LLM Alignment Week 2-3 Progression

Yuanqi Zhao

University of Oxford

1.Aug.2024

catalogue

- Reward Model and the code implementation.
- **PPO review and the code implementation.**
- The whole procedure for a RLHF Project.

PPO Review

Take TRL library PPO_trainer as an example;
The Objective Function to be optimized:

$$L^{\text{CLIP+VF}}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) \right],$$

where:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right],$$

$$L^{VF}(\theta) = \tfrac{1}{2}\mathbb{E} \left[\max(L^{VF1}(\theta), L^{VF2}(\theta)) \right],$$

$$L^{VF1}(\theta) = (V_\theta(s_t) - V_t^{\text{targ}})^2,$$

$$L^{VF2}(\theta) = \left(\text{clip}(V_\theta(s_t), V_t - \epsilon, V_t + \epsilon) - V_t^{\text{targ}} \right)^2,$$

$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$, $\hat{A}_t$ is the Estimated Advantage Function.

Code Implementation for PPO - A Few Details

- LLM log-probability calculation method;
- How to compute Advantage function? State-value function and state function?
- The code implementation;

LLM log-probability calculation method

The main trick is converting the output of the LLM to *token_id*. Then, by using this *token_id*, applying Softmax to the last hidden layer output to get the relevant log probability.

```python
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel

model_name = "gpt2"
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model.eval()

input_text = "Hello, how are you?" # 定义输入文本
input_ids = tokenizer.encode(input_text, return_tensors='pt') # 对输入文本进行编码

with torch.no_grad():
    outputs = model(input_ids) # 获取模型的输出（包括 logits）
logits = outputs.logits

log_probs = torch.nn.functional.log_softmax(logits, dim=-1) # 计算 log 概率

token_log_probs = [log_probs[0, i, token_id].item()
                   for i, token_id in enumerate(input_ids[0])] # 获取每个 token 的 log 概率

# 计算句子的 log 概率
sentence_log_prob = sum(token_log_probs)
```

Figure 1: an easy example

How to compute Advantage function?

We achieve this by adding the additional ValueHead to the last layer of our language model, this ValueHead simply projects the last hidden states onto a scalar to estimate the value of a state.

```
base_model_output = self.pretrained_model(
    input_ids=input_ids,
    attention_mask=attention_mask,
    **kwargs,
)

last_hidden_state = base_model_output.hidden_states[-1]
lm_logits = base_model_output.logits
loss = base_model_output.loss

if last_hidden_state.device != self.v_head.summary.weight.device:
    last_hidden_state = last_hidden_state.to(self.v_head.summary.weight.device)

value = self.v_head(last_hidden_state).squeeze(-1)

# force upcast in fp32 if logits are in half-precision
if lm_logits.dtype != torch.float32:
    lm_logits = lm_logits.float()

if return_past_key_values:
    return (lm_logits, loss, value, base_model_output.past_key_values)
else:
    return (lm_logits, loss, value)
```

Figure 2: Adding the additional ValueHead (See AutoModelForCausalLMWithValueHead in HuggingFace)

```
def forward(self, hidden_states):
    output = self.dropout(hidden_states)

    # For now force upcast in fp32 if needed. Let's keep the
    # output in fp32 for numerical stability.
    if output.dtype != self.summary.weight.dtype:
        output = output.to(self.summary.weight.dtype)

    output = self.summary(output)
    return output
```

Figure 3: Forward Method in ValueHead Class

How to compute Advantage function?

Incorporating a value head into a Language Learning Model (LLM) generally involves adding a fully connected layer to the model's final layer. This layer is designed to approximate the state value function. Additionally, the Generalized Advantage Estimation (GAE) method can be utilized to approximate the Advantage function. This approximation enables us to calculate the value function loss component, L^{VF}, in the PPO loss function.

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$r_t = \mathit{score\_from\_reward\_model} - \beta * KL(\pi_{\theta_{old}} || \pi_\theta)$$

Code for computing Advantage Function

```python
def compute_advantages(
    self,
    values: torch.FloatTensor,
    rewards: torch.FloatTensor,
    mask: torch.FloatTensor,
):
    lastgaelam = 0
    advantages_reversed = []
    gen_len = rewards.shape[-1]

    values = values * mask
    rewards = rewards * mask

    if self.config.whiten_rewards:
        rewards = masked_whiten(rewards, mask, shift_mean=False)

    for t in reversed(range(gen_len)):
        nextvalues = values[:, t + 1] if t < gen_len - 1 else 0.0
        delta = rewards[:, t] + self.config.gamma * nextvalues - values[:, t]
        lastgaelam = delta + self.config.gamma * self.config.lam * lastgaelam
        advantages_reversed.append(lastgaelam)
    advantages = torch.stack(advantages_reversed[::-1]).transpose(0, 1)

    returns = advantages + values
    advantages = masked_whiten(advantages, mask)
    advantages = advantages.detach()
    return values, advantages, returns
```

Figure 4

Code for L^{VF}

$$L^{VF}(\theta) = \frac{1}{2}\mathbb{E}\left[\max(L^{VF1}(\theta), L^{VF2}(\theta))\right],$$

$$L^{VF1}(\theta) = (V_\theta(s_t) - V_t^{\text{targ}})^2,$$

$$L^{VF2}(\theta) = \left(\text{clip}(V_\theta(s_t), V_t - \epsilon, V_t + \epsilon) - V_t^{\text{targ}}\right)^2$$

```
vpredclipped = clip_by_value(
    vpreds,
    values - self.config.cliprange_value,
    values + self.config.cliprange_value,
)

vf_losses1 = (vpreds - returns) ** 2
vf_losses2 = (vpredclipped - returns) ** 2
vf_loss = 0.5 * masked_mean(torch.max(vf_losses1, vf_losses2), mask)
vf_clipfrac = masked_mean(torch.gt(vf_losses2, vf_losses1).float(), mask)
```

Figure 5: code to cumpute L^{VF}

Note: returns (:= values + advantages) is calculated based on a big batch, while vpreds is calculated based on the small batch from the big batch.

Code for computing L^{CLIP}

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right],$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$$

```
ratio = torch.exp(logprobs - old_logprobs) # To compute r_{t}

pg_losses = -advantages * ratio
pg_losses2 = -advantages * torch.clamp(ratio, 1.0 - self.config.cliprange, 1.0 + self.config.cliprange)

pg_loss = masked_mean(torch.max(pg_losses, pg_losses2), mask)
pg_clipfrac = masked_mean(torch.gt(pg_losses2, pg_losses).float(), mask)
```

Figure 6: Code for computing L^{CLIP}

The total loss

from above, we can get the total loss

$$L^{\text{CLIP+VF}}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) \right],$$

Therefore, we can use this to update our parameter:

```
self.model.train()
loss_p, loss_v, train_stats = self.loss(
    old_logprobs, values, logits, vpreds, logprobs, mask, advantages, returns
)
loss = loss_p + loss_v
self.accelerator.backward(loss)
if self.config.max_grad_norm is not None:
    if self.accelerator.sync_gradients:
        self.accelerator.clip_grad_norm_(self.model_params, self.config.max_grad_norm)
self.optimizer.step()
# we call optimizer.zero_grad() every time and let `accelerator` handle accumulation
# see https://huggingface.co/docs/accelerate/usage_guides/gradient_accumulation#the-finished-code
self.optimizer.zero_grad()
```

Figure 7

all in all, one step in PPO

- get values, advantages, returns from a big batch;
- for small batch in the big batch, we get the logprobs, logits, vpreds;
- then train our model using this small batch;
- log the stats;

get values, advantages, returns from a big batch

calculate the values, advantages and returns based on the output of ref model and policy model.

```
with torch.no_grad():
    all_logprobs, logits_or_none, values, masks = self.batched_forward_pass(
        self.model,
        queries,
        responses,
        model_inputs,
        response_masks=response_masks,
        return_logits=full_kl_penalty,
    )
    with self.optional_peft_ctx():
        ref_logprobs, ref_logits_or_none, _, _ = self.batched_forward_pass(
            self.model if self.is_peft_model else self.ref_model,
            queries,
            responses,
            model_inputs,
            return_logits=full_kl_penalty,
        )

    rewards, non_score_reward, kls = self.compute_rewards(scores, all_logprobs, ref_logprobs, masks)
    values, advantages, returns = self.compute_advantages(values, rewards, masks)
```

Figure 8

inside small batch

Then for a small batch in our big batch:

```
# vpreds 为LLM的value head的输出，预测state function
# 注意 此时是对small batch去计算log_prob
logprobs, logits, vpreds, _ = self.batched_forward_pass(
    self.model,
    mini_batch_dict["queries"],
    mini_batch_dict["responses"],
    model_inputs,
    return_logits=True,
)
# 在这其中，预估优势函数的时候，我们用的reward得分，是减去了KL惩罚。
train_stats = self.train_minibatch(
    mini_batch_dict["logprobs"],
    mini_batch_dict["values"],
    logprobs,
    logits,
    vpreds,
    mini_batch_dict["masks"],
    mini_batch_dict["advantages"],
    mini_batch_dict["returns"],
)
```

Figure 9: for simplification purpose only

Reference

- HuggingFace TRL library: https://huggingface.co/docs/trl/en/index