Relevant Code

Code Repository

Repository: Pseudocode - No Repository Found

Status: Pseudocode Implementation (Based on Paper Algorithm)

Language: Python (Conceptual Implementation)

Last Updated: March 17, 2025 (Paper Date)

Stars/Popularity: N/A

Architecture Overview

The DAPO algorithm consists of four core technical components that work together to enable large-scale reinforcement learning for language models. Since the official repository is not publicly available, this implementation represents a conceptual architecture based on the paper's detailed algorithm description. The system is designed to work with the verl framework but can be adapted to other RL training frameworks.

The architecture consists of: (1) a dynamic sampling module that filters training prompts based on accuracy, (2) an asymmetric clipping policy optimizer that allows different exploration and exploitation bounds, (3) a token-level loss computation system that rebalances gradient contributions, and (4) a reward shaping module that handles overlong responses intelligently.

Directory Structure

```
dapo_conceptual/
├── algorithms/
│   ├── dapo_core.py         # Core DAPO algorithm implementation
│   ├── clipping.py          # Asymmetric clipping strategy
│   ├── dynamic_sampling.py  # Dynamic sampling and filtering
│   └── reward_shaping.py    # Overlong reward shaping
├── models/
│   ├── policy_model.py      # Policy model wrapper
│   └── reward_model.py      # Rule-based reward computation
```

```
├── training/
│   ├── trainer.py           # Main training loop
│   ├── rollout.py           # Response generation and evaluation
│   └── optimizer.py         # Custom optimizer setup
├── data/
│   ├── math_dataset.py      # DAPO-Math-17K dataset handling
│   └── preprocessing.py     # Data transformation utilities
└── utils/
    ├── monitoring.py        # Training metrics and logging
    └── evaluation.py        # AIME evaluation utilities
```

Key Implementation

Core DAPO Algorithm

Based on Algorithm 1 from the paper, this implements the main training loop with all four key techniques integrated:

```
import torch
import torch.nn.functional as F
from typing import List, Dict, Tuple
import numpy as np

class DAPOTrainer:
    def __init__(self,
                 policy_model,
                 reward_model,
                 epsilon_low: float = 0.2,
                 epsilon_high: float = 0.28,
                 max_length: int = 16384,
                 cache_length: int = 4096,
                 group_size: int = 16):

        self.policy_model = policy_model
        self.reward_model = reward_model
        self.epsilon_low = epsilon_low
        self.epsilon_high = epsilon_high
        self.max_length = max_length
        self.cache_length = cache_length
        self.group_size = group_size
```

```
        self.old_policy = None

    def compute_advantages(self, rewards: List[float]) -> List[float]:
        """Compute group-relative advantages using normalization"""
        rewards = np.array(rewards)
        mean_reward = np.mean(rewards)
        std_reward = np.std(rewards)

        # Avoid division by zero
        if std_reward < 1e-8:
            return np.zeros_like(rewards).tolist()

        advantages = (rewards - mean_reward) / std_reward
        return advantages.tolist()

    def clip_importance_ratio(self,
                              ratio: torch.Tensor,
                              advantages: torch.Tensor) -> torch.Tensor:
        """Apply asymmetric clipping to importance sampling ratios"""
        # Separate positive and negative advantages
        pos_mask = advantages > 0
        neg_mask = ~pos_mask

        clipped_ratio = ratio.clone()

        # For positive advantages (increase probability), use epsilon_high
        clipped_ratio[pos_mask] = torch.clamp(
            ratio[pos_mask],
            1 - self.epsilon_low,
            1 + self.epsilon_high
        )

        # For negative advantages (decrease probability), use epsilon_low
        clipped_ratio[neg_mask] = torch.clamp(
            ratio[neg_mask],
            1 - self.epsilon_low,
            1 + self.epsilon_low
        )

        return clipped_ratio
```

```
    def compute_length_penalty(self, response_length: int) -> float:
        """Compute soft overlength punishment reward shaping"""
        if response_length <= self.max_length - self.cache_length:
            return 0.0
        elif response_length <= self.max_length:
            # Gradual punishment in cache interval
            return (self.max_length - self.cache_length - response_length) / self.cache_length
        else:
            return -1.0  # Severe punishment for excessive length

    def filter_prompts_by_accuracy(self,
                                   prompts: List[str],
                                   responses_list: List[List[str]],
                                   rewards_list: List[List[float]]) -> Tuple[List[str], List[List[st
        """Dynamic sampling: filter out prompts with accuracy=1 or accuracy=0"""
        filtered_prompts = []
        filtered_responses = []
        filtered_rewards = []

        for prompt, responses, rewards in zip(prompts, responses_list, rewards_list):
            # Check if all responses are correct or all incorrect
            all_correct = all(r > 0 for r in rewards)
            all_incorrect = all(r <= 0 for r in rewards)

            # Keep only prompts with mixed results
            if not (all_correct or all_incorrect):
                filtered_prompts.append(prompt)
                filtered_responses.append(responses)
                filtered_rewards.append(rewards)

        return filtered_prompts, filtered_responses, filtered_rewards

    def compute_dapo_loss(self,
                          old_log_probs: torch.Tensor,
                          new_log_probs: torch.Tensor,
                          advantages: torch.Tensor,
                          response_lengths: List[int]) -> torch.Tensor:
        """Compute token-level DAPO loss with all four techniques"""

        # Compute importance sampling ratio
        log_ratio = new_log_probs - old_log_probs
```

```
        ratio = torch.exp(log_ratio)

        # Apply asymmetric clipping
        clipped_ratio = self.clip_importance_ratio(ratio, advantages)

        # Compute surrogate losses
        surrogate1 = ratio * advantages
        surrogate2 = clipped_ratio * advantages

        # Take minimum (PPO-style)
        surrogate_loss = torch.min(surrogate1, surrogate2)

        # Apply length-aware reward shaping
        length_penalties = torch.tensor([
            self.compute_length_penalty(length) for length in response_lengths
        ], device=surrogate_loss.device)

        # Expand length penalties to match token-level losses
        expanded_penalties = []
        for i, (penalty, length) in enumerate(zip(length_penalties, response_lengths)):
            expanded_penalties.extend([penalty] * length)

        length_penalty_tensor = torch.tensor(expanded_penalties, device=surrogate_loss.device)

        # Combine losses
        total_loss = -(surrogate_loss + length_penalty_tensor).mean()

        return total_loss

    def train_step(self, prompts: List[str], max_iterations: int = 16) -> Dict[str, float]:
        """Main DAPO training step implementing Algorithm 1"""

        # Step 3: Update old policy
        if self.old_policy is None:
            self.old_policy = self.policy_model.copy()
        else:
            self.old_policy.load_state_dict(self.policy_model.state_dict())

        # Step 4: Generate responses
        responses_list = []
        rewards_list = []
```

```
        log_probs_list = []

        for prompt in prompts:
            prompt_responses = []
            prompt_rewards = []
            prompt_log_probs = []

            for _ in range(self.group_size):
                # Generate response
                response, log_probs = self.generate_response(prompt)
                prompt_responses.append(response)
                prompt_log_probs.append(log_probs)

                # Compute reward (rule-based + length shaping)
                base_reward = self.reward_model.evaluate_response(prompt, response)
                length_penalty = self.compute_length_penalty(len(response.split()))
                total_reward = base_reward + length_penalty
                prompt_rewards.append(total_reward)

            responses_list.append(prompt_responses)
            rewards_list.append(prompt_rewards)
            log_probs_list.append(prompt_log_probs)

        # Step 6: Dynamic sampling filtering
        filtered_prompts, filtered_responses, filtered_rewards = self.filter_prompts_by_accuracy(
            prompts, responses_list, rewards_list
        )

        # Continue if buffer is not full enough
        if len(filtered_prompts) < len(prompts) * 0.5:  # Threshold for minimum batch size
            return {"loss": 0.0, "entropy": 0.0}  # Skip training step

        # Step 9: Compute advantages for all tokens
        all_advantages = []
        all_old_log_probs = []
        all_new_log_probs = []
        all_response_lengths = []

        for prompt, responses, rewards, old_log_probs_batch in zip(
            filtered_prompts, filtered_responses, filtered_rewards, log_probs_list
        ):
```

```
            # Compute advantages for this group
            advantages = self.compute_advantages(rewards)

            for i, (response, advantage, old_log_probs) in enumerate(
                zip(responses, advantages, old_log_probs_batch)
            ):
                # Expand advantage to all tokens in the response
                token_advantages = [advantage] * len(old_log_probs)
                all_advantages.extend(token_advantages)
                all_old_log_probs.extend(old_log_probs)
                all_response_lengths.append(len(old_log_probs))

        # Convert to tensors
        advantages_tensor = torch.tensor(all_advantages)
        old_log_probs_tensor = torch.tensor(all_old_log_probs)

        # Step 10-11: Multiple gradient updates
        total_loss = 0.0
        for iteration in range(max_iterations):
            # Forward pass with current policy
            new_log_probs_tensor = self.get_new_log_probs(filtered_prompts, filtered_responses)

            # Compute DAPO loss
            loss = self.compute_dapo_loss(
                old_log_probs_tensor,
                new_log_probs_tensor,
                advantages_tensor,
                all_response_lengths
            )

            # Backward pass
            loss.backward()
            total_loss += loss.item()

            # Update policy
            self.optimizer.step()
            self.optimizer.zero_grad()

        # Compute metrics
        avg_loss = total_loss / max_iterations
        entropy = self.compute_entropy()
```

```
        return {
            "loss": avg_loss,
            "entropy": entropy,
            "num_filtered_prompts": len(filtered_prompts),
            "total_prompts": len(prompts)
        }
```

Dynamic Sampling Module

Implements the intelligent filtering mechanism that ensures consistent gradient signals:

```
class DynamicSampler:
    def __init__(self, min_batch_ratio: float = 0.5, max_attempts: int = 10):
        self.min_batch_ratio = min_batch_ratio
        self.max_attempts = max_attempts

    def sample_until_full_batch(self,
                                dataset: List[str],
                                batch_size: int,
                                policy_model,
                                reward_model,
                                group_size: int = 16) -> Tuple[List[str], List[List[str]], List[List
        """
        Continuously sample until batch is filled with prompts that have mixed accuracy.
        Implements the dynamic sampling strategy from Equation (11).
        """

        final_prompts = []
        final_responses = []
        final_rewards = []

        attempts = 0

        while len(final_prompts) < batch_size and attempts < self.max_attempts:
            # Sample a candidate batch
            remaining_needed = batch_size - len(final_prompts)
            candidate_prompts = self.sample_candidates(dataset, remaining_needed)

            # Generate responses and compute rewards
```

```
            candidate_results = []
            for prompt in candidate_prompts:
                responses, rewards = self.evaluate_prompt(prompt, policy_model, reward_model, group
                candidate_results.append((prompt, responses, rewards))

            # Filter for mixed accuracy
            for prompt, responses, rewards in candidate_results:
                if len(final_prompts) >= batch_size:
                    break

                # Check accuracy distribution
                correct_count = sum(1 for r in rewards if r > 0)
                accuracy = correct_count / len(rewards)

                # Keep only if 0 < accuracy < 1 (mixed results)
                if 0 < accuracy < 1:
                    final_prompts.append(prompt)
                    final_responses.append(responses)
                    final_rewards.append(rewards)

            attempts += 1

        # If we still don't have enough, pad with remaining candidates
        if len(final_prompts) < batch_size * self.min_batch_ratio:
            # Add some filtered prompts to meet minimum requirement
            for prompt, responses, rewards in candidate_results:
                if len(final_prompts) >= batch_size * self.min_batch_ratio:
                    break
                final_prompts.append(prompt)
                final_responses.append(responses)
                final_rewards.append(rewards)

        return final_prompts, final_responses, final_rewards

    def sample_candidates(self, dataset: List[str], num_samples: int) -> List[str]:
        """Sample candidate prompts from dataset"""
        return np.random.choice(dataset, size=num_samples, replace=False).tolist()

    def evaluate_prompt(self, prompt: str, policy_model, reward_model, group_size: int) -> Tuple[Li
        """Generate responses and compute rewards for a single prompt"""
        responses = []
```

```
        rewards = []

        for _ in range(group_size):
            response = policy_model.generate(prompt)
            responses.append(response)

            # Rule-based reward
            reward = reward_model.evaluate_response(prompt, response)
            rewards.append(reward)

        return responses, rewards
```

Asymmetric Clipping Strategy

Implements the Clip-Higher technique that prevents entropy collapse:

```
class AsymmetricClipping:
    def __init__(self, epsilon_low: float = 0.2, epsilon_high: float = 0.28):
        self.epsilon_low = epsilon_low
        self.epsilon_high = epsilon_high

    def apply_clip_higher(self,
                          old_log_probs: torch.Tensor,
                          new_log_probs: torch.Tensor,
                          advantages: torch.Tensor) -> torch.Tensor:
        """
        Apply asymmetric clipping with different bounds for positive and negative advantages.
        Implements the Clip-Higher strategy from Section 3.1.
        """

        # Compute importance sampling ratio
        log_ratio = new_log_probs - old_log_probs
        ratio = torch.exp(log_ratio)

        # Create clipped version with asymmetric bounds
        clipped_ratio = ratio.clone()

        # Apply different clipping based on advantage sign
        positive_adv_mask = advantages > 0
        negative_adv_mask = ~positive_adv_mask
```

```
        # For positive advantages (want to increase probability): use epsilon_high
        clipped_ratio[positive_adv_mask] = torch.clamp(
            ratio[positive_adv_mask],
            1 - self.epsilon_low,
            1 + self.epsilon_high
        )

        # For negative advantages (want to decrease probability): use epsilon_low
        clipped_ratio[negative_adv_mask] = torch.clamp(
            ratio[negative_adv_mask],
            1 - self.epsilon_low,
            1 + self.epsilon_low
        )

        return clipped_ratio

    def compute_surrogate_loss(self,
                               ratio: torch.Tensor,
                               clipped_ratio: torch.Tensor,
                               advantages: torch.Tensor) -> torch.Tensor:
        """Compute PPO-style surrogate loss with asymmetric clipping"""

        # Surrogate objectives
        surrogate1 = ratio * advantages
        surrogate2 = clipped_ratio * advantages

        # Take minimum for stability
        surrogate_loss = torch.min(surrogate1, surrogate2)

        return -surrogate_loss.mean()  # Negative for maximization
```

Overlong Reward Shaping

Implements the intelligent length penalty system:

```
class OverlongRewardShaping:
    def __init__(self, max_length: int = 16384, cache_length: int = 4096):
        self.max_length = max_length
        self.cache_length = cache_length
```

```
        self.threshold = max_length - cache_length

    def compute_length_penalty(self, response_length: int) -> float:
        """
        Compute length-aware penalty as described in Equation (13).
        Implements soft overlong punishment.
        """
        if response_length <= self.threshold:
            # No penalty for acceptable length
            return 0.0
        elif response_length <= self.max_length:
            # Gradual punishment in cache interval
            return (self.threshold - response_length) / self.cache_length
        else:
            # Severe punishment for excessive length
            return -1.0

    def shape_rewards(self,
                      base_rewards: List[float],
                      response_lengths: List[int]) -> List[float]:
        """Apply length shaping to base rewards"""

        shaped_rewards = []
        for reward, length in zip(base_rewards, response_lengths):
            length_penalty = self.compute_length_penalty(length)
            shaped_reward = reward + length_penalty
            shaped_rewards.append(shaped_reward)

        return shaped_rewards

    def filter_overlong_responses(self,
                                  responses: List[str],
                                  rewards: List[float]) -> Tuple[List[str], List[float]]:
        """
        Optionally filter out severely overlong responses.
        Implements the Overlong Filtering strategy mentioned in Section 3.4.
        """

        filtered_responses = []
        filtered_rewards = []
```

```
        for response, reward in zip(responses, rewards):
            length = len(response.split())

            # Filter only severely overlong responses
            if length <= self.max_length * 1.5:  # Allow some flexibility
                filtered_responses.append(response)
                filtered_rewards.append(reward)

        return filtered_responses, filtered_rewards
```

Relation to Paper

Paper Section	Code Component	Notes
Algorithm 1	`DAPOTrainer.train_step()`	Complete implementation of main training loop
Section 3.1: Clip-Higher	`AsymmetricClipping`	Asymmetric clipping with ε_low=0.2, ε_high=0.28
Section 3.2: Dynamic Sampling	`DynamicSampler`	Filtering prompts with accuracy=1 or accuracy=0
Section 3.3: Token-Level Loss	`DAPOTrainer.compute_dapo_loss()`	Token-level loss computation instead of sample-level
Section 3.4: Overlong Reward Shaping	`OverlongRewardShaping`	Soft punishment mechanism with cache interval
Equation (8)	`DAPOTrainer.compute_dapo_loss()`	Main DAPO objective function implementation

Paper Section	Code Component	Notes
Equation (11)	`DynamicSampler.sample_until_full_batch()`	Dynamic sampling constraint implementation
Equation (13)	`OverlongRewardShaping.compute_length_penalty()`	Length-aware reward shaping

Key Differences from Paper

- **Framework Integration**: The implementation is designed to work with PyTorch and common RL frameworks, while the paper used the verl framework specifically
- **Modular Design**: The code is organized into separate modules for each technique, making it easier to understand and modify individual components
- **Monitoring Extensions**: Additional logging and monitoring capabilities are included for training stability, which were not detailed in the paper
- **Dataset Handling**: The implementation includes placeholder methods for the DAPO-Math-17K dataset processing that would need to be adapted based on actual data availability

This conceptual implementation provides a complete foundation for understanding and implementing the DAPO algorithm, capturing all four key technical innovations described in the paper while maintaining clarity and extensibility for practical applications.