CS336 Assignment5 SFT部分

cs336的第五章主要是做alignment，需要从0实现SFT与GRPO的训练

SFT的实现

tokenize_prompt_and_output

这个函数用于将SFT数据的问题以及回答部分进行tokenize，以方便在训练时使用。

我们需要将问题与回答的文本进行拼接，然后再tokenize。同时，由于大模型都是预测next token的概率，因此label就是输入往后移一位。

在计算对数概率的时候，我们只需要回答的部分，因此需要一个掩码，以便在之后的运算中将prompt部分mask掉

```python
def tokenize_prompt_and_output(prompts, outputs, tokenizer: PreTrainedTokenizer):
    prompt_ids = []
    output_ids = []
    resp_mask = []
    for prompt, output in zip(prompts, outputs):
        prompt_id = tokenizer(prompt, padding=True, truncation=True)
        output_id = tokenizer(output, padding=True, truncation=True)
        output_ids.append(prompt_id['input_ids'] + output_id['input_ids'])
        resp_mask.append([0] * len(prompt_id['input_ids']) + [1] * len(output_id['input_ids']))
    batch_size = len(output_ids)
    max_seq_len = max(len(output_id) for output_id in output_ids)
    ids_tensor = torch.full((batch_size, max_seq_len), tokenizer.pad_token_id, dtype=torch.long)
    resp_mask_tensor = torch.full((batch_size, max_seq_len), 0, dtype=torch.long)
    for i, (output_id, resp_mask) in enumerate(zip(output_ids, resp_mask)):
        ids_tensor[i, :len(output_id)] = torch.tensor(output_id, dtype=torch.long)
        resp_mask_tensor[i, :len(resp_mask)] = torch.tensor(resp_mask, dtype=torch.long)
    return {
        "input_ids": ids_tensor[:, :-1],
        "labels": ids_tensor[:, 1:],
        "response_mask": resp_mask_tensor[:, 1:]
    }
```

Per-token entropy

在做SFT与RL时，我们要用熵来衡量语言模型在预测下一个词的时候有多“确定”或多“不确定”。 这可以帮我们判断模型在经过训练后，是否变得过于“自信”了。熵的计算公式如下：

$$
H(p) = - \sum_{i=1}^{n} p(x_i) \log_b p(x_i)
$$

```python
def compute_entropy(logits: Tensor):
    p = F.log_softmax(logits, dim=-1)
    return -torch.sum(torch.exp(p) * p, dim=-1)
```

当 `softmax` 的输出概率 `probs` 中有非常接近0的值时， `torch.log(probs)` 会得到一个非常大的负数（接近负无穷）。在计算机浮点数运算中，这很容易导致精度问题或 `NaN` 的结果。

而 `F.log_softmax` 函数在内部使用了专门的数学技巧（如 Log-Sum-Exp trick）来避免这些问题，使得计算过程在数值上更加稳定和精确。因此，在实践中，**先用 `log_softmax` 再用 `exp` 是计算熵的标准做法。**

response log-probabilities

```python
def get_response_log_probs(
    model: PreTrainedModel,
    input_ids: torch.Tensor,
    labels: torch.Tensor,
    return_token_entropy: bool = False,
) -> dict[str, torch.Tensor]:
    logits = model(input_ids).logits
    log_probs = F.log_softmax(logits, dim=-1) # (batch_size, seq_len, vocab_size)
    res = {}
    res["log_probs"] = log_probs.gather(dim=-1, index=labels.unsqueeze(-1)).squeeze(-1) # (batch_size, seq_len)
    if return_token_entropy:
        res["token_entropy"] = compute_entropy(logits) # (batch_size, seq_len)
    return res
```

在这个函数中，我们先通过model计算出其对于输入token的logits，然后计算这些token的对数概率。

由于模型输出的最后一个维度大小是vocab_size，即词表中所有词汇的概率，而我们只需要对应的token的概率，因此使用gather来提取。

masked_normalize

我们在SFT（监督微调）中最小化的损失函数，是在给定提示（prompt）的情况下，目标输出（target output）的**负对数似然（negative log-likelihood）**。为了计算这个损失，我们需要计算出在给定提示

下，模型对目标输出中每个token的对数概率（log-probabilities），然后将输出部分所有词元的这些值相加，同时屏蔽掉提示和填充（padding）部分的token。

```python
def masked_normalize(
    tensor: torch.Tensor,
    mask: torch.Tensor,
    normalize_constant: float,
    dim: int | None = None,
) -> torch.Tensor:
    masked_tensor = tensor * mask
    if dim is None:
        return masked_tensor.sum() / normalize_constant
    return masked_tensor.sum(dim=dim) /  normalize_constant
```

SFT microbatch train step

```python
def sft_microbatch_train_step(
    policy_log_probs: torch.Tensor,
    response_mask: torch.Tensor,
    gradient_accumulation_steps: int,
    normalize_constant: float = 1.0,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    batch_size, seq_len = policy_log_probs.shape
    loss_sum = -masked_normalize(policy_log_probs, response_mask, normalize_constant)
    loss = loss_sum / batch_size / gradient_accumulation_steps
    loss.backward()
    metadata = {
        "loss":loss.item()
    }
    return loss.detach(), metadata
```

SFT experiment

现在可以实现完整的SFT流程了。由于MATH的sft数据并没有提供，我们可以使用作业中提供的gsm8k数据集，其位置在data/gsm8k gsm8k的一条数据如下：

```json
{
  "question": "Janet\u2019s ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder at the farmers' market daily for $2 per fresh duck egg. How much in dollars does she make every day at the farmers' market?",
  "answer": "Janet sells 16 - 3 - 4 = <<16-3-4=9>>9 duck eggs a day.\nShe makes 9 * 2 = $<<9*2=18>>18 every day at the farmer\u2019s market.\n#### 18"
}
```

其中answer部分包含了CoT和答案，于是我们可以对数据进行简单地处理

```python
data = {"prompt": [], "label": [], "answer": []}
with open(path, "r", encoding="utf-8") as f:
    for line in f:
        item = json.loads(line)
        data["prompt"].append(template.format(question=item["question"]))
        response = item["answer"].split("####") # CoT和answer用####来分开
        output, answer = response[0], response[1].strip()
        data["label"].append(output + "</think> <answer> " + answer + " </answer>")
        data["answer"].append(answer)
path = DATA_PATH / "sft_test.jsonl"
with open(path, "w", encoding="utf-8") as f:
    for i in range(len(data["prompt"])):
        item = {
            "prompt": data["prompt"][i],
            "label": data["label"][i],
            "answer": data["answer"][i]
        }
        f.write(json.dumps(item, ensure_ascii=False) + "\n")
```

我们使用了r1_zero的prompt，其内容如下：

```
A conversation between User and Assistant. The User asks a question, and the Assistant solves it. The Assistant first thinks about the reasoning process in the mind and then provides the User with the answer. The reasoning process is enclosed within <think> </think> and answer is enclosed within <answer> </answer> tags, respectively, i.e., <think> reasoning process here </think> <answer> answer here </answer>.
User: {question}
Assistant: <think>
```

处理后的单条数据如下：

```json
{
  "prompt": "A conversation between User and Assistant. The User asks a question, and the Assistant solves it. The Assistant first thinks about the reasoning process in the mind and then provides the User with the answer. The reasoning process is enclosed within <think> </think> and answer is enclosed within <answer> </answer> tags, respectively, i.e., <think> reasoning process here </think> <answer> answer here </answer>.\nUser: A robe takes 2 bolts of blue fiber and half that much white fiber.  How many bolts in total does it take?\nAssistant: <think>",
  "label": "It takes 2/2=<<2/2=1>>1 bolt of white fiber\nSo the total amount of fabric is 2+1=<<2+1=3>>3 bolts of fabric\n</think> <answer> 3 </answer>",
  "answer": "3"
}
```

training部分的代码如下

```python
process_bar = tqdm(range(total_batch), total = total_batch, desc=f'Training epoch {epoch + 1}/{num_epoch}')
global_step = 0
for i in process_bar:
    batch = self.train_data[i * self.batch_size: (i + 1) * self.batch_size]
    prompt, label = batch["prompt"], batch["label"]
    lossess = []
    for j in range(0, self.batch_size, self.micro_batch_size):
        micro_batch_prompt, micro_batch_label = prompt[j: j + self.micro_batch_size], label[j: j + self.micro_batch_size]
        micro_batch_data = tokenize_prompt_and_output(micro_batch_prompt, micro_batch_label, self.tokenizer)
        micro_input_ids, micro_labels, micro_response_mask = micro_batch_data["input_ids"].to(self.device), micro_batch_data["labels"].to(self.device), micro_batch_data["response_mask"].to(self.device)
        log_probs = get_response_log_probs(self.model, micro_input_ids, micro_labels)["log_probs"]
        loss, _ = sft_microbatch_train_step(
            log_probs,
            micro_response_mask,
            gradient_accumulation_steps=self.gradient_accumulation_steps,
            normalize_constant=self.normalize_constant,
        )
        micro_step = (j // self.micro_batch_size) + 1
        if (micro_step) % self.gradient_accumulation_steps == 0:
            gradient_clipping(self.model.parameters(), self.args.max_grad_norm)
            update_step = global_step // self.gradient_accumulation_steps
            lr = self.scheduler(update_step)
            self.optimizer.set_lr(lr)
            self.optimizer.step()
            self.optimizer.zero_grad()
            global_step += 1
        self._log_loss(process_bar, sum(lossess) / len(lossess), i)
    if (i + 1) % self.eval_interval == 0:
        eval_metrics = self.eval()
        pprint(eval_metrics)
```

由于我们设置了gradient_accumulation_steps，因此在相应步的microbatch之后更新一次参数

使用microbatch可以使得训练时的峰值显存降低，在gradient_accumulation_steps步进行一次更新，用时间换空间，通过多次计算来模拟一个大的批次，从而在有限的显存下实现大批量训练的效果

实验结果

由于只有3090，单卡无法进行全量微调，因此我在这使用了lora进行微调，效果与全量微调差异不大

Expert iteration

Algorithm 2 Expert iteration (EI)

Input initial policy model $\pi_{\theta_{\text{init}}}$; reward function R; task questions $\mathcal{D}$

policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
for step = 1, ..., `n_ei_steps` **do**
    Sample a batch of questions $\mathcal{D}_b$ from $\mathcal{D}$
    Set the old policy model $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
    Sample G outputs $\{o^{(i)}\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
    Compute rewards $\{r^{(i)}\}_{i=1}^{G}$ for each sampled output $o^{(i)}$ by running reward function $R(q, o^{(i)})$
    Filter out wrong outputs (i.e., $o^{(i)}$ with $r^{(i)} = 0$) to obtain a dataset $\mathcal{D}_{\text{sft}}$ of correct question-response pairs
    $\pi_\theta \leftarrow \text{SFT}(\pi_\theta, \mathcal{D}_{\text{sft}})$ (Algorithm 1)
end for

Output π_θ

对于每一个question，采样G条输出，过滤掉错误的输出，然后用得到的问答对来做sft

GRPO

Algorithm 3 Group Relative Policy Optimization (GRPO)

Input initial policy model $\pi_{\theta_{\text{init}}}$; reward function R; task questions $\mathcal{D}$

policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
for step = 1, ..., `n_grpo_steps` **do**
    Sample a batch of questions $\mathcal{D}_b$ from $\mathcal{D}$
    Set the old policy model $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
    Sample G outputs $\{o^{(i)}\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$ for each question $q \in \mathcal{D}_b$
    Compute rewards $\{r^{(i)}\}_{i=1}^{G}$ for each sampled output $o^{(i)}$ by running reward function $R(q, o^{(i)})$
    Compute $A^{(i)}$ with group normalization (Eq. 28)
    for train step = 1, ..., `n_train_steps_per_rollout_batch` **do**
        Update the policy model π_θ by maximizing the GRPO-Clip objective (to be discussed, Eq. 29)
    end for
end for

Output π_θ

GRPO算法是将PPO算法中的GAE优势替换为群体相对优势，具体来说：

1. 先对一个问题采样G条responses
2. 对这G条response计算奖励
3. 优势的计算公式 $A_i = (r_i - \text{mean}(r)) / \text{std}(r)$

compute_group_normalized_rewards

这个函数要求给定一组响应以及其对应的答案，计算群体相对优势

其中rollout_response的长度是 `n_prompt*group_size`

```python
def compute_group_normalized_rewards(
    reward_fn: Callable[[str, str], dict[str, float]],
    rollout_responses: list[str],
    repeated_ground_truths: list[str],
    group_size: int,
    advantage_eps: float,
    normalize_by_std: bool,
) -> Tuple[torch.Tensor, torch.Tensor, dict[str, float]]:
    rollout_batch_size = len(repeated_ground_truths)
    assert rollout_batch_size % group_size == 0
    advantages = torch.zeros(rollout_batch_size, dtype=torch.float32)
    rewards = torch.zeros(rollout_batch_size, dtype=torch.float32)
    for i in range(0, rollout_batch_size, group_size):
        batch_resp = rollout_responses[i : i + group_size]
        batch_gt = repeated_ground_truths[i : i + group_size]
        batch_advantages = []
        for resp, gt in zip(batch_resp, batch_gt):
            reward_dict = reward_fn(resp, gt)
            reward = reward_dict.get("reward", 0)
            batch_advantages.append(reward)
        rewards[i : i + group_size] = torch.tensor(batch_advantages, dtype=torch.float32)
        denom = advantage_eps + rewards[i : i + group_size].std() if normalize_by_std else 1
        advantages[i : i + group_size] = (rewards[i : i + group_size] - rewards[i : i + group_size].mean()) / denom

    metadata = {
        "reward_mean": rewards.mean().item(),
        "reward_std": rewards.std().item(),
        "advantage_mean": advantages.mean().item(),
        "advantage_std": advantages.std().item(),
        "response_length": sum([len(resp) for resp in rollout_responses]) // len(rollout_responses),
    }
    return advantages, rewards.detach(), metadata
```

Naive policy gradient loss

这个函数不计算重要性采样部分，只是简单地将优势与对数概率相乘即可

$$-A_t \cdot \log p_\theta(o_t|q, o_{<t}).$$

```python
def compute_naive_policy_gradient_loss(
    raw_rewards_or_advantages: torch.Tensor,
    policy_log_probs: torch.Tensor,
) -> torch.Tensor:
    return -raw_rewards_or_advantages * policy_log_probs
```

GRPO clip loss

这一部分是真正的GRPO的loss，我们需要引入重要性采样，同时引入clip，使得新策略与旧策略不会相差过大

```python
def compute_grpo_clip_loss(
    advantages: torch.Tensor,  # (batch_size, 1)
    policy_log_probs: torch.Tensor,  # (batch_size, sequence_length)
    old_log_probs: torch.Tensor,  # (batch_size, sequence_length)
    cliprange: float,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    ratio = torch.exp(policy_log_probs - old_log_probs)
    clipped_ratio = torch.clamp(ratio, 1 - cliprange, 1 + cliprange)
    loss = -torch.min(ratio, clipped_ratio) * advantages
    return loss, {"ratio": ratio.detach(), "clipped_ratio": clipped_ratio.detach()}
```

compute_policy_gradient_loss

这个函数用于在训练时作为接口，根据loss_type来区分使用哪个loss，方便后面做消融实验

```python
def compute_policy_gradient_loss(
    policy_log_probs: torch.Tensor,
    loss_type: Literal["no_baseline", "reinforce_with_baseline", "grpo_clip"],
    raw_rewards: torch.Tensor | None = None,
    advantages: torch.Tensor | None = None,
    old_log_probs: torch.Tensor | None = None,
    cliprange: float | None = None,
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    if loss_type == "no_baseline":
        assert raw_rewards is not None
        loss = compute_naive_policy_gradient_loss(raw_rewards, policy_log_probs)
        return loss, {}
    elif loss_type == "reinforce_with_baseline":
        assert advantages is not None
        loss = compute_naive_policy_gradient_loss(advantages, policy_log_probs)
        return loss, {}
    elif loss_type == "grpo_clip":
        assert advantages is not None
        assert old_log_probs is not None
        assert cliprange is not None
        loss, metadata = compute_grpo_clip_loss(advantages,policy_log_probs,old_log_probs,cliprange,)
        return loss, metadata
```

GRPO experiment

现在我们可以完整地实现GRPO训练的流程了，GRPO训练的流程大致如下：

1. 先采样一批问题 D
2. 从这批问题中，对每一个问题都采样G条答案

```
prompts = batch['prompts']
ground_truths = batch['ground_truths']

responses = self._generate_responses(prompts,  G)
```

3. 计算奖励与优势

```
advantages, raw_rewards, reward_metadata = compute_group_normalized_rewards(
            self.reward_fn,
            responses,
            repeated_ground_truths,
            self.args.group_size,
            self.args.advantage_eps,
            self.args.use_std_normalization
        )
```

responses是长度为(n_prompt*group_size)的list

最后得到的advatanges形状是(n_prompt*group_size,)

4. 使用当前的模型，计算所有response的对数概率

```
policy_log_probs, response_mask = self._get_log_probs(repeated_prompts, responses)
```

将prompt和response拼接，再次输入给大模型进行一次前向传播，得到每个token的logits

将当前的log_probs设置为old policy的log_probs

5. 将batch分成micro_batch，计算损失并更新参数

```
micro_policy_log_probs = policy_log_probs[i:end_idx]
micro_response_mask = response_mask[i:end_idx]
micro_advantages = advantages[i:end_idx].unsqueeze(-1)  # 添加维度以匹配
micro_raw_rewards = raw_rewards[i:end_idx].unsqueeze(-1)
micro_old_log_probs = old_log_probs[i:end_idx] if old_log_probs is not None else None
```

每一个micro_batch，计算一次损失：

```
loss, metadata = grpo_microbatch_train_step(
                micro_policy_log_probs,
                micro_response_mask,
                self.args.gradient_accumulation_steps,
                self.args.loss_type,
                raw_rewards=micro_raw_rewards if self.args.loss_type == "no_baseline" else None,
                advantages=micro_advantages if self.args.loss_type in ["reinforce_with_baseline", "grpo_clip"] else None,
                old_log_probs=micro_old_log_probs,
                cliprange=self.args.cliprange if self.args.loss_type == "grpo_clip" else None
            )
```

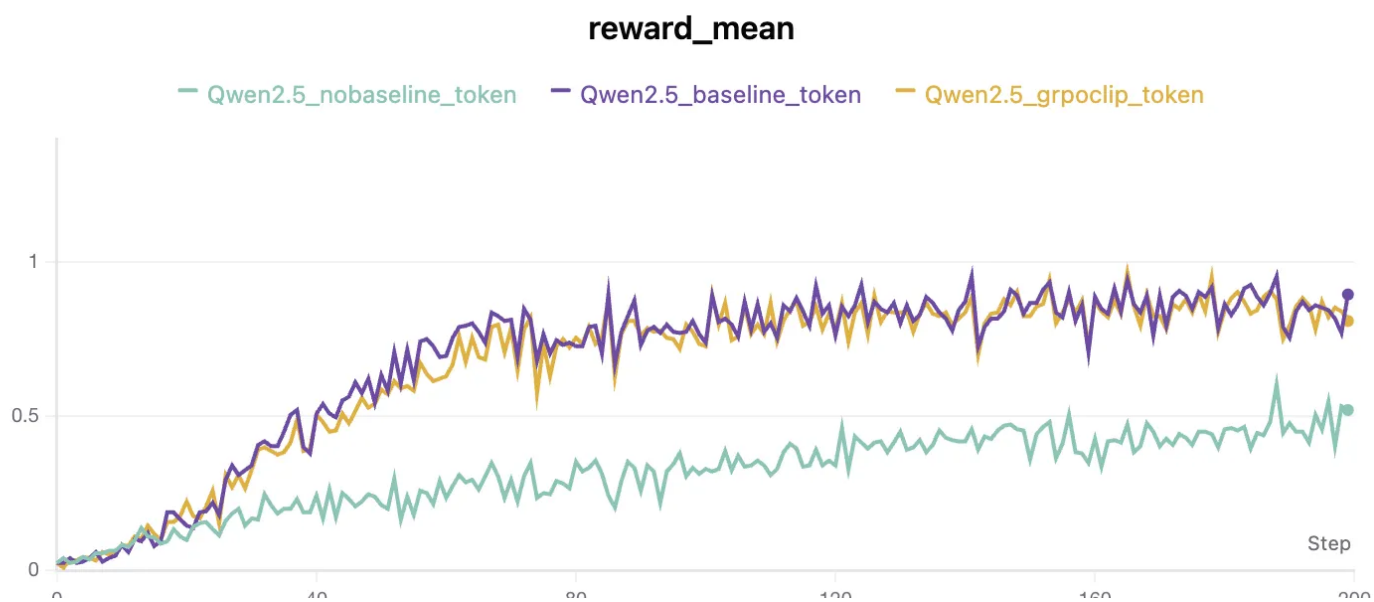
在训练过程中，用到了vllm来加快模型的推理，因此训练的时候需要用到两张GPU，一张用来装载训练模型，一张用于装载vllm模型。

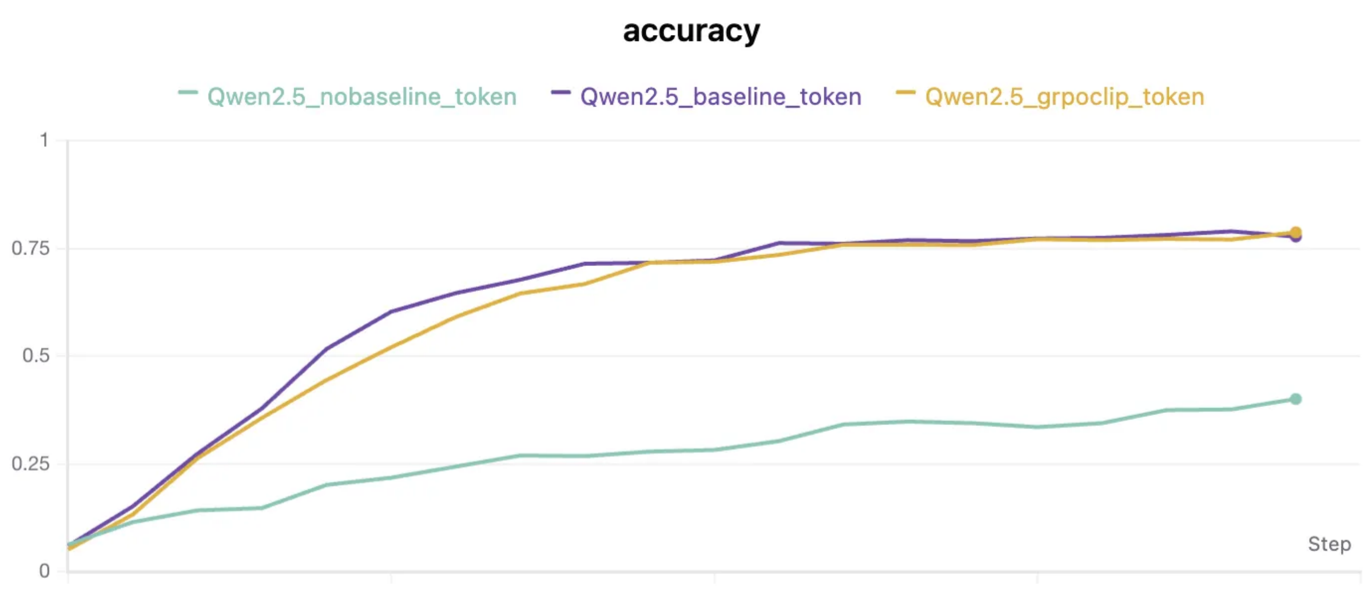
完整的训练代码如下:

```python
global_step = 0
process_bar = tqdm(self.train_dataloader, total=min(len(self.train_dataloader), self.args.n_grpo_steps), desc="Training")
for grpo_step, batch in enumerate(process_bar):
    if grpo_step >= self.args.n_grpo_steps:
        break
    self.policy.eval()
    if self.vllm_model is not None:
        self._load_policy_to_vllm()
    prompts = batch["prompts"]
    ground_truths = batch["ground_truths"]

    responses = self._generate_responses(prompts, n_samples=self.args.group_size)

    rollout_prompts = [p for p in prompts for _ in range(self.args.group_size)]
    repeated_ground_truths = [gt for gt in ground_truths for _ in range(self.args.group_size)]

    advantages, raw_rewards, reward_metadata = compute_group_normalized_rewards(
        self.reward_fn,
        responses,
        repeated_ground_truths,
        self.args.group_size,
        self.args.advantage_eps,
        self.args.use_std_normalization,
    )
    advantages = advantages.unsqueeze(-1)
    raw_rewards = raw_rewards.unsqueeze(-1)
    self._log(reward_metadata)
    if reward_metadata["reward_mean"] <= 0:
        print("Reward mean is zero, stop train.")
    break
    with torch.no_grad():
        if self.args.loss_type == "grpo_clip":
            old_log_probs = []
    for i in range(0, len(responses), self.micro_train_batch_size):
        micro_batch_prompts = rollout_prompts[i : i + self.micro_train_batch_size]
    micro_batch_responses = responses[i : i + self.micro_train_batch_size]
    masked_log_probs, _ = self._get_log_probs(micro_batch_prompts, micro_batch_responses, self.args.importance_sample_level)
    old_log_probs.append(masked_log_probs.detach())
```

```python
    else:
        old_log_probs = None
    # === Training Phase ===
    self.policy.train()
    for _ in range(self.args.epochs_per_rollout_batch):
        losses = []
        for i in range(0, len(responses), self.micro_train_batch_size):
            global_step += 1
            micro_batch_slice = slice(i, i + self.micro_train_batch_size)
            batch_prompts = rollout_prompts[micro_batch_slice]
            batch_responses = responses[micro_batch_slice]
            batch_advantages = advantages[micro_batch_slice].to(self.policy.device)
            batch_raw_rewards = raw_rewards[micro_batch_slice].to(self.policy.device)
            batch_old_log_probs = old_log_probs[i // self.micro_train_batch_size].to(self.policy.device) if old_log_probs  else None

            policy_log_probs, response_mask = self._get_log_probs(batch_prompts, batch_responses, self.args.importance_sample_level)

            loss, metadata = grpo_microbatch_train_step(
                policy_log_probs=policy_log_probs,
                response_mask=response_mask,
                gradient_accumulation_steps=self.args.gradient_accumulation_steps,
                loss_type=self.args.loss_type,
                raw_rewards=batch_raw_rewards,
                advantages=batch_advantages,
                old_log_probs=batch_old_log_probs,
                cliprange=self.args.cliprange,
            )
            if ((i // self.micro_train_batch_size) + 1) % self.args.gradient_accumulation_steps == 0:
                torch.nn.utils.clip_grad_norm_(self.trainable_parameters, self.args.max_grad_norm)
                self.optimizer.step()
                self.optimizer.zero_grad()
    if (grpo_step + 1) % self.args.eval_interval == 0:
        self.policy.eval()
        eval_metrics = self.evaluate(output_dir=self.output_dir / f"checkpoint-step-{grpo_step + 1}")
        self._log(eval_metrics)
```

实验结果





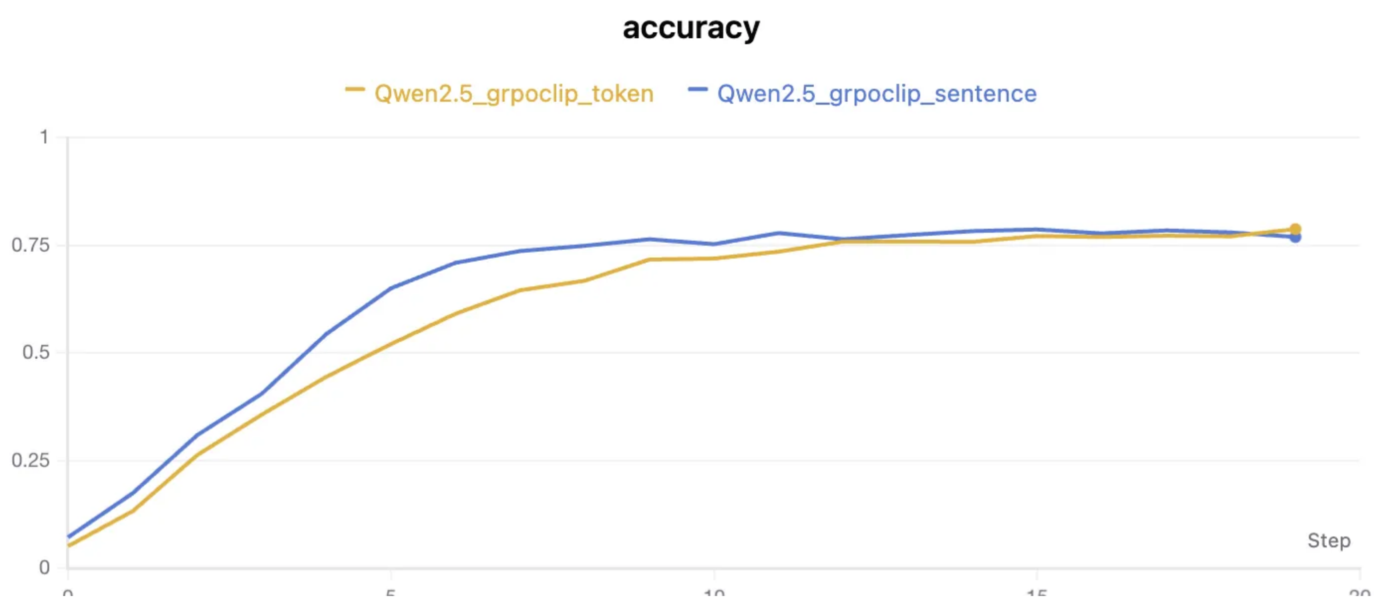
重要性采样—token or sentence

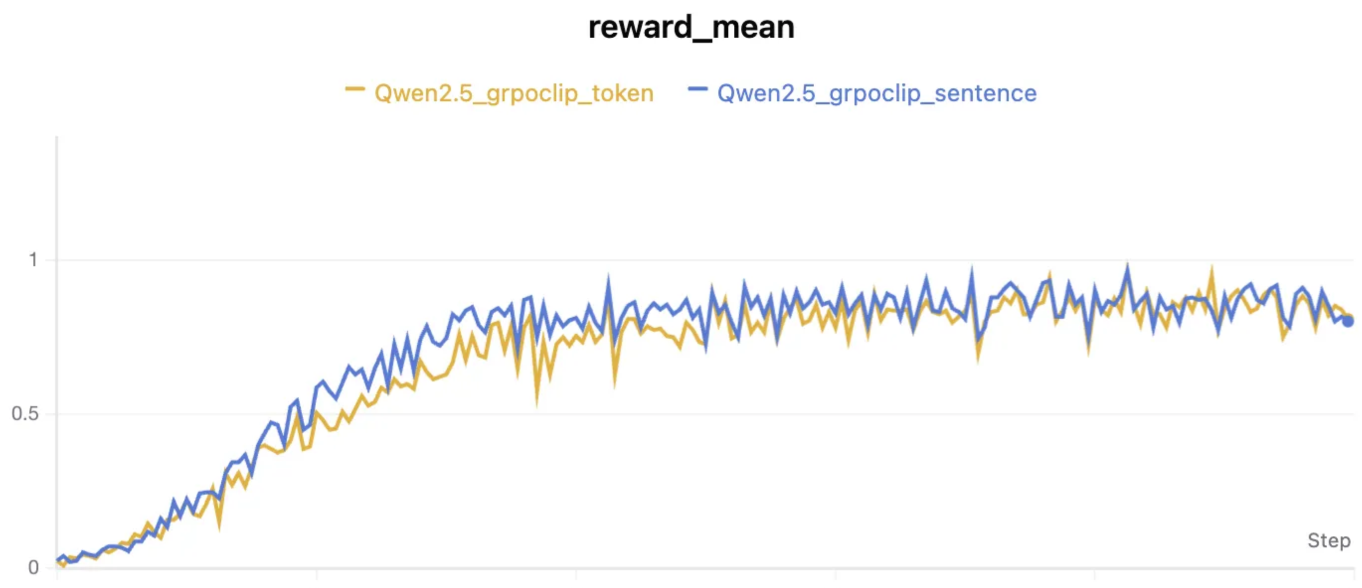
GRPO的重要性采样是token level，即每个token的对数概率都要乘以优势

GSPO则是用每条完整回答的概率（sequence likelihood）来做重要性采样

在实现的时候，在计算log-probabilities的时候就进行mask_mean，这样就得到了完整回答的概率

对比token level 与 sentence level，实验结果如下：





可以看到sentence级别的重要性采样相比较于token级的重要性采样，收敛会更快一些