Async Off-Policy GRPO

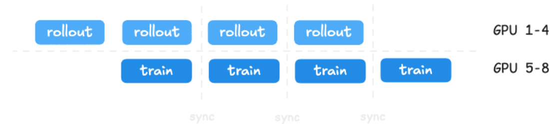
A Minimal Implementation

Xuan Li

June 3, 2026

I. Overview

Async off-policy GRPO overlaps the policy update on the training engine and the trajectory generation on the rollout engine concurrently. The train and rollout engine can be disaggregated to different GPUs for simplicity. This design accelerates post-training but requires meticulous hyperparameter setting to bypass instability.

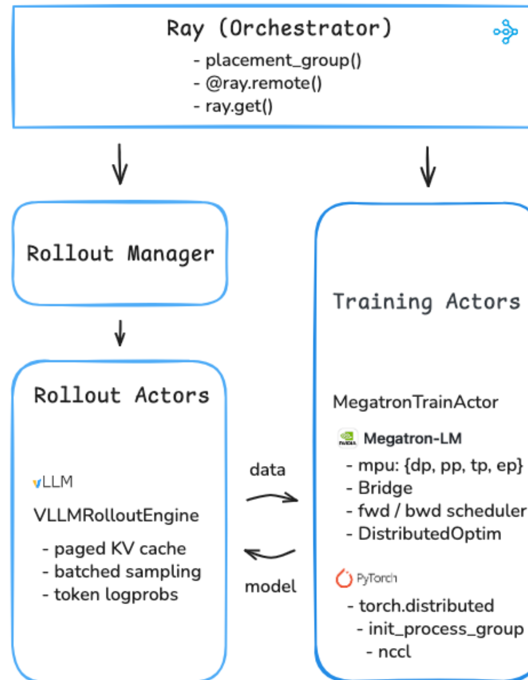


For minimal implementation, it requires:

- `Ray`: GPU placement, process orchestration, and async scheduling
- `Megatron-LM`: backbone for the training engine. It contains parallelism partitioning, forward and backward pass scheduler, distributed optimizer (ZeRO), and bridge to convert model weight from HuggingFace. Besides, `torch.distributed` is used for NCCL communication.
- `vLLM`: backbone for the rollout engine. It produces completions and log probabilities per-token.

II. Development

- Device: L4 GPU on Lightning AI, which has monthly free credits
- VS Code: connect to instance on Lightning with better interface
- Claude Code: plan mode for infra and default mode for implementation & debugging



III. Verification

- code can run: out of GPU memory; appropriate parallelism, model size and seq_len to circumvent OOM
- correctness: rollout engine, training engine, overfit to small dataset (next token prediction); to check the correctness of training engine, try to overfit the model to small dataset for next-token prediction
- stability: small learning rate with warm-ups; to ensure stable fine-tuning, start with learning rate and warm-up steps
- trackable: resume from checkpoint; checkpoint on model weights, optimizer, lr scheduler, and data loader for efficient recovery

IV. Pseudocode

```
# ---- Main ----
def main(config):
    ray.init()

    train_pg   = ray.placement_group([{"GPU": 1} * 4])
    rollout_pg = ray.placement_group([{"GPU": 1} * 4])

    # Rollout actors
    engines = [
        VLLMRolloutEngine.remote(config) for _ in range(4)
    ]
    manager = RolloutManager.remote(engines, datasets, ...)

    # Training actors
    train_actors = [
    MegatronTrainActor.remote(rank, ...) for rank in range(4)
    ]
    ray.get([
        actor.init.remote(config) for actor in train_actors
    ])

    # Async loop
    future = manager.generate.remote(0)
    for epoch in range(epochs):
        batch = ray.get(future)
        future = manager.generate.remote(epoch+1)
        metrics = ray.get([
            a.train_step.remote(batch) for a in train_actors
        ])
```

```
# ---- Training actor ----
@ray.remote(num_gpus=1)
class MegatronTrainActor:
    def __init__(self, rank, ...):

    def init(self, config):
        # NCCL communication across traning rank
        torch.distributed.init_process_group(
            "nccl", world_size=4, rank=self.rank
        )

        # [Megatron] partition TP, DP, etc.
        mpu.initialize_model_parallel(tp_size=4, pp_size=1)

        # [Megatron] bridge model and distributed optim
        self.model, self.optimizer, self.bridge = \
            load_model_with_ddp_and_optimizer(config, tp=4)

    def train_step(self, rollout_batch, rollout_id):
        with torch.no_grads():
            # [Megatron] calls [torch.distributed]
            log_probs_old = compute_log_probs(...)

        advantages = group_advantages(rollout_batch.rewards)

        def forward_step(batch, model):
            logits = model(batch.input_ids)
            def loss_func(logits, log_probs_old, advantages):

        return logits, loss_func

        # [Megatron] fwd/bwd scheduler per micro-batch
        forward_backward_func = get_forward_backward_func()
        forward_backward_func(
            forward_step,
            iter(micro_batches),
            self.model
        )

        # [Megatron] DistributedOptimizer
        self.optimizer.step()
        self.optimizer.zero_grad()

        return metrics
```

```
# ---- Rollout Manager ----
@ray.remote(num_gpus=0)
class RolloutManager:
    def __init__(self, engines, datasets, ...):
        self.engines = engines
        self.dataset = datasets

    def generate(self, rollout_id):
        prompts, gold = self.dataset.next_batch()
        shards = round_robin(prompts, gold)
        futures = [
            engine.generate_batch.remote(s, group_size)
                for engine, s in zip(self.engines, shards)
        ]
        results = ray.get(futures)

        rewards = reward_func(results, gold)
        return RolloutBatch(prompts, results, ...)
```

```
# ---- Rollout Engine Actor ----
@ray.remote(num_gpus=1)
class VLLMRolloutEngine:
    def __init__(self, model_path):
        self.llm = vllm.LLM(model=model_path)

    def generate_batch(self, prompts, group_size):
        params  = SamplingParams(n=group_size, logprobs=1)
        outputs = self.llm.gnerate(prompts, params)
        return [
            {
                "prompt": o.prompt,
                "response": c.text,
                "response_tokens": c.token_ids,
                "log_probs": c.log_probs
            }
            for o in outputs for c in o.outputs
        ]

    def reload_weights(self, checkpoint_path):
```