Emotion-Based MCTS Reinforcement Learning System - Detailed Design Specification

Overview

This system is an MCTS-based reinforcement learning algorithm using emotion tokens. While traditional MCTS explores concrete actions a, this approach achieves generalization by exploring abstract emotions e.

---

1. MCTS Algorithm

1.1 Basic Structure

Each node n maintains the following information:

$$n = \{N(n), W(n), Q(n), P(n), \text{children}(n)\}$$

- $N(n)$: Visit count
- $W(n)$: Cumulative value
- $Q(n) = W(n)/N(n)$: Average value
- $P(n)$: Prior probability (Policy head output)
- $\text{children}(n)$: Set of child nodes

1.2 UCB Selection Formula

At node n, the next emotion e is selected as:

$$e^* = \arg\max_{e \in \mathcal{E}} \left[Q(n, e) + c_{\text{puct}} \cdot P(n, e) \cdot \frac{\sqrt{N(n)}}{1 + N(n, e)} \right]$$

Where:

- $\mathcal{E} = \{0, 1, \ldots, 7\}$: Emotion set (8 types)
- $Q(n, e)$: Average value of emotion e
- $P(n, e)$: Prior probability of emotion e
- $N(n, e)$: Visit count of emotion e
- c_{puct}: Exploration coefficient (default: 1.0)

1.3 MCTS Search Flow

Input: Current sequence $s = [s_0, a_0, e_0, r_0, \ldots, s_t]$

Output: Visit distribution $\pi = [N(e_0), N(e_1), \ldots, N(e_7)]$ (normalized)

Algorithm

```
function MCTS_SEARCH(model, sequence, n_simulations):
  # 1. Create root node
  root ← CREATE_NODE()

  # 2. Expand root node
  outputs ← model.forward(sequence)
  root.P ← outputs.policy  # [8]
  root.children ← {}

  for e in {0, 1, ..., 7}:
    child ← CREATE_NODE()
    child.parent ← root
    child.emotion ← e

    # Store model predictions
    child.action_logits ← outputs.actions[e]    # [4]
    child.predicted_state ← outputs.dynamics[e]  # [16]
    child.value ← outputs.values[e]           # scalar

    root.children[e] ← child

  # 3. Simulations
  for i in 1 to n_simulations:
    SIMULATE(root, sequence, model)

  # 4. Compute visit distribution
  visit_counts ← [N(root.children[e]) for e in {0, ..., 7}]
  π ← NORMALIZE(visit_counts)

  return π
```

1.4 Simulation Procedure

```
function SIMULATE(root, real_sequence, model):
  node ← root
  path ← []  # Search path
  imagined_state ← real_sequence.last_state  # Imagined starting state
  depth ← 0
  MAX_DEPTH ← 3
```

```
# === Selection Phase ===
while NOT node.is_leaf() AND depth < MAX_DEPTH:
    # UCB selection
    e ← SELECT_CHILD_UCB(node)

    path.append((node, e))

    # Get next imagined state
    imagined_state ← node.children[e].predicted_state

    node ← node.children[e]
    depth ← depth + 1

# === Expansion Phase ===
if node.visits > 0 AND depth < MAX_DEPTH:
    # Construct imagined sequence
    imagined_seq ← APPEND_IMAGINED_STATE(real_sequence, imagined_state)

    # Evaluate with model
    outputs ← model.forward(imagined_seq)

    # Expand node
    for e in {0, ..., 7}:
        child ← CREATE_NODE()
        child.parent ← node
        child.action_logits ← outputs.actions[e]
        child.predicted_state ← outputs.dynamics[e]
        child.value ← outputs.values[e]
        node.children[e] ← child

    # Leaf value (average Value across all emotions)
    leaf_value ← MEAN(outputs.values)
else:
    # If unexpanded, use parent's predicted Value
    if path is not empty:
        parent, e ← path[-1]
        leaf_value ← parent.children[e].value
    else:
        leaf_value ← 0.0

# === Backpropagation Phase ===
for (node, e) in REVERSED(path):
    child ← node.children[e]
    child.N ← child.N + 1
```

```
    child.W ← child.W + leaf_value
    child.Q ← child.W / child.N
```

1.5 Imagined Sequence Construction

```
function APPEND_IMAGINED_STATE(real_seq, imagined_state):
  # Copy real sequence
  new_seq ← COPY(real_seq)

  # Add imagined state (with dummy a, e, r)
  new_seq.states.append(imagined_state)
  new_seq.actions.append(0)     # Dummy
  new_seq.emotions.append(0)    # Dummy
  new_seq.rewards.append(0.0)   # Dummy

  return new_seq
```

1.6 Important Invariants

MCTS imagined sequences are NOT added to the true sequence

```
# During simulation
imagined_seq = real_seq + [imagined_state, dummy_a, dummy_e, dummy_r]  # Temporary

# After simulation, discarded

# Only actual experience is added to true sequence
e_selected ← ARGMAX(visit_counts)
a_selected ← model.action_heads[e_selected](state)
s_next, r_next ← env.step(a_selected)

real_seq ← real_seq + [a_selected, e_selected, r_next, s_next]  # Only this persists
```

---

2. Learning Algorithm

2.1 Four Outputs and Their Roles

The model outputs the following in a single forward pass:

1. **Policy** $\pi_\theta(e|s)$: Emotion probability distribution $[8]$
2. **Value** $V_\theta^e(s)$: Expected value of emotion e $[8 \times 1]$
3. **Dynamics** $Z_\theta^e(s, a^e)$: Next state prediction for emotion e $[8 \times 16]$
4. **Action** $A_\theta^e(s)$: Action logits for emotion e $[8 \times 4]$

2.2 Learning Objective Functions

Each output is trained with a different objective function:

2.2.1 Policy Learning (Output 1)

Objective: Imitate MCTS visit distribution

$$\mathcal{L}_\pi = -\sum_{e=0}^{7} \pi_{\text{MCTS}}(e) \log \pi_\theta(e|s)$$

Where:

- $\pi_{\text{MCTS}}(e) = N(e) / \sum_{e'} N(e')$: MCTS visit distribution (normalized)
- $\pi_\theta(e|s)$: Model's Policy output (softmax applied)

Gradient: Policy head and Transformer

$$\nabla_{\theta_\pi, \theta_{\text{transformer}}} \mathcal{L}_\pi$$

2.2.2 Value Learning (Output 2)

Objective: Predict actual returns

$$\mathcal{L}_V = \left(V_\theta^{e_{\text{selected}}}(s) - G_{\text{actual}}\right)^2$$

Where:

- e_{selected}: Emotion selected by MCTS
- $G_{\text{actual}} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$: Actual discounted return
- $\gamma = 0.99$: Discount factor

Gradient: Value head and Transformer

$$\nabla_{\theta_V, \theta_{\text{transformer}}} \mathcal{L}_V$$

2.2.3 Dynamics Learning (Output 3)

Objective: Predict next actual state

$$\mathcal{L}_Z = \left\| Z_\theta^{e_{\text{selected}}}(s, a^{e_{\text{selected}}}) - s_{\text{next}}^{\text{actual}} \right\|_2^2$$

Where:

- $a^{e_{\text{selected}}}$: Action head output (**frozen, no gradient**)
- $s_{\text{next}}^{\text{actual}}$: Actual next state from environment

Important: Action head is frozen

$$a^{e_{\text{selected}}} = \text{stopgrad}(A_\theta^{e_{\text{selected}}}(s))$$

Gradient: Dynamics head and Transformer

$$\nabla_{\theta_Z, \theta_{\text{transformer}}} \mathcal{L}_Z$$

2.2.4 Action Learning (Output 4)

Objective: Reproduce the state predicted by Dynamics

$$\mathcal{L}_A = \| Z_\theta^{e_{\text{selected}}}(s, a^{e_{\text{selected}}}) - z_{\text{target}} \|_2^2$$

Where:

- $a^{e_{\text{selected}}} = A_\theta^{e_{\text{selected}}}(s)$: Action head output (**trainable**)
- $z_{\text{target}} = \text{stopgrad}(Z_\theta^{e_{\text{selected}}}(s, a^{e_{\text{selected}}}))$: Dynamics prediction (**frozen**)

Important: Dynamics head is frozen, only Action is trained

Gradient: Action head and Transformer

$$\nabla_{\theta_A, \theta_{\text{transformer}}} \mathcal{L}_A$$

2.3 Alternating Updates

The four losses are optimized **alternately**:

```
for each timestep t in episode:
  sequence_t ← [s_0, a_0, e_0, r_0, ..., s_t]
  e_selected ← emotions[t]

  # === Phase 1: Dynamics update ===
  outputs ← model.forward(sequence_t)

  a_e ← stopgrad(outputs.actions[e_selected])  # Action frozen
  h_t ← outputs.hidden

  z_pred ← model.dynamics_heads[e_selected](h_t, a_e)
  s_actual ← states[t+1]

  L_dynamics ← MSE(z_pred, s_actual)

  optimizer_dynamics.zero_grad()
  L_dynamics.backward()
  optimizer_dynamics.step()

  # === Phase 2: Action update ===
  outputs ← model.forward(sequence_t)

  a_e ← outputs.actions[e_selected]  # Action trainable
  h_t ← stopgrad(outputs.hidden)

  z_target ← stopgrad(outputs.dynamics[e_selected])  # Dynamics frozen
  z_pred ← model.dynamics_heads[e_selected](h_t, a_e)

  L_action ← MSE(z_pred, z_target)

  optimizer_action.zero_grad()
  L_action.backward()
  optimizer_action.step()

  # === Phase 3: Value update ===
  outputs ← model.forward(sequence_t)

  v_pred ← outputs.values[e_selected]
  G_actual ← returns[t]

  L_value ← MSE(v_pred, G_actual)

  optimizer_value.zero_grad()
  L_value.backward()
  optimizer_value.step()
```

```
# === Phase 4: Policy update ===
outputs ← model.forward(sequence_t)

π_pred ← outputs.policy
π_mcts ← mcts_distributions[t]

L_policy ← -SUM(π_mcts * LOG(π_pred + ε))

optimizer_policy.zero_grad()
L_policy.backward()
optimizer_policy.step()
```

2.4 Gradient Flow

Phase 1: Dynamics Learning

```
s_actual → MSE → Dynamics Head → Transformer → backward
                      ↑
              Action Head (frozen, no grad)
```

Phase 2: Action Learning

```
z_target (frozen) → MSE → Dynamics Head (frozen)
                            ↑
                    Action Head → Transformer → backward
```

Phase 3: Value Learning

```
G_actual → MSE → Value Head → Transformer → backward
```

Phase 4: Policy Learning

```
π_mcts → CrossEntropy → Policy Head → Transformer → backward
```

Important: Gradients flow to the Transformer in all Phases. This optimizes the history encoding h_t for each task.

2.5 Important Design Decisions

Why Dynamics Takes Action as Input

$$Z_\theta(s, a) = f_\theta(\text{concat}(h(s), a))$$

Reasons:

1. Action gradients flow through Dynamics (Phase 2)
2. Dynamics is conditioned on Action ($p(s_{t+1}|s_t, a_t)$)
3. Dynamics can be used as a "differentiable environment model" during Action learning

Incorrect Implementation:

```
python

# ❌ This doesn't propagate gradients
z_pred = dynamics_head(h_t)  # Doesn't take a
a = action_head(h_t)
loss = MSE(z_pred, target)  # No gradient to a
```

Correct Implementation:

```
python

# ✓ Gradients flow
a = action_head(h_t)
z_pred = dynamics_head(h_t, a)  # Takes a as input
loss = MSE(z_pred, target)  # Gradients flow to a
```

Why Alternating Updates

Reasons:

1. **Phase 1**: Dynamics learns reality → Accurate world model
2. **Phase 2**: Action utilizes Dynamics → Better action selection
3. Mutual improvement feedback loop
4. **Transformer learns from all four tasks** → Multi-task learning effect

Regular end-to-end learning can be unstable when Dynamics and Action are updated simultaneously.

About Four Transformer Updates:

At each timestep, the Transformer is updated four times (once per Phase). This has the following advantages:

1. Learns from multiple objective functions → More general representation
2. Doesn't overspecialize to each task
3. Learns h_t useful for Policy/Value/Dynamics/Action

However, there's a risk of overfitting without proper learning rate settings. Recommended:

- Head learning rate: 3×10^{-4}

- Transformer learning rate: 1×10^{-4} (lower than Heads)

Alternatively, you can share Transformer parameters across all Optimizers, effectively making it a single update (implementation-dependent).

---

3. Important Implementation Points

3.1 Sequence Management

```python
# ✓ Correct: Store only actual experience
real_sequence = [s_0, a_0, e_0, r_0, s_1, a_1, e_1, r_1, ...]

# ✗ Wrong: Store MCTS imagination
# real_sequence += imagined_transitions  # Don't do this
```

3.2 Optimizer Configuration

Important: Transformer is shared across all Optimizers

```python
# ✓ Correct: Optimize each Head with Transformer
optimizer_dynamics = Adam(
    list(dynamics_heads.parameters()) + list(transformer.parameters())
)
optimizer_action = Adam(
    list(action_heads.parameters()) + list(transformer.parameters())
)
optimizer_value = Adam(
    list(value_heads.parameters()) + list(transformer.parameters())
)
optimizer_policy = Adam(
    list(policy_head.parameters()) + list(transformer.parameters())
)

# ✗ Wrong 1: Transformer not trained
# optimizer_dynamics = Adam(dynamics_heads.parameters())  # No Transformer

# ✗ Wrong 2: Single Optimizer
# optimizer = Adam(model.parameters())  # Can't do Alternating Updates
```

Note: The Transformer is updated four times (once per Phase). This is intentional design, acquiring general representations by learning from multiple tasks.

3.3 Saving MCTS Visit Distribution

```python
# ✓ Correct: Save during episode collection
episode = {
    'states': [...],
    'actions': [...],
    'emotions': [...],
    'rewards': [...],
    'mcts_distributions': [...]  # This is essential
}

# ✗ Wrong: Don't save visit distribution
# → Can't train Policy
```

3.4 Temperature Scheduling for Emotion Selection

```python
# Early training: High temperature (exploration-focused)
temperature = 1.0

# Late training: Low temperature (exploitation-focused)
temperature = 0.1

# Sampling
if temperature == 0:
    e_selected = argmax(visit_dist)
else:
    probs = visit_dist ** (1 / temperature)
    probs = probs / sum(probs)
    e_selected = random.choice(probs)
```

---

4. Mathematical Details

4.1 Derivation of UCB Formula

The UCB formula combines two terms:

Exploitation term:

$$Q(n, e) = \frac{W(n, e)}{N(n, e)}$$

Exploration term:

$$U(n,e) = c_{\text{puct}} \cdot P(n,e) \cdot \frac{\sqrt{N(n)}}{1 + N(n,e)}$$

This maximizes the upper bound:

$$\text{UCB}(n,e) = Q(n,e) + U(n,e)$$

As $N(n,e) \to \infty, U(n,e) \to 0$, and exploitation dominates.

4.2 Justification for Policy Loss

The MCTS visit distribution π_{MCTS} approximates the optimal policy with sufficient simulations. The Policy network distills this distribution:

$$\min_{\theta} D_{\text{KL}}(\pi_{\text{MCTS}} \| \pi_\theta) = -\sum_{e} \pi_{\text{MCTS}}(e) \log \pi_\theta(e) + \text{const}$$

This enables the Policy network to "select good emotions without MCTS".

4.3 Theory of Dynamics-Action Separated Learning

Goal: Modeling $p(s_{t+1}|s_t, a_t)$

Dynamics learns the laws of the world:

$$Z_\theta(s_t, a_t) \approx s_{t+1}$$

Action is optimized using Dynamics as the environment:

$$\max_{a} V(Z_\theta(s_t, a))$$

This is a form of model-based RL; if Dynamics is accurate, Action also improves.

---

5. Overall Algorithm Flow

```
# === Training Loop ===
for episode in range(num_episodes):
    env.reset()
    sequence = [initial_state]
    episode_data = {states: [], actions: [], emotions: [], rewards: [], mcts_dists: []}
```

```
  while not done:
    # 1. Run MCTS
    visit_dist = MCTS_SEARCH(model, sequence, n_simulations=50)

    # 2. Select emotion
    e = SAMPLE(visit_dist, temperature)

    # 3. Get action
    outputs = model.forward(sequence)
    a_logits = outputs.actions[e]
    a = SAMPLE(softmax(a_logits))

    # 4. Execute in environment
    s_next, r, done = env.step(a)

    # 5. Store experience
    episode_data.states.append(current_state)
    episode_data.actions.append(a)
    episode_data.emotions.append(e)
    episode_data.rewards.append(r)
    episode_data.mcts_dists.append(visit_dist)

    # 6. Update sequence (only actual experience)
    sequence += [a, e, r, s_next]
    current_state = s_next

  # 7. Compute returns
  episode_data.returns = COMPUTE_RETURNS(episode_data.rewards, gamma=0.99)

  # 8. Alternating Updates
  for t in range(len(episode_data.states) - 1):
    TRAIN_STEP(model, episode_data, t, optimizers)

  # 9. Save to buffer
  buffer.add(episode_data)
```

---

6. Checklist

Items to verify during implementation:

- [ ] MCTS selects nodes using UCB formula
- [ ] MCTS returns visit distribution
- [ ] MCTS imagined sequences are not added to true sequence
- [ ] Dynamics Head takes (h_t, a) as input

- [ ] Phase 1: stopgrad Action
- [ ] Phase 2: stopgrad Dynamics, Action is trainable
- [ ] Phase 3: Train Value with actual returns
- [ ] Phase 4: Train Policy with MCTS visit distribution
- [ ] Four independent Optimizers
- [ ] Save MCTS visit distribution to episode data
- [ ] Implement temperature scheduling

This is the implementation-level detailed design specification.