Relevant Code

Code Repository

Repository: Not Available (Pseudocode Implementation)

Status: Pseudocode (No Repository Found)

Language: Python (Conceptual Implementation)

Last Updated: November 2025

Stars/Popularity: N/A

Architecture Overview

Based on the GSPO paper, the algorithm consists of several key components that work together to provide stable reinforcement learning training for large language models. The architecture follows a sequence-level approach rather than token-level optimization, which aligns the training objective with how rewards are actually computed and assigned.

The conceptual architecture includes:
1. **Policy Model**: The language model being trained, which generates responses to queries
2. **Reward Model**: Evaluates generated responses and assigns rewards
3. **GSPO Optimizer**: Implements sequence-level importance sampling and clipping
4. **Advantage Calculator**: Computes group-based advantages for response batches
5. **Sequence Processor**: Handles sequence-level computations and length normalization

Directory Structure

```
gspo_implementation/
├── core/
│   ├── gspo_optimizer.py     - Main GSPO algorithm implementation
│   ├── sequence_processor.py - Sequence-level computations
│   └── advantage_calculator.py - Group-based advantage estimation
├── models/
│   ├── policy_model.py       - Language model wrapper
│   └── reward_model.py       - Reward evaluation model
```

```
├── utils/
│   ├── clipping.py           - Sequence-level clipping mechanisms
│   └── normalization.py      - Length normalization utilities
└── training/
    ├── trainer.py            - Main training loop
    └── data_loader.py        - Query-response data handling
```

Key Implementation

Core GSPO Algorithm

This implements the main GSPO objective function from Equation (5) in the paper, providing sequence-level policy optimization with group-based advantage estimation.

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

class GSPOOptimizer:
    def __init__(self, policy_model, clip_range: Tuple[float, float] = (3e-4, 4e-4)):
        self.policy_model = policy_model
        self.clip_range = clip_range
        self.device = next(policy_model.parameters()).device

    def compute_sequence_likelihood(self, query: torch.Tensor, response: torch.Tensor) -> torch.Ten
        """
        Compute sequence likelihood π_θ(y|x) for a given query-response pair.
        Implements the sequence probability calculation.
        """
        with torch.no_grad():
            # Get logits for each token in the response
            logits = self.policy_model(query, response[:, :-1])
            log_probs = F.log_softmax(logits, dim=-1)

            # Compute log likelihood for each token
            token_log_probs = torch.gather(log_probs, -1, response[:, 1:].unsqueeze(-1)).squeeze(-1

            # Sum log probabilities to get sequence log likelihood
            sequence_log_likelihood = torch.sum(token_log_probs, dim=-1)
```

```
        return torch.exp(sequence_log_likelihood)

    def compute_importance_ratio(self,
                                 current_likelihood: torch.Tensor,
                                 old_likelihood: torch.Tensor,
                                 response_length: torch.Tensor) -> torch.Tensor:
        """
        Compute sequence-level importance ratio s_i(θ) from Equation (7):
        s_i(θ) = (π_θ(y_i|x) / π_θ_old(y_i|x))^(1/|y_i|)
        """
        ratio = current_likelihood / old_likelihood
        length_normalized_ratio = torch.pow(ratio, 1.0 / response_length.float())
        return length_normalized_ratio

    def compute_group_advantages(self, rewards: List[float]) -> torch.Tensor:
        """
        Compute group-based advantages using Equation (6):
        Â_i = (r(x,y_i) - mean({r(x,y_i)})) / std({r(x,y_i)})
        """
        rewards_tensor = torch.tensor(rewards, dtype=torch.float32, device=self.device)
        mean_reward = torch.mean(rewards_tensor)
        std_reward = torch.std(rewards_tensor)

        # Avoid division by zero
        if std_reward < 1e-8:
            return torch.zeros_like(rewards_tensor)

        advantages = (rewards_tensor - mean_reward) / std_reward
        return advantages

    def apply_sequence_clipping(self, importance_ratios: torch.Tensor, advantages: torch.Tensor) ->
        """
        Apply sequence-level clipping as in Equation (5):
        min(s_i(θ)Â_i, clip(s_i(θ), 1-ε, 1+ε)Â_i)
        """
        clip_min, clip_max = self.clip_range

        # Apply clipping to importance ratios
        clipped_ratios = torch.clamp(importance_ratios, clip_min, clip_max)

        # Choose between clipped and unclipped based on advantage sign
```

```
        unclipped_objective = importance_ratios * advantages
        clipped_objective = clipped_ratios * advantages

        # Take minimum as in PPO-style clipping
        final_objective = torch.where(
            advantages >= 0,
            torch.minimum(unclipped_objective, clipped_objective),
            torch.maximum(unclipped_objective, clipped_objective)
        )

        return final_objective

    def compute_gspo_loss(self,
                          queries: List[torch.Tensor],
                          responses: List[torch.Tensor],
                          old_likelihoods: List[torch.Tensor],
                          rewards: List[float]) -> torch.Tensor:
        """
        Compute GSPO loss for a batch of query-response pairs.
        Implements the main objective from Equation (5).
        """
        batch_size = len(queries)
        total_loss = 0.0

        # Process each query group separately
        for i in range(batch_size):
            query = queries[i].unsqueeze(0).to(self.device)
            response = responses[i].unsqueeze(0).to(self.device)
            old_likelihood = old_likelihoods[i].to(self.device)
            response_length = torch.tensor(response.shape[1], dtype=torch.float, device=self.device

            # Compute current sequence likelihood
            current_likelihood = self.compute_sequence_likelihood(query, response)

            # Compute importance ratio
            importance_ratio = self.compute_importance_ratio(
                current_likelihood, old_likelihood, response_length
            )

            # Compute advantages for this response group
            group_rewards = rewards[i:i+1]  # In practice, this would be multiple responses per que
```

```
            advantages = self.compute_group_advantages(group_rewards)

            # Apply sequence clipping and compute loss
            clipped_objective = self.apply_sequence_clipping(importance_ratio, advantages)

            # Negative log likelihood for policy gradient
            with torch.no_grad():
                logits = self.policy_model(query, response[:, :-1])
                log_probs = F.log_softmax(logits, dim=-1)
                token_log_probs = torch.gather(log_probs, -1, response[:, 1:].unsqueeze(-1)).squeez
                nll_loss = -torch.sum(token_log_probs) / response_length

            # Combine with clipped objective
            loss = nll_loss * clipped_objective
            total_loss += loss

        return total_loss / batch_size
```

Key aspects:
- Implements sequence-level importance sampling instead of token-level
- Uses length normalization to control variance across different response lengths
- Applies PPO-style clipping at the sequence level rather than token level
- Computes group-based advantages to eliminate the need for value models

GSPO-Token Variant Implementation

This implements the GSPO-token variant from Equation (13), allowing token-wise advantage customization while maintaining sequence-level importance weighting.

```
class GSPOTokenOptimizer(GSPOOptimizer):
    def __init__(self, policy_model, clip_range: Tuple[float, float] = (3e-4, 4e-4)):
        super().__init__(policy_model, clip_range)

    def compute_token_importance_ratio(self,
                                       query: torch.Tensor,
                                       response: torch.Tensor,
                                       sequence_importance: torch.Tensor) -> torch.Tensor:
        """
        Compute token-level importance ratios for GSPO-token variant from Equation (14):
        s_i,t(θ) = sg[s_i(θ)] * (π_θ(y_i,t|x,y_i,<t) / sg[π_θ(y_i,t|x,y_i,<t)])
        """
```

```
        with torch.no_grad():
            # Get current policy logits
            logits = self.policy_model(query, response[:, :-1])
            current_probs = F.softmax(logits, dim=-1)

            # Get current token probabilities (stop gradient for denominator)
            current_token_probs = torch.gather(current_probs, -1, response[:, 1:].unsqueeze(-1)).sq

        # Compute token-level importance ratios
        # The sg[π_θ(y_i,t)] term has numerical value of 1, so s_i,t(θ) = s_i(θ) numerically
        token_importance = sequence_importance.unsqueeze(-1).expand_as(current_token_probs)

        return token_importance

    def compute_gspo_token_loss(self,
                                queries: List[torch.Tensor],
                                responses: List[torch.Tensor],
                                old_likelihoods: List[torch.Tensor],
                                rewards: List[float],
                                token_advantages: List[torch.Tensor] = None) -> torch.Tensor:
        """
        Compute GSPO-token loss with token-wise advantages from Equation (13).
        """
        batch_size = len(queries)
        total_loss = 0.0

        for i in range(batch_size):
            query = queries[i].unsqueeze(0).to(self.device)
            response = responses[i].unsqueeze(0).to(self.device)
            old_likelihood = old_likelihoods[i].to(self.device)
            response_length = torch.tensor(response.shape[1], dtype=torch.float, device=self.device

            # Compute sequence importance ratio (same as standard GSPO)
            current_likelihood = self.compute_sequence_likelihood(query, response)
            sequence_importance = self.compute_importance_ratio(
                current_likelihood, old_likelihood, response_length
            )

            # Get token-level importance ratios
            token_importance = self.compute_token_importance_ratio(
                query, response, sequence_importance
```

```
            )

            # Use provided token advantages or fall back to sequence-level advantages
            if token_advantages is not None:
                advantages = token_advantages[i].to(self.device)
            else:
                group_rewards = rewards[i:i+1]
                advantages = self.compute_group_advantages(group_rewards)
                advantages = advantages.expand_as(token_importance)

            # Apply clipping at token level
            clipped_objective = self.apply_sequence_clipping(token_importance, advantages)

            # Compute policy gradient loss
            with torch.no_grad():
                logits = self.policy_model(query, response[:, :-1])
                log_probs = F.log_softmax(logits, dim=-1)
                token_log_probs = torch.gather(log_probs, -1, response[:, 1:].unsqueeze(-1)).squeez
                nll_loss = -token_log_probs

            # Combine losses
            loss = (nll_loss * clipped_objective).mean()
            total_loss += loss

        return total_loss / batch_size
```

Key aspects:
- Maintains the same numerical optimization as standard GSPO when token advantages are uniform
- Allows fine-grained advantage customization for multi-turn RL scenarios
- Preserves sequence-level stability while enabling token-level control

Training Loop Implementation

```
class GSPOTrainer:
    def __init__(self,
                 policy_model,
                 reward_model,
                 optimizer,
                 gspo_optimizer: GSPOOptimizer,
```

```
                group_size: int = 4):
        self.policy_model = policy_model
        self.reward_model = reward_model
        self.optimizer = optimizer
        self.gspo_optimizer = gspo_optimizer
        self.group_size = group_size
        self.device = next(policy_model.parameters()).device

    def generate_responses(self, queries: List[str], max_length: int = 512) -> List[torch.Tensor]:
        """Generate multiple responses per query for group-based training."""
        responses = []

        for query in queries:
            query_responses = []
            for _ in range(self.group_size):
                # Generate response using current policy
                response = self.policy_model.generate(query, max_length=max_length)
                query_responses.append(response)
            responses.append(query_responses)

        return responses

    def compute_rewards(self, queries: List[str], response_groups: List[List[torch.Tensor]]) -> Lis
        """Compute rewards for all generated responses."""
        all_rewards = []

        for query, responses in zip(queries, response_groups):
            query_rewards = []
            for response in responses:
                reward = self.reward_model.evaluate(query, response)
                query_rewards.append(reward)
            all_rewards.append(query_rewards)

        return all_rewards

    def train_step(self, queries: List[str]) -> Dict[str, float]:
        """Perform one GSPO training step."""
        # Generate responses for current policy
        with torch.no_grad():
            old_responses = self.generate_responses(queries)
```

```
        # Compute old likelihoods
        old_likelihoods = []
        for query, responses in zip(queries, old_responses):
            query_likelihoods = []
            for response in responses:
                likelihood = self.gspo_optimizer.compute_sequence_likelihood(
                    torch.tensor(query).unsqueeze(0).to(self.device),
                    response.unsqueeze(0).to(self.device)
                )
                query_likelihoods.append(likelihood)
            old_likelihoods.append(query_likelihoods)

        # Generate new responses and compute rewards
        new_responses = self.generate_responses(queries)
        rewards = self.compute_rewards(queries, new_responses)

        # Compute GSPO loss
        loss = 0.0
        for i, (query, old_group_likelihoods, new_group_responses, group_rewards) in enumerate(
            zip(queries, old_likelihoods, new_responses, rewards)
        ):
            for j, (old_likelihood, new_response, reward) in enumerate(
                zip(old_group_likelihoods, new_group_responses, group_rewards)
            ):
                step_loss = self.gspo_optimizer.compute_gspo_loss(
                    [torch.tensor(query)],
                    [new_response],
                    [old_likelihood],
                    [reward]
                )
                loss += step_loss

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

        # Compute metrics
        avg_reward = sum(sum(group) for group in rewards) / sum(len(group) for group in rewards)

        return {
```

```
            'loss': loss.item(),
            'avg_reward': avg_reward,
            'num_responses': sum(len(group) for group in new_responses)
        }

    def train(self,
              dataset: List[str],
              num_epochs: int,
              save_interval: int = 100) -> List[Dict[str, float]]:
        """Main training loop."""
        metrics_history = []

        for epoch in range(num_epochs):
            epoch_metrics = []

            # Process queries in batches
            for i in range(0, len(dataset), self.group_size):
                batch_queries = dataset[i:i+self.group_size]

                # Perform training step
                step_metrics = self.train_step(batch_queries)
                epoch_metrics.append(step_metrics)

            # Aggregate epoch metrics
            epoch_summary = {
                'epoch': epoch,
                'avg_loss': sum(m['loss'] for m in epoch_metrics) / len(epoch_metrics),
                'avg_reward': sum(m['avg_reward'] for m in epoch_metrics) / len(epoch_metrics),
                'total_responses': sum(m['num_responses'] for m in epoch_metrics)
            }
            metrics_history.append(epoch_summary)

            print(f"Epoch {epoch}: Loss = {epoch_summary['avg_loss']:.4f}, "
                  f"Reward = {epoch_summary['avg_reward']:.4f}")

            # Save checkpoint
            if epoch % save_interval == 0:
                torch.save({
                    'epoch': epoch,
                    'policy_model_state_dict': self.policy_model.state_dict(),
                    'optimizer_state_dict': self.optimizer.state_dict(),
```

```
                    'metrics': epoch_summary
                }, f'gspo_checkpoint_epoch_{epoch}.pt')

        return metrics_history
```

Key aspects:
- Implements group-based training with multiple responses per query
- Handles advantage computation without requiring value models
- Supports both standard GSPO and GSPO-token variants
- Includes checkpointing and metrics tracking

Relation to Paper

Paper Section	Code Component	Notes
Section 4.1: GSPO Algorithm	`GSPOOptimizer.compute_gspo_loss()`	Implements main objective from Equation (5)
Equation 6: Group Advantages	`GSPOOptimizer.compute_group_advantages()`	Advantage estimation without value model
Equation 7: Importance Ratio	`GSPOOptimizer.compute_importance_ratio()`	Sequence-level importance with length normalization
Section 4.3: GSPO-token	`GSPOTokenOptimizer.compute_gspo_token_loss()`	Token-level variant implementation
Section 5: Experiments	`GSPOTrainer.train()`	Training loop matching paper's experimental setup
MoE Training	Architecture handles sparse models	No routing replay needed for stable training

Key Differences from Paper

- **Implementation Detail**: The code includes gradient computation and optimization steps that are implied but not explicitly detailed in the paper
- **Batch Processing**: The implementation handles batching and parallelization considerations not covered in the theoretical description
- **Numerical Stability**: Additional safeguards against numerical instability are included (e.g., handling zero standard deviation in advantage computation)
- **Memory Management**: The code includes memory-efficient implementations for large-scale training scenarios