Relevant Code

Code Repository

Repository: Pseudocode - No Official Repository Found

Status: Conceptual Implementation Based on Paper

Language: Python (PyTorch-based pseudocode)

Last Updated: November 2025 (based on arXiv:2203.02155)

Stars/Popularity: N/A (educational pseudocode)

Architecture Overview

The InstructGPT implementation follows a three-stage architecture that transforms a pre-trained language model into an instruction-following assistant. The system consists of:

1. **Supervised Fine-Tuning (SFT) Module**: Handles initial training on human demonstrations
2. **Reward Model (RM) Module**: Learns to predict human preferences between different outputs
3. **Reinforcement Learning (RL) Module**: Optimizes the policy using PPO with the reward model as the objective function

The architecture is modular, allowing each component to be trained independently and iteratively. The base GPT-3 model serves as the foundation, with each stage progressively aligning the model better with human intentions.

Directory Structure

```
instructgpt/
├── src/
│   ├── models/
│   │   ├── sft_model.py          # Supervised fine-tuning implementation
│   │   ├── reward_model.py       # Reward model for human preferences
│   │   ├── ppo_model.py          # PPO-optimized policy model
│   │   └── base_gpt3.py          # Base GPT-3 architecture
```

```
│   ├── training/
│   │   ├── supervised_trainer.py # SFT training loop
│   │   ├── reward_trainer.py     # RM training on comparisons
│   │   ├── ppo_trainer.py        # RL optimization with PPO
│   │   └── data_loader.py        # Dataset handling utilities
│   ├── data/
│   │   ├── prompt_dataset.py     # API prompt and demonstration handling
│   │   ├── comparison_dataset.py # Preference comparison processing
│   │   └── pretrain_mix.py       # Pretraining data mixing for PPO-ptx
│   └── evaluation/
│       ├── human_evaluator.py    # Labeler preference evaluation
│       ├── truthfulness_eval.py  # TruthfulQA benchmark evaluation
│       └── toxicity_eval.py      # RealToxicityPrompts evaluation
├── configs/
│   ├── sft_config.yaml           # Supervised fine-tuning parameters
│   ├── rm_config.yaml            # Reward model configuration
│   └── ppo_config.yaml           # PPO optimization settings
└── scripts/
    ├── train_sft.py              # SFT training script
    ├── train_rm.py               # Reward model training
    ├── train_ppo.py              # PPO optimization
    └── evaluate.py               # Comprehensive evaluation
```

Key Implementation

Supervised Fine-Tuning (SFT)

The SFT component trains the base GPT-3 model on human demonstrations of desired behavior. This stage provides the foundation for instruction following.

```
class SFTModel(nn.Module):
    """
    Supervised Fine-Tuning model for instruction following.
    Based on GPT-3 architecture with instruction-tuning capabilities.
    """

    def __init__(self, base_model_path: str, max_length: int = 2048):
        super().__init__()
        self.base_model = self.load_gpt3_model(base_model_path)
        self.max_length = max_length
```

```
    def load_gpt3_model(self, model_path: str):
        """Load pre-trained GPT-3 model weights"""
        # Implementation loads GPT-3 with standard transformer architecture
        pass

    def forward(self, input_ids: torch.Tensor, attention_mask: torch.Tensor):
        """Standard language modeling forward pass"""
        outputs = self.base_model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            return_dict=True
        )
        return outputs.logits

    def generate(self, prompt: str, max_new_tokens: int = 256):
        """Generate response following instruction style"""
        inputs = self.tokenizer(prompt, return_tensors="pt", truncation=True)
        outputs = self.base_model.generate(
            **inputs,
            max_new_tokens=max_new_tokens,
            temperature=0.7,
            do_sample=True,
            pad_token_id=self.tokenizer.eos_token_id
        )
        return self.tokenizer.decode(outputs[0], skip_special_tokens=True)

class SFTTrainer:
    """Trainer for supervised fine-tuning stage"""

    def __init__(self, model: SFTModel, config: dict):
        self.model = model
        self.config = config
        self.optimizer = torch.optim.AdamW(
            model.parameters(),
            lr=config['learning_rate'],
            weight_decay=config['weight_decay']
        )
        self.scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
            self.optimizer, T_max=config['epochs']
        )
```

```
    def train_epoch(self, dataloader):
        """Train for one epoch on demonstration data"""
        self.model.train()
        total_loss = 0

        for batch in dataloader:
            # batch contains: input_ids, attention_mask, labels
            input_ids = batch['input_ids'].to(self.device)
            attention_mask = batch['attention_mask'].to(self.device)
            labels = batch['labels'].to(self.device)

            # Forward pass
            logits = self.model(input_ids, attention_mask)

            # Calculate language modeling loss
            loss_fct = nn.CrossEntropyLoss(ignore_index=-100)
            loss = loss_fct(logits.view(-1, logits.size(-1)), labels.view(-1))

            # Backward pass
            self.optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
            self.optimizer.step()

            total_loss += loss.item()

        self.scheduler.step()
        return total_loss / len(dataloader)
```

Key aspects:
- Trains on ~13k human demonstrations across diverse tasks
- Uses 16 epochs with cosine learning rate decay
- Implements residual dropout of 0.2 for regularization
- Selects best model based on reward model validation score

Reward Model (RM) Training

The reward model learns to predict human preferences between different model outputs, serving as the objective function for reinforcement learning.

```
class RewardModel(nn.Module):
    """
    Reward model for predicting human preferences between model outputs.
    Takes prompt and response, outputs scalar reward score.
    """

    def __init__(self, sft_model: SFTModel):
        super().__init__()
        # Use SFT model as base, remove final language modeling head
        self.transformer = sft_model.base_model
        self.reward_head = nn.Linear(self.transformer.config.n_embd, 1)

    def forward(self, input_ids: torch.Tensor, attention_mask: torch.Tensor):
        """Forward pass to compute reward score"""
        outputs = self.transformer(
            input_ids=input_ids,
            attention_mask=attention_mask,
            output_hidden_states=True
        )

        # Use last hidden state of last token for reward prediction
        last_hidden_state = outputs.last_hidden_state
        # Find last non-padding token
        sequence_lengths = attention_mask.sum(dim=1) - 1
        batch_size = input_ids.size(0)

        reward_tokens = last_hidden_state[range(batch_size), sequence_lengths]
        reward = self.reward_head(reward_tokens).squeeze(-1)

        return reward

class RewardModelTrainer:
    """Trainer for reward model using human comparison data"""

    def __init__(self, model: RewardModel, config: dict):
        self.model = model
        self.config = config
        self.optimizer = torch.optim.AdamW(model.parameters(), lr=config['learning_rate'])

    def compute_comparison_loss(self, batch):
```

```
        """
        Compute loss for K choose 2 comparisons from each prompt.
        All comparisons from same prompt processed as single batch element.
        """
        # batch contains: prompt_ids, chosen_ids, rejected_ids, attention_masks
        prompt_ids = batch['prompt_ids']
        chosen_ids = batch['chosen_ids']
        rejected_ids = batch['rejected_ids']

        # Concatenate prompt with each response
        chosen_input = torch.cat([prompt_ids, chosen_ids], dim=1)
        rejected_input = torch.cat([prompt_ids, rejected_ids], dim=1)

        # Create attention masks
        chosen_mask = (chosen_input != self.tokenizer.pad_token_id).long()
        rejected_mask = (rejected_input != self.tokenizer.pad_token_id).long()

        # Get reward scores
        chosen_reward = self.model(chosen_input, chosen_mask)
        rejected_reward = self.model(rejected_input, rejected_mask)

        # Compute pairwise comparison loss
        # Loss encourages chosen_reward > rejected_reward
        reward_diff = chosen_reward - rejected_reward
        loss = -F.logsigmoid(reward_diff).mean()

        return loss

    def train_step(self, batch):
        """Single training step on comparison data"""
        self.model.train()
        self.optimizer.zero_grad()

        loss = self.compute_comparison_loss(batch)
        loss.backward()

        torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
        self.optimizer.step()

        return loss.item()
```

Key aspects:
- Trains on ~33k comparison prompts with 4-9 outputs ranked per prompt
- Processes all K choose 2 comparisons from each prompt as single batch
- Uses cross-entropy loss with preference pairs as labels
- 6B parameter model size for stability and efficiency

PPO Reinforcement Learning

The PPO stage optimizes the SFT model using the reward model as the objective function, with KL penalties to prevent over-optimization.

```
class PPOModel(nn.Module):
    """
    PPO-optimized policy model for instruction following.
    Uses reward model as objective function with KL regularization.
    """

    def __init__(self, sft_model: SFTModel, reward_model: RewardModel):
        super().__init__()
        self.policy = sft_model  # Policy to optimize
        self.reward_model = reward_model
        self.reference_model = copy.deepcopy(sft_model)  # For KL penalty

    def forward(self, input_ids: torch.Tensor, attention_mask: torch.Tensor):
        """Policy forward pass"""
        return self.policy(input_ids, attention_mask)

    def generate_response(self, prompt_ids: torch.Tensor, max_length: int = 256):
        """Generate response using current policy"""
        return self.policy.generate(prompt_ids, max_new_tokens=max_length)

class PPOTrainer:
    """PPO trainer with pretraining mix (PPO-ptx) for alignment"""

    def __init__(self, model: PPOModel, config: dict):
        self.model = model
        self.config = config
        self.kl_coef = config.get('kl_coef', 0.2)
        self.pretrain_coef = config.get('pretrain_coef', 0.0)

        self.policy_optimizer = torch.optim.AdamW(
```

```
            model.policy.parameters(), lr=config['policy_lr']
        )

    def compute_ppo_loss(self, batch):
        """Compute PPO loss with KL penalty and pretraining mix"""
        prompt_ids = batch['prompt_ids']
        prompt_mask = batch['prompt_mask']

        # Generate responses using current policy
        with torch.no_grad():
            response_ids, response_logprobs = self._sample_responses(prompt_ids)

        # Concatenate prompt and response
        full_ids = torch.cat([prompt_ids, response_ids], dim=1)
        full_mask = torch.cat([prompt_mask, response_ids != self.tokenizer.pad_token_id], dim=1)

        # Compute rewards using reward model
        with torch.no_grad():
            rewards = self.model.reward_model(full_ids, full_mask)

        # Compute reference log probabilities (for KL penalty)
        with torch.no_grad():
            ref_logprobs = self._compute_logprobs(
                self.model.reference_model, prompt_ids, response_ids
            )

        # Compute current policy log probabilities
        current_logprobs = self._compute_logprobs(
            self.model.policy, prompt_ids, response_ids
        )

        # Compute KL penalty
        kl_penalty = current_logprobs - ref_logprobs
        kl_reward = -self.kl_coef * kl_penalty

        # Total reward = model reward + KL penalty
        total_rewards = rewards + kl_reward

        # Compute PPO loss
        ratio = torch.exp(current_logprobs - response_logprobs)
        clipped_ratio = torch.clamp(ratio, 1 - self.config['clip_eps'], 1 + self.config['clip_eps']
```

```
        policy_loss = -torch.min(ratio * total_rewards, clipped_ratio * total_rewards).mean()

        return policy_loss

    def compute_pretrain_loss(self, pretrain_batch):
        """Compute pretraining language modeling loss for PPO-ptx"""
        input_ids = pretrain_batch['input_ids']
        attention_mask = pretrain_batch['attention_mask']
        labels = pretrain_batch['labels']

        logits = self.model.policy(input_ids, attention_mask)

        loss_fct = nn.CrossEntropyLoss(ignore_index=-100)
        pretrain_loss = loss_fct(logits.view(-1, logits.size(-1)), labels.view(-1))

        return pretrain_loss

    def train_step(self, batch, pretrain_batch=None):
        """Combined PPO and pretraining update"""
        self.model.train()

        # PPO update
        self.policy_optimizer.zero_grad()
        ppo_loss = self.compute_ppo_loss(batch)

        total_loss = ppo_loss

        # Add pretraining loss if provided (PPO-ptx)
        if pretrain_batch is not None and self.pretrain_coef > 0:
            pretrain_loss = self.compute_pretrain_loss(pretrain_batch)
            total_loss += self.pretrain_coef * pretrain_loss

        total_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.model.policy.parameters(), 1.0)
        self.policy_optimizer.step()

        return {
            'ppo_loss': ppo_loss.item(),
            'total_loss': total_loss.item(),
            'pretrain_loss': pretrain_loss.item() if pretrain_batch is not None else 0.0
```

```
        }

    def _sample_responses(self, prompt_ids):
        """Sample responses from current policy"""
        # Implementation samples multiple responses and keeps best based on reward
        pass

    def _compute_logprobs(self, model, prompt_ids, response_ids):
        """Compute log probabilities of responses under given model"""
        # Implementation computes log probabilities of generated responses
        pass
```

Key aspects:
- Uses PPO algorithm with reward model as scalar reward function
- Includes per-token KL penalty from SFT model to prevent over-optimization
- PPO-ptx variant mixes pretraining gradients to minimize performance regressions
- Operates in bandit environment with random prompts and episode termination

Relation to Paper

Paper Section	Code Component	Notes
Section 3.1: High-level methodology	`SFTTrainer`, `RewardModelTrainer`, `PPOTrainer`	Implements three-step RLHF process
Algorithm 1: SFT training	`SFTTrainer.train_epoch()`	Supervised fine-tuning on demonstrations
Equation 1: Reward model loss	`RewardModelTrainer.compute_comparison_loss()`	Cross-entropy loss on preference pairs
Section 3.5: PPO optimization	`PPOTrainer.compute_ppo_loss()`	PPO with KL penalty and pretraining mix
Equation 2: PPO-ptx objective	`PPOTrainer.train_step()`	Combined PPO and pretraining objectives

Paper Section	Code Component	Notes
Figure 2: Three-step diagram	Module structure	Architecture follows paper's process flow
Section 4.1: API evaluation	`HumanEvaluator` class	Implements labeler preference evaluation

Key Differences from Paper

- **Simplified implementation**: Pseudocode focuses on core algorithmic concepts rather than engineering optimizations
- **Missing distributed training**: Paper uses large-scale distributed training not shown here
- **Hyperparameter details**: Specific hyperparameters from paper's Appendix C not fully implemented
- **Data pipeline**: Actual API data processing and PII filtering not detailed
- **Evaluation metrics**: Full evaluation suite from Section 3.6 simplified for clarity

Usage Examples

```
# Example usage of the InstructGPT training pipeline

# 1. Initialize models
base_model = SFTModel("gpt3-base")
reward_model = RewardModel(base_model)
ppo_model = PPOModel(base_model, reward_model)

# 2. Train SFT model
sft_trainer = SFTTrainer(base_model, sft_config)
for epoch in range(16):  # Paper uses 16 epochs
    loss = sft_trainer.train_epoch(sft_dataloader)
    print(f"SFT Epoch {epoch}: Loss = {loss:.4f}")

# 3. Train reward model
rm_trainer = RewardModelTrainer(reward_model, rm_config)
for step in range(num_rm_steps):
```

```
    loss = rm_trainer.train_step(comparison_batch)
    if step % 100 == 0:
        print(f"RM Step {step}: Loss = {loss:.4f}")

# 4. Train PPO model
ppo_trainer = PPOTrainer(ppo_model, ppo_config)
for step in range(num_ppo_steps):
    # Get prompts from API distribution
    batch = sample_prompts(api_prompts)

    # Get pretraining data (for PPO-ptx)
    pretrain_batch = sample_pretrain_data() if ppo_trainer.pretrain_coef > 0 else None

    # PPO update
    metrics = ppo_trainer.train_step(batch, pretrain_batch)

    if step % 100 == 0:
        print(f"PPO Step {step}: {metrics}")

# 5. Generate responses
final_model = ppo_model.policy
prompt = "Write a short story about a space explorer discovering a new planet."
response = final_model.generate(prompt)
print(f"Prompt: {prompt}\nResponse: {response}")
```

This pseudocode provides the core algorithmic foundation for implementing the InstructGPT methodology described in the paper, demonstrating how human feedback can be used to align language models with user intentions.