Relevant Code

Code Repository

Repository: Pseudocode - No Repository Found

Status: Pseudocode (Based on Paper Description)

Language: Python (Pseudocode)

Last Updated: January 2025

Stars/Popularity: N/A

Architecture Overview

Based on the DeepSeek-R1 paper, the system architecture consists of several key components that work together to enable reasoning capabilities through reinforcement learning. The core innovation is the use of pure RL without supervised fine-tuning, allowing models to self-evolve reasoning behaviors. The architecture includes the GRPO (Group Relative Policy Optimization) algorithm for cost-effective RL training, a dual-component reward system, and a multi-stage training pipeline that combines RL with supervised fine-tuning for enhanced performance.

The conceptual architecture can be organized into these main components:

Directory Structure

```
deepseek_r1/
├── core/
│   ├── grpo.py              # GRPO RL algorithm implementation
│   ├── reward_model.py      # Rule-based reward system
│   └── policy_model.py      # Base language model wrapper
├── training/
│   ├── trainer_zero.py      # DeepSeek-R1-Zero training pipeline
│   ├── trainer_r1.py        # DeepSeek-R1 multi-stage training
│   └── data_collector.py    # Cold-start and rejection sampling
├── inference/
│   ├── model_wrapper.py     # Model inference interface
│   └── reasoning_engine.py  # Chain-of-thought generation
└── distillation/
    ├── distiller.py         # Model distillation pipeline
    └── small_model_configs/ # Configs for 1.5B-70B models
```

Key Implementation

GRPO (Group Relative Policy Optimization) Algorithm

This is the core RL algorithm that eliminates the need for expensive critic models by using group-based advantage estimation.

```
class GRPOTrainer:
    def __init__(self, policy_model, ref_model, group_size=8, epsilon=0.2, beta=0.1):
        self.policy_model = policy_model
        self.ref_model = ref_model
        self.group_size = group_size
        self.epsilon = epsilon
        self.beta = beta

    def compute_group_advantage(self, rewards):
        """Compute advantage scores using group-based normalization"""
        group_mean = np.mean(rewards)
        group_std = np.std(rewards)
        advantages = (rewards - group_mean) / (group_std + 1e-8)
        return advantages

    def grpo_objective(self, old_log_probs, new_log_probs, advantages, ref_log_probs):
        """Compute GRPO objective function"""
        # Policy ratio
        ratio = torch.exp(new_log_probs - old_log_probs)

        # Clipped surrogate objective
        clipped_ratio = torch.clamp(ratio, 1 - self.epsilon, 1 + self.epsilon)
        surrogate = torch.min(ratio * advantages, clipped_ratio * advantages)

        # KL divergence penalty
        kl_penalty = self.beta * (ref_log_probs - new_log_probs)

        return surrogate - kl_penalty
```

```
def train_step(self, questions):
    """Single training step with group sampling"""
    batch_outputs = []
    batch_rewards = []
    batch_old_log_probs = []

    for question in questions:
        # Sample group of outputs from old policy
        group_outputs = []
        group_old_log_probs = []

        for _ in range(self.group_size):
            output, log_prob = self.policy_model.generate_with_logprob(question)
            group_outputs.append(output)
            group_old_log_probs.append(log_prob)

            # Compute reward for this output
            reward = self.compute_reward(question, output)
            batch_rewards.append(reward)

        batch_outputs.extend(group_outputs)
        batch_old_log_probs.extend(group_old_log_probs)

    # Compute advantages
    advantages = self.compute_group_advantage(batch_rewards)

    # Update policy
    for i, (question, output, old_log_prob, advantage) in enumerate(
        zip(questions * self.group_size, batch_outputs, batch_old_log_probs, advantages)):

        # Forward pass with current policy
        new_log_prob = self.policy_model.get_logprob(question, output)
        ref_log_prob = self.ref_model.get_logprob(question, output)

        # Compute loss
        loss = -self.grpo_objective(old_log_prob, new_log_prob, advantage, ref_log_prob)

        # Backward pass
        loss.backward()
```

```
            self.optimizer.step()
            self.optimizer.zero_grad()
```

Key aspects:
- Eliminates critic model by using group-based advantage estimation
- Reduces training costs by approximately 50% compared to traditional PPO
- Maintains stability through KL divergence regularization
- Uses clipping to prevent large policy updates

Rule-Based Reward System

The dual-component reward system provides learning signals without requiring neural reward models.

```
class RuleBasedRewardSystem:
    def __init__(self):
        self.accuracy_weight = 1.0
        self.format_weight = 0.1

    def compute_reward(self, question, response):
        """Compute total reward for a response"""
        accuracy_reward = self.compute_accuracy_reward(question, response)
        format_reward = self.compute_format_reward(response)

        total_reward = (self.accuracy_weight * accuracy_reward +
                       self.format_weight * format_reward)
        return total_reward

    def compute_accuracy_reward(self, question, response):
        """Rule-based accuracy evaluation"""
        if question.type == "math":
            return self.evaluate_math_answer(question, response)
        elif question.type == "coding":
            return self.evaluate_code_solution(question, response)
        elif question.type == "logic":
            return self.evaluate_logic_answer(question, response)
        else:
            return 0.0

    def evaluate_math_answer(self, question, response):
        """Evaluate mathematical answer correctness"""
        try:
            # Extract final answer from boxed format
            answer_match = re.search(r'\\boxed\{([^}]+)\}', response)
            if not answer_match:
                return 0.0

            predicted_answer = answer_match.group(1)
            correct_answer = question.answer

            # Numerical comparison
            if self.is_numerical(predicted_answer) and self.is_numerical(correct_answer):
                pred_val = float(predicted_answer)
                true_val = float(correct_answer)
                return 1.0 if abs(pred_val - true_val) < 1e-6 else 0.0

            # Symbolic comparison
            return 1.0 if predicted_answer.strip() == correct_answer.strip() else 0.0

        except Exception:
            return 0.0

    def evaluate_code_solution(self, question, response):
        """Evaluate code solution using test cases"""
        try:
            # Extract code from response
            code_match = re.search(r'```python\n(.*?)\n```', response, re.DOTALL)
            if not code_match:
                return 0.0

            code = code_match.group(1)

            # Execute with test cases
            test_results = []
            for test_case in question.test_cases:
                result = self.execute_code(code, test_case)
                test_results.append(result)

            # Return success rate
            return sum(test_results) / len(test_results)

        except Exception:
```

```
            return 0.0

    def compute_format_reward(self, response):
        """Reward for proper thinking format"""
        format_score = 0.0

        # Check for thinking tags
        if '<think>' in response and '</think>' in response:
            format_score += 0.5

        # Check for reasoning process length
        thinking_content = self.extract_thinking_content(response)
        if len(thinking_content.split()) > 50:  # Minimum reasoning length
            format_score += 0.3

        # Check for summary section
        if '<summary>' in response and '</summary>' in response:
            format_score += 0.2

        return format_score
```

Key aspects:
- Avoids reward hacking by using deterministic rule-based evaluation
- Separates accuracy and format rewards for stable training
- Supports multiple question types (math, coding, logic)
- Provides immediate feedback without neural network evaluation

Multi-Stage Training Pipeline (DeepSeek-R1)

The enhanced training pipeline combines cold-start data with iterative RL and SFT stages.

```
class DeepSeekR1Trainer:
    def __init__(self, base_model):
        self.base_model = base_model
        self.current_model = base_model
        self.reward_system = RuleBasedRewardSystem()

    def train_deepseek_r1(self):
        """Complete DeepSeek-R1 training pipeline"""

        # Stage 1: Cold Start Fine-tuning
        print("Stage 1: Cold start fine-tuning")
        cold_start_data = self.collect_cold_start_data()
        self.current_model = self.supervised_fine_tune(
            self.current_model, cold_start_data
        )

        # Stage 2: Reasoning-oriented RL
        print("Stage 2: Reasoning-oriented RL")
        reasoning_prompts = self.load_reasoning_prompts()
        grpo_trainer = GRPOTrainer(self.current_model, self.base_model)

        for iteration in range(self.reasoning_rl_iterations):
            grpo_trainer.train_step(reasoning_prompts)

            # Add language consistency reward after warmup
            if iteration > self.warmup_iterations:
                self.add_language_consistency_reward(grpo_trainer)

        # Stage 3: Rejection Sampling and SFT
        print("Stage 3: Rejection sampling and SFT")
        rl_checkpoint = self.current_model

        # Collect reasoning data via rejection sampling
        reasoning_data = self.collect_reasoning_data(rl_checkpoint)

        # Collect non-reasoning data
        non_reasoning_data = self.collect_non_reasoning_data()

        # Combine datasets
        combined_data = reasoning_data + non_reasoning_data

        # SFT on combined data
        self.current_model = self.supervised_fine_tune(
            self.base_model, combined_data
        )

        # Stage 4: RL for all scenarios
        print("Stage 4: RL for all scenarios")
        all_prompts = self.load_all_scenario_prompts()
        final_grpo_trainer = GRPOTrainer(self.current_model, self.base_model)
        for iteration in range(self.final_rl_iterations):
            final_grpo_trainer.train_step(all_prompts)

        return self.current_model

    def collect_cold_start_data(self):
        """Collect thousands of long CoT examples"""
        cold_start_data = []

        # Method 1: Few-shot prompting with long CoT examples
        few_shot_examples = self.generate_few_shot_examples()

        # Method 2: Direct prompting for detailed answers
        detailed_prompts = self.create_detailed_prompts()

        # Method 3: Process R1-Zero outputs for readability
        r1_zero_outputs = self.load_r1_zero_outputs()
        readable_outputs = self.post_process_for_readability(r1_zero_outputs)

        # Method 4: Human annotation and refinement
        human_refined = self.get_human_annotations()

        cold_start_data.extend(few_shot_examples)
        cold_start_data.extend(detailed_prompts)
        cold_start_data.extend(readable_outputs)
        cold_start_data.extend(human_refined)

        return cold_start_data[:self.cold_start_limit]  # Thousands of examples

    def collect_reasoning_data(self, rl_checkpoint):
        """Collect reasoning data via rejection sampling"""
        reasoning_data = []
        reasoning_prompts = self.load_reasoning_prompts()

        for prompt in reasoning_prompts:
            # Sample multiple responses
            responses = []
            for _ in range(self.rejection_samples):
                response = rl_checkpoint.generate(prompt)
                is_correct = self.reward_system.compute_reward(prompt, response) > 0.5
                if is_correct:
                    responses.append(response)

            # Keep correct responses
            if responses:
                # Filter for readability
                readable_responses = self.filter_readable_responses(responses)
                reasoning_data.extend(readable_responses)

        return reasoning_data

    def filter_readable_responses(self, responses):
        """Filter responses for readability and language consistency"""
        readable = []

        for response in responses:
            # Check for language mixing
            if self.has_language_mixing(response):
                continue

            # Check for proper formatting
            if not self.has_proper_formatting(response):
                continue

            # Check for reasonable length
            if len(response) > self.max_response_length:
                continue

            readable.append(response)

        return readable
```

Key aspects:
- Four-stage pipeline combining RL and SFT
- Cold-start data improves readability and convergence
- Rejection sampling collects high-quality reasoning trajectories
- Language consistency rewards improve user experience

Model Distillation Framework

Transfers reasoning capabilities from large models to smaller dense models.

```python
class ReasoningDistiller:
    def __init__(self, teacher_model, student_model):
        self.teacher_model = teacher_model
        self.student_model = student_model

    def distill_reasoning_capability(self, training_prompts):
        """Distill reasoning patterns from teacher to student"""

        # Generate reasoning data from teacher
        reasoning_data = []

        for prompt in training_prompts:
            # Generate teacher response with chain-of-thought
            teacher_response = self.teacher_model.generate(
                prompt,
                max_length=32768,
                temperature=0.6
            )

            # Extract reasoning process and summary
            reasoning_process = self.extract_reasoning_process(teacher_response)
            summary = self.extract_summary(teacher_response)

            # Create training example
            training_example = {
                'prompt': prompt,
                'reasoning_process': reasoning_process,
                'summary': summary
            }

            reasoning_data.append(training_example)

        # Fine-tune student model on generated data
        self.student_model = self.supervised_fine_tune(
            self.student_model,
            reasoning_data
        )

        return self.student_model
    def extract_reasoning_process(self, response):
        """Extract chain-of-thought reasoning from response"""
        # Look for thinking tags
        if '<think>' in response and '</think>' in response:
            start = response.find('<think>') + len('<think>')
            end = response.find('</think>')
            return response[start:end].strip()

        # Fallback: look for reasoning patterns
        reasoning_patterns = [
            "Let me think step by step",
            "First, I'll",
            "Next, I need to",
            "Let me consider",
            "I should"
        ]

        for pattern in reasoning_patterns:
            if pattern in response:
                # Extract section containing reasoning
                start = response.find(pattern)
                return response[start:].strip()

        return response

    def extract_summary(self, response):
        """Extract final summary from response"""
        if '<summary>' in response and '</summary>' in response:
            start = response.find('<summary>') + len('<summary>')
            end = response.find('</summary>')
            return response[start:end].strip()

        # Fallback: look for conclusion indicators
        conclusion_indicators = [
            "In conclusion",
            "Therefore",
            "The answer is",
            "Final answer",
            "So the solution is"
        ]
        for indicator in conclusion_indicators:
            if indicator in response:
                start = response.find(indicator)
                return response[start:].strip()

        return response[-200:]  # Last 200 characters as fallback
```

Key aspects:
- Direct distillation outperforms RL on small models
- Preserves reasoning patterns discovered by larger models
- Enables efficient deployment of reasoning capabilities
- Works across different model architectures (Qwen, Llama)

Relation to Paper

Paper Section	Code Component	Notes
Section 2.2.1: GRPO Algorithm	`core/grpo.py:GRPOTrainer`	Implements group-based advantage estimation without critic model
Equation 1-3	`core/grpo.py:grpo_objective()`	Mathematical implementation of GRPO objective function
Section 2.2.2: Reward Modeling	`core/reward_model.py:RuleBasedRewardSystem`	Rule-based accuracy and format rewards, avoiding neural reward models
Section 2.3: Multi-stage Training	`training/trainer_r1.py:DeepSeekR1Trainer`	Four-stage pipeline with cold-start, RL, SFT, and final RL
Section 2.4: Distillation	`distillation/distiller.py:ReasoningDistiller`	Transfers reasoning patterns to smaller dense models
Figure 2: Performance Evolution	`training/trainer_zero.py`	Tracks AIME performance improvement during RL training

Paper Section	Code Component	Notes
Table 3: Aha Moment	`inference/reasoning_engine.py`	Implements reflection and re-evaluation behaviors

Key Differences from Paper

- **Simplified Implementation**: Pseudocode focuses on core algorithmic concepts rather than engineering optimizations
- **Abstraction Level**: Paper implementation likely includes more sophisticated parallelization and distributed training
- **Reward System**: Paper may include additional edge cases and safety considerations not shown in pseudocode
- **Model Architecture**: Specific model architecture details (MoE structure, attention mechanisms) are abstracted away
- **Training Infrastructure**: Actual implementation includes sophisticated data pipelines and monitoring systems

Usage Examples

Basic Usage for Reasoning Tasks

```
# Load trained DeepSeek-R1 model
model = load_deepseek_r1("deepseek-r1-checkpoint")

# Generate reasoning response
prompt = "Solve: sqrt(a - sqrt(a + x)) = x, where sum of real solutions equals a > 1"

response = model.generate(
    prompt,
    max_length=32768,
    temperature=0.6,
    top_p=0.95
)

print(response)
```

```
# Output will include detailed reasoning process in <think> tags
# and final summary in <summary> tags
```

Distillation for Small Models

```
# Create distilled 7B model
teacher = load_deepseek_r1("deepseek-r1-checkpoint")
student = load_qwen2_5("Qwen2.5-Math-7B")

distiller = ReasoningDistiller(teacher, student)
distilled_model = distiller.distill_reasoning_capability(math_prompts)

# Use distilled model for efficient reasoning
response = distilled_model.generate(math_problem, max_length=8192)
```

Batch Evaluation

```
# Evaluate on benchmark
evaluator = ReasoningEvaluator(model)
results = evaluator.evaluate_on_aime2024()

print(f"AIME 2024 Pass@1: {results['pass@1']:.1f}%")
print(f"AIME 2024 Cons@64: {results['cons@64']:.1f}%")
```