forbidden\_states = set(random.sample(range(1, target), forbidden\_count))

# Run episode

current = 0

step = 0

max\_steps = stage['max\_steps']

agent.reset\_subgoals()

while step < max\_steps and current != target:

action = agent.choose\_action(current, target, step, max\_steps, forbidden\_states)

next\_current = current + action

reward = compute\_reward(current, next\_current, target, step, forbidden\_states)

done = (next\_current == target) or (step + 1 >= max\_steps)

agent.store\_experience(current, target, step, max\_steps, action,

reward, next\_current, step + 1, done, forbidden\_states)

current = next\_current

step += 1

# Learning updates

if episode\_count % 10 == 0:

agent.learn\_from\_experience()

episode\_count += 1

# Progress tracking

if episode\_count % 500 == 0:

success\_rate = evaluate\_agent(agent, test\_targets=[50, 75, 100])

print(f"Episode {episode\_count}: Success rate = {success\_rate:.1%}")

print("Training complete!")

```

### 3. Hyperparameter Settings

\*\*Network Architecture\*\*:

- Hidden dimensions: 256 units

- Dropout rate: 0.1

- Activation: ReLU

- Output layers: 3 (for actions {1,3,5})

\*\*Training Parameters\*\*:

- Learning rate: 0.0005

- Batch size: 64

- Discount factor (γ): 0.99

- Target network update frequency: 200 steps

\*\*Exploration Schedule\*\*:

- Base epsilon: 0.1

- Decay rate: 0.995 per 100 steps

- Minimum epsilon: 0.01

- Difficulty bonus: up to 0.2 for large targets

## Evaluation Methods

### 1. Generalization Testing Protocol

```python

def test\_generalization(agent, test\_targets):

"""

Comprehensive generalization evaluation.

"""

results = {}

for target in test\_targets:

print(f"Testing target {target}")

# Generate novel forbidden states

forbidden\_count = random.randint(3, 6)

forbidden\_states = set(random.sample(range(1, target), forbidden\_count))

# Calculate theoretical minimum

theoretical\_min = math.ceil(target / 5)

max\_steps = int(theoretical\_min \* 1.5) # 50% buffer

# Run test episode

agent.reset\_subgoals()

current = 0

step = 0

path = [current]

while step < max\_steps and current != target:

action = agent.choose\_action(current, target, step, max\_steps,

forbidden\_states, training=False)

current += action

step += 1

path.append(current)

# Record results

success = (current == target)

efficiency = (theoretical\_min / step \* 100) if success else 0

results[target] = {

'success': success,

'steps\_taken': step,

'theoretical\_min': theoretical\_min,

'efficiency': efficiency,

'path': path,

'forbidden\_states': forbidden\_states

}

print(f" Result: {'SUCCESS' if success else 'FAILED'}")

print(f" Steps: {step}/{max\_steps}")

print(f" Efficiency: {efficiency:.1f}%")

return results

```

### 2. Performance Metrics

\*\*Primary Metrics\*\*:

- \*\*Success Rate\*\*: Percentage of problems solved within step limit

- \*\*Step Efficiency\*\*: (Theoretical minimum steps) / (Actual steps taken)

- \*\*Generalization Ratio\*\*: (Largest solved target) / (Largest training target)

\*\*Secondary Metrics\*\*:

- \*\*Strategy Consistency\*\*: Frequency of coarse-to-fine pattern usage

- \*\*Constraint Handling\*\*: Success rate on problems with forbidden states

- \*\*Adaptation Speed\*\*: Steps to solution on novel constraint configurations

\*\*Comparative Baselines\*\*:

- Random policy

- Greedy policy (always choose largest valid action)

- Standard DQN without hierarchical features

- Human performance (when available)

### 3. Ablation Study Framework

```python

def run\_ablation\_study():

"""

Systematic evaluation of component contributions.

"""

configurations = [

{

'name': 'Full System',

'hierarchical\_state': True,

'goal\_decomposition': True,

'phase\_adaptive': True,

'curriculum': True

},

{

'name': 'No Hierarchical State',

'hierarchical\_state': False,

'goal\_decomposition': True,

'phase\_adaptive': True,

'curriculum': True

},

{

'name': 'No Goal Decomposition',

'hierarchical\_state': True,

'goal\_decomposition': False,

'phase\_adaptive': True,

'curriculum': True

},

{

'name': 'No Phase Adaptation',

'hierarchical\_state': True,

'goal\_decomposition': True,

'phase\_adaptive': False,

'curriculum': True

},

{

'name': 'No Curriculum',

'hierarchical\_state': True,

'goal\_decomposition': True,

'phase\_adaptive': True,

'curriculum': False

}

]

results = {}

test\_targets = [123, 278, 431]

for config in configurations:

print(f"Testing configuration: {config['name']}")

agent = create\_agent\_with\_config(config)

train\_agent\_with\_config(agent, config)

results[config['name']] = test\_generalization(agent, test\_targets)

return results

```

## Performance Optimization

### 1. Memory Management

\*\*Experience Buffer Optimization\*\*:

```python

class PrioritizedReplayBuffer:

"""

Memory-efficient experience replay with prioritization.

"""

def \_\_init\_\_(self, capacity=10000, alpha=0.6, beta=0.4):

self.capacity = capacity

self.alpha = alpha

self.beta = beta

self.buffer = []

self.priorities = np.zeros((capacity,))

self.position = 0

def push(self, experience, td\_error):

"""Store experience with priority based on TD error."""

priority = (abs(td\_error) + 1e-6) \*\* self.alpha

if len(self.buffer) < self.capacity:

self.buffer.append(experience)

else:

self.buffer[self.position] = experience

self.priorities[self.position] = priority

self.position = (self.position + 1) % self.capacity

def sample(self, batch\_size):

"""Sample batch with priority-based selection."""

if len(self.buffer) == self.capacity:

priorities = self.priorities

else:

priorities = self.priorities[:self.position]

probabilities = priorities \*\* self.alpha

probabilities /= probabilities.sum()

indices = np.random.choice(len(self.buffer), batch\_size, p=probabilities)

experiences = [self.buffer[idx] for idx in indices]

# Importance sampling weights

weights = (len(self.buffer) \* probabilities[indices]) \*\* (-self.beta)

weights /= weights.max()

return experiences, weights, indices

```

### 2. Computational Efficiency

\*\*Feature Computation Optimization\*\*:

```python

@lru\_cache(maxsize=1000)

def compute\_cached\_features(current, target, step, max\_steps, forbidden\_tuple):

"""

Cache frequently computed features to reduce overhead.

"""

forbidden\_states = set(forbidden\_tuple) if forbidden\_tuple else set()

state = HierarchicalState(current, target, step, max\_steps, forbidden\_states)

return state.to\_features()

```

\*\*Batch Processing\*\*:

```python

def batch\_compute\_q\_values(agent, states\_batch):

"""

Process multiple states simultaneously for efficiency.

"""

features\_batch = torch.FloatTensor([state.to\_features() for state in states\_batch])

with torch.no\_grad():

q\_values\_batch, phase\_probs\_batch = agent.q\_network(features\_batch)

return q\_values\_batch, phase\_probs\_batch

```

### 3. Training Acceleration

\*\*Mixed Precision Training\*\*:

```python

from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

def optimized\_learning\_step(agent, batch):

"""

Use mixed precision for faster training.

"""

with autocast():

loss = agent.compute\_loss(batch)

agent.optimizer.zero\_grad()

scaler.scale(loss).backward()

scaler.unscale\_(agent.optimizer)

torch.nn.utils.clip\_grad\_norm\_(agent.q\_network.parameters(), 1.0)

scaler.step(agent.optimizer)

scaler.update()

```

\*\*Early Stopping\*\*:

```python

def train\_with\_early\_stopping(agent, patience=500):

"""

Stop training when performance plateaus.

"""

best\_performance = 0

patience\_counter = 0

for episode in range(max\_episodes):

# Training step

train\_episode(agent)

# Evaluation

if episode % 100 == 0:

current\_performance = evaluate\_agent(agent)

if current\_performance > best\_performance:

best\_performance = current\_performance

patience\_counter = 0

save\_checkpoint(agent, f"best\_model\_episode\_{episode}.pt")

else:

patience\_counter += 100

if patience\_counter >= patience:

print(f"Early stopping at episode {episode}")

break

```

## Troubleshooting Guide

### 1. Common Training Issues

\*\*Problem: Agent gets stuck in loops\*\*

```

Symptoms: Repeating same actions, no progress toward target

Causes:

- Insufficient exploration

- Poor reward signal

- Overfitting to specific patterns

Solutions:

- Increase epsilon during training

- Add randomness to forbidden state generation

- Implement action history penalty

- Use entropy regularization in loss function

```

\*\*Problem: Poor generalization to large targets\*\*

```

Symptoms: Success on training targets, failure on test targets >2x training size

Causes:

- Absolute state representation

- Insufficient curriculum progression

- Network capacity limitations

Solutions:

- Verify relational feature computation

- Extend curriculum to larger targets gradually

- Increase network hidden dimensions

- Add more diverse training examples

```

\*\*Problem: Constraint violations\*\*

```

Symptoms: Agent enters forbidden states despite penalties

Causes:

- Weak constraint penalties

- Poor constraint feature representation

- Exploration vs exploitation imbalance

Solutions:

- Increase constraint violation penalty (-50 → -100)

- Improve danger proximity computation

- Add constraint violation to state features

- Use hard constraints in action selection

```

### 2. Performance Debugging

\*\*Training Diagnostics\*\*:

```python

def diagnose\_training(agent, episode\_data):

"""

Comprehensive training analysis.

"""

print("=== TRAINING DIAGNOSTICS ===")

# Loss progression

losses = [ep['loss'] for ep in episode\_data if 'loss' in ep]

print(f"Loss trend: {np.mean(losses[-100:]):.4f} (recent 100 episodes)")

# Success rate progression

success\_rates = [ep['success'] for ep in episode\_data[-500:]]

print(f"Recent success rate: {np.mean(success\_rates):.1%}")

# Strategy analysis

coarse\_to\_fine\_usage = analyze\_strategy\_patterns(episode\_data)

print(f"Coarse-to-fine strategy usage: {coarse\_to\_fine\_usage:.1%}")

# Phase distribution

phase\_distribution = analyze\_phase\_usage(agent)

print(f"Phase distribution: {phase\_distribution}")

# Network weight analysis

analyze\_network\_weights(agent.q\_network)

```

\*\*Performance Profiling\*\*:

```python

import cProfile

import pstats

def profile\_training\_step():

"""

Profile performance bottlenecks.

"""

profiler = cProfile.Profile()

profiler.enable()

# Run training step

train\_episode(agent)

profiler.disable()

stats = pstats.Stats(profiler)

stats.sort\_stats('cumulative')

stats.print\_stats(10) # Top 10 time consumers

```

### 3. Model Validation

\*\*Sanity Checks\*\*:

```python

def validate\_model(agent):

"""

Comprehensive model validation.

"""

checks = []

# Feature computation check

test\_state = HierarchicalState(50, 100, 10, 30, {25, 75})

features = test\_state.to\_features()

checks.append(("Features in [0,1]", all(0 <= f <= 1 for f in features[:9])))

# Network output check

q\_values, phase\_probs = agent.q\_network(torch.FloatTensor(features).unsqueeze(0))

checks.append(("Q-values finite", torch.isfinite(q\_values).all()))

checks.append(("Phase probs sum to 1", abs(phase\_probs.sum() - 1.0) < 1e-6))

# Goal decomposition check

subgoals = agent.decompose\_target(0, 200)

checks.append(("Subgoals increasing", all(subgoals[i] < subgoals[i+1] for i in range(len(subgoals)-1))))

# Action selection check

action = agent.choose\_action(50, 100, 10, 30, training=False)

checks.append(("Valid action", action in agent.actions))

print("=== MODEL VALIDATION ===")

for check\_name, result in checks:

print(f"{check\_name}: {'PASS' if result else 'FAIL'}")

return all(result for \_, result in checks)

```

## Extension Guidelines

### 1. Adding New Action Types

```python

def extend\_action\_space(agent, new\_actions):

"""

Dynamically extend agent's action space.

"""

old\_action\_dim = len(agent.actions)

agent.actions.extend(new\_actions)

new\_action\_dim = len(agent.actions)

# Extend network output layers

for head\_name in ['exploration\_head', 'navigation\_head', 'precision\_head']:

head = getattr(agent.q\_network, head\_name)

old\_layer = head[-1]

# Create new layer with extended output dimension

new\_layer = nn.Linear(old\_layer.in\_features, new\_action\_dim)

# Copy old weights

with torch.no\_grad():

new\_layer.weight[:old\_action\_dim] = old\_layer.weight

new\_layer.bias[:old\_action\_dim] = old\_layer.bias

# Initialize new action weights

nn.init.xavier\_uniform\_(new\_layer.weight[old\_action\_dim:])

nn.init.zeros\_(new\_layer.bias[old\_action\_dim:])

# Replace layer

head[-1] = new\_layer

# Update optimizer to include new parameters

agent.optimizer = optim.Adam(agent.q\_network.parameters(), lr=agent.optimizer.param\_groups[0]['lr'])

```

### 2. Multi-Objective Extensions

```python

class MultiObjectiveState(HierarchicalState):

"""

Extended state representation for multi-objective problems.

"""

def \_\_init\_\_(self, current, targets, weights, step, max\_steps, forbidden\_states=None):

self.current = current

self.targets = targets # Multiple targets

self.weights = weights # Target priorities

self.step = step

self.max\_steps = max\_steps

self.forbidden\_states = forbidden\_states or set()

def to\_features(self):

"""

Multi-objective feature computation.

"""

base\_features = []

for target, weight in zip(self.targets, self.weights):

# Compute features for each target

single\_target\_features = super(MultiObjectiveState, self).\_\_init\_\_(

self.current, target, self.step, self.max\_steps, self.forbidden\_states

).to\_features()

# Weight by target importance

weighted\_features = single\_target\_features \* weight

base\_features.extend(weighted\_features)

# Add global features

global\_features = [

len(self.targets), # Number of objectives

np.std(self.weights), # Priority variance

min(abs(self.current - t) for t in self.targets), # Distance to nearest target

]

return np.concatenate([base\_features, global\_features])

```

### 3. Continuous Action Spaces

```python

class ContinuousActionAgent(GoalDecompositionAgent):

"""

Extension to continuous action spaces.

"""

def \_\_init\_\_(self, action\_dim=1, action\_range=(-10, 10)):

super().\_\_init\_\_()

self.action\_dim = action\_dim

self.action\_range = action\_range

# Replace discrete Q-network with actor-critic

self.actor = self.\_build\_actor()

self.critic = self.\_build\_critic()

def \_build\_actor(self):

return nn.Sequential(

nn.Linear(12, 256),

nn.ReLU(),

nn.Linear(256, 128),

nn.ReLU(),

nn.Linear(128, self.action\_dim),

nn.Tanh() # Output in [-1, 1]

)

def \_build\_critic(self):

return nn.Sequential(

nn.Linear(12 + self.action\_dim, 256),

nn.ReLU(),

nn.Linear(256, 128),

nn.ReLU(),

nn.Linear(128, 1)

)

def choose\_action(self, current, target, step, max\_steps, forbidden\_states=None, training=True):

"""

Continuous action selection with constraint handling.

"""

state\_features = self.get\_state\_features(current, target, step, max\_steps, forbidden\_states)

state\_tensor = torch.FloatTensor(state\_features).unsqueeze(0)

# Get action from actor network

with torch.no\_grad():

action\_normalized = self.actor(state\_tensor)

# Scale to action range

action\_range\_size = self.action\_range[1] - self.action\_range[0]

action = action\_normalized \* (action\_range\_size / 2) + (self.action\_range[0] + self.action\_range[1]) / 2

# Constraint checking for continuous actions

proposed\_next\_state = current + action.item()

if forbidden\_states:

# Check if proposed state violates constraints

for forbidden in forbidden\_states:

if abs(proposed\_next\_state - forbidden) < 0.5: # Continuous constraint violation threshold

# Adjust action to avoid constraint

if proposed\_next\_state > forbidden:

action = torch.tensor([[forbidden + 0.5 - current]])

else:

action = torch.tensor([[forbidden - 0.5 - current]])

break

return action.item()

```

### 4. Multi-Agent Coordination

```python

class CoordinatedAgent(GoalDecompositionAgent):

"""

Multi-agent extension with coordination mechanisms.

"""

def \_\_init\_\_(self, agent\_id, num\_agents, communication\_dim=32):

super().\_\_init\_\_()

self.agent\_id = agent\_id

self.num\_agents = num\_agents

self.communication\_dim = communication\_dim

# Communication components

self.message\_encoder = nn.Linear(12, communication\_dim)

self.message\_decoder = nn.Linear(communication\_dim \* num\_agents, 64)

# Modified Q-network with communication input

self.q\_network = self.\_build\_communicative\_network()

def \_build\_communicative\_network(self):

"""

Network architecture with communication channels.

"""

return nn.Sequential(

nn.Linear(12 + 64, 256), # State + communication features

nn.ReLU(),

nn.Linear(256, 256),

nn.ReLU(),

nn.Linear(256, len(self.actions))

)

def choose\_action(self, current, target, step, max\_steps, forbidden\_states=None,

other\_agents\_messages=None, training=True):

"""

Action selection with inter-agent communication.

"""

state\_features = self.get\_state\_features(current, target, step, max\_steps, forbidden\_states)

# Encode own message

own\_message = self.message\_encoder(torch.FloatTensor(state\_features))

# Process messages from other agents

if other\_agents\_messages is not None:

all\_messages = torch.cat([own\_message] + other\_agents\_messages, dim=0)

else:

all\_messages = own\_message.repeat(self.num\_agents, 1)

communication\_features = self.message\_decoder(all\_messages.flatten())

# Combine state and communication features

combined\_features = torch.cat([torch.FloatTensor(state\_features), communication\_features])

# Action selection

with torch.no\_grad():

q\_values = self.q\_network(combined\_features.unsqueeze(0))

# Constraint-aware selection (same as base class)

return self.\_select\_constrained\_action(current, q\_values, forbidden\_states)

```

---

This technical documentation provides a comprehensive guide for implementing, training, evaluating, and extending the hierarchical relational reinforcement learning system. The modular design allows for easy experimentation with different components and extensions to new domains.