```
# ====== Data ======
TRAIN_DATA_DIR = "./data/train"
TEST_DATA_DIR = "./data/test"
TRAIN_FRACTION = 1.0

# ====== LoRA ======
RANK = 64
ALPHA = 64.0

# ====== Sharding ======
MESH = [(1, 4), ("fsdp", "tp")]

# ====== GRPO ======
# === Generation during GRPO training ===
MAX_PROMPT_LENGTH = 256
TOTAL_GENERATION_STEPS = 512
# Important to keep a high-ish temperature for varied, diverse responses
during
# training.
TEMPERATURE = 0.9
TOP_P = 1.0
TOP_K = 50
# The number of times the policy generates multiple responses for a given
prompt
# within a single training step. This corresponds to `G` in Algorithm 1
in the
# paper. The "group" in GRPO comes from here.
NUM_GENERATIONS = 4

# === other GRPO configs ===
# The number of iterations per batch (μ in GRPO algo 1).
NUM_ITERATIONS = 1
# The coefficient for the KL divergence penalty (β) in the GRPO loss
function.
# Important to keep a high enough value for this, otherwise, the KL
divergence
# can increase unchecked.
BETA = 0.08
```

```
# Epsilon value for clipping (ε in GRPO loss in paper). Similar to PPO,
for
# stable updates.
EPSILON = 0.2

# ====== Training ======
TRAIN_MICRO_BATCH_SIZE = 2
# Increase `NUM_BATCHES` and `MAX_STEPS` for better results.
NUM_BATCHES = 3738
# Keep `NUM_TEST_BATCHES` low so that evaluation runs quickly. It can be
# increased to a max. of 330 (if batch size is 4).
NUM_TEST_BATCHES = 100

EVAL_EVERY_N_STEPS = 10  # this doesn't matter if `TRAIN_FRACTION = 1.0`.
NUM_EPOCHS = 1  # can potentially train for more epochs

# Number of training steps.
MAX_STEPS = int(NUM_BATCHES * NUM_ITERATIONS * TRAIN_FRACTION *
NUM_EPOCHS)

# === AdamW, warmup, cosine scheduler ===
LEARNING_RATE = 3e-6
B1 = 0.9
B2 = 0.99
WEIGHT_DECAY = 0.1
# == Cosine decay with warmup scheduler ==
# Linearly increase learning rate from 0. to 5e-6 in the first 10%
training
# steps, and then gradually decrease the learning rate to 0 using cosine
# scheduler.
WARMUP_STEPS = 0.1 * MAX_STEPS
# == Grad clipping ==
# Grad clipping to prevent large gradients. Found this
# important to keep KL divergence in check.
MAX_GRAD_NORM = 0.1

# Checkpoint saving
INTERMEDIATE_CKPT_DIR = "/tmp/content/intermediate_ckpt/"
CKPT_DIR = "/tmp/content/ckpts/"
SAVE_INTERVAL_STEPS = 500
MAX_TO_KEEP = 4
```

```
# ====== Inference ======
GENERATION_CONFIGS = {
    # greedy search
    "greedy": {"temperature": 1e-4, "top_k": 1, "top_p": 1.0},
    # some randomness
    "standard": {"temperature": 0.7, "top_k": 50, "top_p": 0.95},
    # liberal
    "liberal": {"temperature": 0.85, "top_k": 2000, "top_p": 1.0},
}
```

这看起来像 **LoRA + FSDP/TP 分片 + GRPO 强化学习式训练** 的一套脚本配置。

---

# 1) Data 数据相关

## `TRAIN_DATA_DIR`, `TEST_DATA_DIR`

- 训练/测试数据所在目录。
- 脚本一般会从这些目录里读 prompt、标签/答案、或者 reward 评估所需信息。

## `TRAIN_FRACTION = 1.0`

- 用训练集的比例。
- `1.0` = 全量训练；`0.1` = 只用 10% 做快速调试。
- 你这里还用它来算 `MAX_STEPS`，所以它不仅影响数据量，也会影响训练总步数。

---

# 2) LoRA 参数（低秩适配）

## `RANK = 64`

- LoRA 的秩（r），决定 LoRA 插入的低秩矩阵容量。
- **更大 rank**：可训练参数更多、表达能力更强、显存和计算更贵，也更可能过拟合小数据。
- 常见取值：8/16/32/64。你这里 64 属于偏大、偏“想要更强适配”的设置。

## `ALPHA = 64.0`

- LoRA 缩放系数（通常实际 scaling = `alpha / rank`）。
- 你这里 `alpha=64, rank=64` → scaling ≈ 1
  含义：LoRA 更新的幅度相对“中性”，不特别放大也不特别缩小。
- **alpha 更大**：LoRA 更新贡献更大（更激进）。
- **alpha 更小**：更新更保守，可能更稳定但学得慢。

> 小经验：很多配置会让 `alpha` 跟 `rank` 同数量级（像你这样），作为稳定起点。

---

# 3) Sharding / 并行分片

## `MESH = [(1, 4), ("fsdp", "tp")]`

这通常表示你在用一个 **设备网格（device mesh）** 来同时做：

- `fsdp`：Fully Sharded Data Parallel（参数/梯度/优化器状态分片）
- `tp`：Tensor Parallel（张量并行，按层内矩阵切分计算）

`(1,4)` 通常意味着网格形状是 1×4（总共 4 张卡），
并把两个维度命名成 `("fsdp","tp")`：

- `fsdp` 维度大小 1（等于没在这个维度上做 FSDP 分片）
- `tp` 维度大小 4（在 4 张卡上做 tensor parallel）

> 也就是说：**你更像是在 4 卡上做 TP，而 FSDP 这维是 1。**
> 当然不同框架（JAX/PJIT、PyTorch DeviceMesh、xFormers）写法略有差异，但直觉上是这个含义。

---

# 4) GRPO（Group Relative Policy Optimization）相关

GRPO 的核心：对同一个 prompt，一次生成多条回答（“group”），用相对优势/排名信号训练策略，同时加 KL 约束别跑太远。

## 4.1 训练时生成（Generation during GRPO training）

## `MAX_PROMPT_LENGTH = 256`

- 输入 prompt 的最大长度（token 数）。
- 超过会截断（可能把关键信息截掉）。
- 如果你任务 prompt 很长（比如长题目/长上下文），256 可能偏小。

**`TOTAL_GENERATION_STEPS = 512`**

- 生成的最大步数（通常是最大生成 token 数）。
- 数值越大：回答更长、计算更贵、训练更慢；但对需要长推理/长输出任务更必要。

**`TEMPERATURE = 0.9`**

- 采样温度，越高越随机、越多样；越低越趋向确定性。
- GRPO 训练阶段通常需要“多样性”来形成组内对比，所以温度偏高是常见做法。

**`TOP_P = 1.0`**

- nucleus sampling，保留累计概率到 p 的候选集合。
- 1.0 基本等于“不用 top-p 限制”。

**`TOP_K = 50`**

- 每一步只在概率最高的 K 个 token 里采样。
- `temperature=0.9` + `top_k=50`：多样但不至于太离谱。
- `top_k` 太大（或 top_p 太大）会引入低质量 token，reward 学习更难。

**`NUM_GENERATIONS = 4`**

- 每个 prompt 生成 4 个候选回答（这就是 “group” 大小 G）。
- G 越大：
  - 组内对比更稳定、信号更好
  - 但生成开销线性增加（训练会慢、显存也更吃紧）
- 4 是比较常见的折中点。

---

## 4.2 其它 GRPO 超参（优化稳定性核心）

**`NUM_ITERATIONS = 1`**

- 每个 batch 做几次策略更新（论文里 μ）。
- 越大：同一批数据反复更新，样本效率高一点，但更容易过拟合/不稳定/KL 爆掉。
- 1 通常最稳。

`BETA = 0.08`

- KL penalty 系数（β），用于约束新策略别偏离参考策略太多。
- **β 太小**：更新会跑飞，KL 变大、模型行为崩坏（胡说/模式坍塌）。
- **β 太大**：模型几乎不敢动，学得很慢。
- 0.08 属于"偏稳健"的取值。

`EPSILON = 0.2`

- clipping 系数（类似 PPO 的 clip range）。
- **更小**：更保守、更稳；可能学得慢。
- **更大**：更激进；可能不稳。
- 0.2 是 PPO/GRPO 里很经典的默认。

---

# 5) Training 训练循环参数

`TRAIN_MICRO_BATCH_SIZE = 2`

- 单卡（或单步）微批大小。
- 如果你有梯度累积（没写出来但很多脚本有），有效 batch size = micro_batch × grad_accum × data_parallel_world_size。
- micro batch 越大：显存越吃紧；但梯度估计噪声更小。

`NUM_BATCHES = 3738`

- 每个 epoch 的 batch 数（或训练数据被切成多少 batch）。
- 这个值跟数据大小、batch size、dataloader 实现有关。

`NUM_TEST_BATCHES = 100`

- 每次评估跑多少个测试 batch。

- 越大评估越准但越慢；你这里是“为了快”。

### `EVAL_EVERY_N_STEPS = 10`

- 每 10 个 step eval 一次。你注释说 `TRAIN_FRACTION=1.0` 时它“不重要”，可能是脚本只在 fraction < 1 时才做频繁 eval。

### `NUM_EPOCHS = 1`

- 训练轮数。更多 epoch 可能更好，但 RL 类训练也更容易 overfit reward 或出现 reward hacking。

### `MAX_STEPS = int(NUM_BATCHES * NUM_ITERATIONS * TRAIN_FRACTION * NUM_EPOCHS)`

- 总训练步数计算方式。
- 你现在是 `3738 * 1 * 1.0 * 1 = 3738` steps。

---

## 6) AdamW + 学习率调度 + 梯度裁剪

### `LEARNING_RATE = 3e-6`

- 强化学习式微调（尤其带 KL）常用非常小 LR。
- 太大：KL 会暴涨、不稳定；太小：学不动。

### `B1 = 0.9`, `B2 = 0.99`

- Adam 的 β1/β2 动量参数。
- β2=0.99 比常见的 0.999 更“反应快”一点（对梯度方差变化更敏感），有时在 RL 微调里更好控。

### `WEIGHT_DECAY = 0.1`

- 权重衰减（AdamW 的正则）。
- 0.1 其实不小；但如果你主要训练 LoRA 参数，影响范围可能有限（取决于实现是否对 LoRA 参数也做 decay）。

### `WARMUP_STEPS = 0.1 * MAX_STEPS`

- 线性预热是一种学习率调度策略，在训练开始阶段**逐步增加学习率**，而不是一开始就使用目标学习率。
- 前 10% steps 线性 warmup（学习率从 0 拉到峰值），后面 cosine decay 到 0。
- 现在 `MAX_STEPS=3738` → warmup ≈ 373.8 steps（一般会取整）。
- warmup 对稳定性很重要，尤其 RL 微调初期。

### `MAX_GRAD_NORM = 0.1`

- 梯度裁剪上限（L2 norm）。
- 0.1 非常保守，目的是防止某些 step 梯度爆炸导致 KL 失控。
- 如果你发现模型几乎不学习，有时也可能是 clip 太小导致更新幅度被压得过狠（但 RL 里宁稳勿躁）。

---

## 7) Checkpoint 保存

### `INTERMEDIATE_CKPT_DIR`, `CKPT_DIR`

- 中间检查点目录 vs 最终/常规目录（具体看脚本怎么用）。

### `SAVE_INTERVAL_STEPS = 500`

- 每 500 step 存一次。你总步数 3738，大概会存 7 次左右。

### `MAX_TO_KEEP = 4`

- 最多保留 4 个 checkpoint，旧的会被删（节省磁盘）。

---

## 8) Inference 推理配置

`GENERATION_CONFIGS` 是推理时的三套采样策略：

### `"greedy": {"temperature": 1e-4, "top_k": 1, "top_p": 1.0}`

- 近似贪心：几乎完全确定性输出。
- 用于可复现评估、或者你想看“模型最确定答案”。

`"standard": {"temperature": 0.7, "top_k": 50, "top_p": 0.95}`

- 典型聊天/生成配置：质量和多样性平衡。

`"liberal": {"temperature": 0.85, "top_k": 2000, "top_p": 1.0}`

- 更发散、更随机：更容易有创意，但也更容易跑偏/胡说。
- `top_k=2000` 很大，基本放开了候选集合。

---

# 最后给你一个“怎么调参”的直觉小抄（很实用）

- **KL 失控 / 模型变胡说**：先加大 `BETA`，再减小 `LEARNING_RATE`，必要时减小 `MAX_GRAD_NORM` 或 `EPSILON`。
- **训练没效果 / reward 不涨**：稍微减小 `BETA` 或增大 `LEARNING_RATE`；或者把 `NUM_GENERATIONS` 增大一点改善组内对比信号。
- **生成太单一，组内差异小**：提高 `TEMPERATURE` 或放宽 `TOP_K / TOP_P`（但别太过）。
- **显存爆 / 太慢**：减小 `TOTAL_GENERATION_STEPS`、`NUM_GENERATIONS`、`RANK`，或降低 `MAX_PROMPT_LENGTH`。

# Data preprocessing

First, let's define some special tokens. We instruct the model to first reason between the `<reasoning>` and `</reasoning>` tokens. After reasoning, we expect it to provide the answer between the `<answer>` and `</answer>` tokens.

```
reasoning_start = "<reasoning>"
reasoning_end = "</reasoning>"
```

```
solution_start = "<answer>"
solution_end = "</answer>"


SYSTEM_PROMPT = f"""You are given a problem. Think about the problem and
\
provide your reasoning. Place it between {reasoning_start} and \
{reasoning_end}. Then, provide the final answer (i.e., just one numerical
\
value) between {solution_start} and {solution_end}."""

TEMPLATE = """<start_of_turn>user
{system_prompt}

{question}<end_of_turn>
<start_of_turn>model"""
```

We use OpenAI's GSM8K dataset, which comprises grade school math word problems.

```
def extract_hash_answer(text: str) -> str | None:
  if "####" not in text:
    return None
  return text.split("####")[1].strip()


def _load_from_tfds(data_dir: str, split: str):
  import tensorflow_datasets.text.gsm8k
  return tfds.data_source(
      "gsm8k",
      split=split,
      data_dir=data_dir,
      builder_kwargs={"file_format": tfds.core.FileFormat.ARRAY_RECORD},
      download=True,
  )


def download_kaggle_dataset(target_dir="./data/gsm8k"):
  os.makedirs(target_dir, exist_ok=True)
  src = kagglehub.dataset_download("thedevastator/grade-school-math-8k-q-
a")
```

```
  src = Path(src)
  dst = Path(target_dir)

  for csv_file in src.glob("*.csv"):  # match all CSV files
    shutil.copy2(csv_file, dst / csv_file.name)
    print(f"Copied {csv_file.name} → {dst/csv_file.name}")
  return target_dir


def get_dataset(data_dir, split="train", source="tfds") ->
grain.MapDataset:
  # Download data
  if not os.path.exists(data_dir):
    os.makedirs(data_dir)

  if source == "tfds":
    import tensorflow_datasets.text.gsm8k
    data = tfds.data_source(
        "gsm8k",
        split=split,
        data_dir=data_dir,
        builder_kwargs={"file_format":
tfds.core.FileFormat.ARRAY_RECORD},
        download=True,
    )

  elif source == "kaggle":
    kaggle_dir = download_kaggle_dataset(data_dir)
    file_name = "main_" + split + ".csv"
    csv_path = os.path.join(kaggle_dir, file_name)  # adjust filename if
needed

    data = []
    with open(csv_path, newline="", encoding="utf-8") as csvfile:
      reader = csv.DictReader(csvfile)
      for row in reader:
        data.append({
            "question": row["question"],
            "answer": row["answer"],
        })
```

```
  else:
    raise ValueError(f"Unknown source: {source}")

  def _as_text(v):
    return v if isinstance(v, str) else v.decode("utf-8")

  dataset = (
      grain.MapDataset.source(data)
      .shuffle(seed=42)
      .map(
          lambda x: {
              # passed to model forward pass
              "prompts": TEMPLATE.format(
                  system_prompt=SYSTEM_PROMPT,
                  question=_as_text(x["question"]),
              ),
              # passed to reward functions
              "question": _as_text(x["question"]),
              # passed to reward functions
              "answer": extract_hash_answer(_as_text(x["answer"])),
          }
      )
  )
  return dataset
```

# 一、`extract_hash_answer`：提取 GSM8K 的最终答案

```
def extract_hash_answer(text: str) -> str | None:
  if "####" not in text:
    return None
  return text.split("####")[1].strip()
```

## 背景

GSM8K 的 `answer` 格式通常是：

```
Let’s solve step by step.
...
Therefore the answer is #### 42
```

也就是说：

- 推理过程（chain-of-thought）在前
- **最终答案统一写在 `####` 后面**

`_load_from_tfds`：从 TFDS 加载 GSM8K（未使用）

`download_kaggle_dataset`：从 Kaggle 下载 GSM8K

**数据来源分支（TFDS vs Kaggle）**

✅ 情况 1：`source="tfds"`

```
data = tfds.data_source(
    "gsm8k",
    split=split,
    data_dir=data_dir,
    ...
)
```

- 使用 **官方 TFDS GSM8K**
- `split` 是 `"train"` 或 `"test"`

📌 这时 `data` 是：

> **TFDS 的 iterable dataset**

---

✅ 情况 2：`source="kaggle"`

```
kaggle_dir = download_kaggle_dataset(data_dir)
file_name = "main_" + split + ".csv"
```

- 自动下载 Kaggle 数据
- 选择 `main_train.csv` 或 `main_test.csv`

然后：

```
data = []
for row in reader:
    data.append({
        "question": row["question"],
        "answer": row["answer"],
    })
```

📌 这时 `data` 是：

```
[
  {"question": "...", "answer": "...#### 42"},
  {"question": "...", "answer": "...#### 17"},
  ...
]
```

# Grain Dataset 变换链（重点）

```
dataset = (
    grain.MapDataset.source(data)
    .shuffle(seed=42)
    .map(lambda x: {...})
)
```

## 1️⃣ `grain.MapDataset.source(data)`

- 把 list / tfds 数据
- 包装成 **Grain Dataset（Google 训练框架）**

---

## 2️⃣ `.shuffle(seed=42)`

- 打乱顺序
- **RL 训练非常重要**（防止模式坍塌）

---

## 3️⃣ `.map(...)`：构造训练样本

```
lambda x: {
    "prompts": TEMPLATE.format(
        system_prompt=SYSTEM_PROMPT,
        question=_as_text(x["question"]),
    ),
    "question": _as_text(x["question"]),
    "answer": extract_hash_answer(_as_text(x["answer"])),
}
```

### 字段含义非常关键：

| key | 用途 |
| --- | --- |
| `prompts` | **喂给模型生成答案** |
| `question` | **奖励函数用** |
| `answer` | **ground truth（最终数值）** |

📌 `TEMPLATE` 一般长这样：

```
<System>
{system_prompt}
</System>

<User>
{question}
</User>
```

---

# 八、最终返回的数据长什么样?

**每个 sample ≈**

```
{
  "prompts": "<SYSTEM>...你是数学专家...</SYSTEM>\n<User>Tom has 3
apples...</User>",
  "question": "Tom has 3 apples...",
  "answer": "42"
}
```

👉 正好满足：

- **生成模型 forward**
- **GRPO reward 计算**
- **KL penalty / accuracy reward**

# 加载策略模型（Policy Model）与参考模型（Reference Model）

**策略模型（policy model）** 是实际参与训练、其参数会被更新的模型。
**参考模型（reference model）** 用于计算 KL 散度（KL divergence），以确保策略模型的更新幅度不会过大，从而避免其行为与参考模型发生过度偏离。

通常情况下，**参考模型** 是基础模型（base model），而 **策略模型** 则是在同一基础模型之上引入了 **LoRA 参数** 的模型。在训练过程中，**仅更新 LoRA 参数**，基础模型参数保持不变。

**注意**：我们在这里使用的是**全精度（fp32）训练**。当然，你也可以借助 **Qwix** 来进行 **量化感知训练（QAT）**。

## Reference Model（参考模型）

- 从这个 checkpoint **加载**
- **不加 LoRA**
- **不更新参数**
- 只用来算 **KL divergence**

👉 它的本质是：Reference Model = 冻结的 Base Gemma

## Policy Model（策略模型）

- 从**同一个 checkpoint**加载
- **加 LoRA**
- **只更新 LoRA 参数**
- 参与 GRPO / PPO 训练

👉 它的本质是：Policy Model = Base Gemma + LoRA（可训练）

```
!rm /tmp/content/intermediate_ckpt/* -rf

!rm /tmp/content/ckpts/* -rf

if model_family == "gemma2":
  params = params_lib.load_and_format_params(
      os.path.join(kaggle_ckpt_path, "gemma2-2b-it")
  )
  gemma = gemma_lib.Transformer.from_params(params, version="2-2b-it")
  checkpointer = ocp.StandardCheckpointer()
  _, state = nnx.split(gemma)
  checkpointer.save(os.path.join(INTERMEDIATE_CKPT_DIR, "state"), state)
  checkpointer.wait_until_finished()
  # Delete the intermediate model to save memory.
  del params
  del gemma
  del state
  gc.collect()
```

> **Checkpoint = 给模型“存档”**

就像你玩游戏：

- 打 Boss 前存个档
- 打到一半崩了，可以从存档继续
- 想回到某个关键节点重新走一遍

**模型训练里的 checkpoint 就是干这个的。**

---

### 二、技术一点：Checkpoint 到底存了什么？

一个 checkpoint 通常包含：

1. **模型参数（weights / state）**
    - 每一层的权重、偏置
2. **（有时）优化器状态**
    - Adam 的动量、二阶矩等

3. **（有时）训练进度**
    - step / epoch / 随机种子等

👉 **有了 checkpoint，就能"无损恢复"模型当时的状态。**

---

## 三、没有 checkpoint 会发生什么？😱

**场景 1：训练中断**

- Kaggle / Colab 掉线
- OOM
- 内核崩了

❌ **没有 checkpoint = 从零开始训练**

**场景 2：你想复现实验**

- "第 10k step 的模型效果最好"
- 但你没存

❌ **回不去了**

```
checkpointer.save(os.path.join(INTERMEDIATE_CKPT_DIR, "state"), state)
```

**这里发生了什么？**

### 1️⃣ 你刚加载的是 Gemma2-2B-IT 的"原始状态"

```
params → gemma → state
```

这一步拿到的是：

> **"还没训练、还没加 LoRA 的 base model 状态"**

---

### 2️⃣ 把这个状态存成 checkpoint

```
INTERMEDIATE_CKPT_DIR/state
```

这相当于：

> ✅ 给 **reference model / policy model 的初始版本**
> 打了一个“永远不会变的存档”

## 省内存（

你注意到这一步了吗?

```
del params
del gemma
del state
gc.collect()
```

逻辑是:

> **“我已经把模型安全存盘了，现在把内存全清空”**

Kaggle 上:

- 2B 模型 = 几 GB
- 不清内存 = 直接 OOM

# 下面加载参考模型和策略模型

## Model Loading and LoRA Application

These two functions work together to load a base model from a checkpoint and apply a LoRA (Low-Rank Adaptation) layer to it.

- `get_ref_model`: Loads the complete Gemma model from a specified checkpoint path. It uses JAX sharding to distribute the model parameters across multiple devices.
- `get_lora_model`: Takes the base model and applies LoRA layers to it. It uses a `LoraProvider` to select specific layers (like attention and MLP layers) to be adapted. The resulting LoRA-infused model is then sharded and updated to ensure it's ready for distributed training.

### JAX 是什么?

JAX 是 Google 开发的一个高性能数值计算库，可以理解为"可以自动求导的 NumPy + GPU/TPU 加速"。

```
# 普通 NumPy
import numpy as np
x = np.array([1, 2, 3])

# JAX（用法类似，但支持GPU加速和自动求导）
import jax.numpy as jnp
x = jnp.array([1, 2, 3])
```

**JAX 的主要特点：**

- 自动求导（训练神经网络需要）
- 自动并行化到多个 GPU/TPU
- JIT 编译加速

## NNX 是什么？

NNX 是 JAX 的一个神经网络库（类似 PyTorch 的 nn.Module），用于构建和管理神经网络模型。

```
# NNX 定义模型的方式
class MyModel(nnx.Module):
    def __init__(self):
        self.dense = nnx.Linear(10, 5)

    def __call__(self, x):
        return self.dense(x)
```

## Mesh（网格）是什么？

这是最关键的概念！当你的模型太大（比如 Gemma 2B 有 20 亿参数），一个 GPU 放不下时，需要把模型**分割到多个设备**上。

**Mesh 就是定义"设备的逻辑布局"**：

```
# 假设你有 8 个 GPU
mesh = jax.make_mesh((2, 4), ('data', 'model'))
# 这表示：把 8 个 GPU 排列成 2x4 的网格
# - 'data' 维度有 2 个设备（用于数据并行）
# - 'model' 维度有 4 个设备（用于模型并行）
```

**形象比喻：**

- 想象你有一张巨大的 Excel 表格（模型参数）
- 一台电脑（GPU）屏幕太小显示不下
- Mesh 就是规划"用几台电脑、怎么排列、每台显示哪部分"

## 逐行解释代码：

```
def get_gemma_ref_model(ckpt_path):
  #网格布局
  mesh = jax.make_mesh(*MESH) # MESH 可能是 ((8,), ('devices',)) 。这意味着用 8 个设备，维度名叫 'devices'

  #获取模型配置model_config
  model_config = gemma_lib.ModelConfig.gemma2_2b() # 这是 Gemma 2B 模型的超参数配置.比如：层数、隐藏维度、注意力头数等

  #创建"抽象模型"（只有形状，没有实际数据）
  abs_gemma: nnx.Module = nnx.eval_shape(
      lambda: gemma_lib.Transformer(model_config, rngs=nnx.Rngs(params=0))
  ) # eval_shape：不实际创建参数，只计算形状，就像"画设计图"而不是"盖房子"

  # 提取抽象参数的状态
  abs_state = nnx.state(abs_gemma)# state 包含所有参数的"占位符"，比如：{'layer1.weight': ShapeInfo, 'layer2.bias': ShapeInfo}

  # 为每个参数添加分片（sharding）信息：比如这个参数应该切成几块，放到哪些设备上
  abs_state = jax.tree.map(
      lambda a, s: jax.ShapeDtypeStruct(a.shape, jnp.bfloat16, sharding=s),
      abs_state,
      nnx.get_named_sharding(abs_state, mesh),
```

```
  )

  # 从检查点恢复参数
  checkpointer = ocp.StandardCheckpointer()
  restored_params = checkpointer.restore(ckpt_path, target=abs_state)

  # 重组模型（split+merge）
  graph_def, _ = nnx.split(abs_gemma)
  gemma = nnx.merge(graph_def, restored_params)
  return gemma, mesh, model_config
```

**Sharding（分片）示例：**

```
# 假设有个 [8, 1024] 的参数矩阵
# Sharding 可以指定：
# - 沿第 0 维切分到 2 个设备 → 每个设备存 [4, 1024]
# - 沿第 1 维切分到 4 个设备 → 每个设备存 [8, 256]
```

## `jax.tree.map` 是什么？

这是 JAX 的"树遍历工具"。神经网络的参数是嵌套字典结构（树状）：

```
# abs_state 的结构示例
abs_state = {
    'layer1': {
        'weight': Array[shape=(1024, 512)],
        'bias': Array[shape=(512,)]
    },
    'layer2': {
        'weight': Array[shape=(512, 256)],
        'bias': Array[shape=(256,)]
    }
}
```

`jax.tree.map(func, tree1, tree2)` 会：

- **同时遍历** `tree1` 和 `tree2` 的每个叶子节点
- 对每对叶子节点调用 `func`

```
# 例子
tree1 = {'a': 1, 'b': {'c': 2}}
tree2 = {'a': 10, 'b': {'c': 20}}
result = jax.tree.map(lambda x, y: x + y, tree1, tree2)
# 结果: {'a': 11, 'b': {'c': 22}}
```

## `nnx.get_named_sharding(abs_state, mesh)` 做什么?

这个函数**自动推断**每个参数应该如何分片。

**输入:**

- `abs_state`: 参数的形状信息
- `mesh`: 设备网格布局

**输出:** 每个参数的分片策略（Sharding）

**示例:**

```
# 假设 mesh 定义
mesh = jax.make_mesh((2, 4), ('data', 'model'))
# 2 个设备用于数据并行，4 个设备用于模型并行

# 对于一个权重矩阵 [1024, 512]
sharding = NamedSharding(
    mesh,
    PartitionSpec('model', None)  # 沿第0维切分到'model'维度的4个设备
)
# 结果：这个矩阵被切成 4 块，每块 [256, 512]
```

**分片策略的几种常见模式:**

| 参数形状 | PartitionSpec | 含义 |
| --- | --- | --- |
| `[1024, 512]` | `('model', None)` | 按行切分到4个设备，每个设备存 `[256, 512]` |
| `[1024, 512]` | `(None, 'model')` | 按列切分到4个设备，每个设备存 `[1024, 128]` |
| `[1024, 512]` | `('data', 'model')` | 二维切分到 2x4=8 个设备，每个设备存 `[512, 128]` |
| `[512]` | `(None,)` | 不切分，每个设备都存完整的 `[512]` |

## Lambda 函数做什么?

```
lambda a, s: jax.ShapeDtypeStruct(a.shape, jnp.bfloat16, sharding=s)
```

**参数:**

- `a`: 当前参数的抽象信息（来自 `abs_state`）
- `s`: 当前参数的分片策略（来自 `get_named_sharding`）

**返回:** `ShapeDtypeStruct` —— 一个"参数描述符"，包含:

- `shape`: 参数形状（如 `[1024, 512]`）
- `dtype`: 数据类型（统一改为 `bfloat16` 节省显存）
- `sharding`: 分片策略（告诉 JAX 如何分布到设备）

**完整例子:**

```
# 假设某个权重的信息
a = ShapeDtypeStruct(shape=(1024, 512), dtype=float32)
s = NamedSharding(mesh, PartitionSpec('model', None))

# Lambda 函数处理后
result = ShapeDtypeStruct(
    shape=(1024, 512),
    dtype=bfloat16,      # 从 float32 改为 bfloat16
    sharding=s           # 添加分片信息
```

```
)
```

---

### **整体流程可视化**
```
原始 abs_state:
├── layer1.weight: ShapeDtypeStruct(shape=[1024,512], dtype=float32, sharding=None)
└── layer1.bias:   ShapeDtypeStruct(shape=[512], dtype=float32, sharding=None)

↓ nnx.get_named_sharding 推断分片策略

分片策略:
├── layer1.weight: NamedSharding(PartitionSpec('model', None))
└── layer1.bias:   NamedSharding(PartitionSpec(None))

↓ jax.tree.map 逐个处理

新的 abs_state:
├── layer1.weight: ShapeDtypeStruct(shape=[1024,512], dtype=bfloat16, sharding=按行切4份)
└── layer1.bias:   ShapeDtypeStruct(shape=[512], dtype=bfloat16, sharding=不切分)
```

## 重组模型

```
graph_def, _ = nnx.split(abs_gemma)
gemma = nnx.merge(graph_def, restored_params)
```

## 为什么要 split 和 merge?

在 NNX 中，模型由两部分组成：

1. **结构（graph_def）**：计算图、层的连接关系（不包含参数值）
2. **参数（state）**：实际的权重、偏置数据

**类比：**

- `graph_def`：汽车的设计图纸（引擎连接传动系统，传动系统连接轮子）
- `state`：真实的零件（实际的引擎、轮子）

---

## `nnx.split` 做什么？

```
graph_def, _ = nnx.split(abs_gemma)
```

**输入：** `abs_gemma` —— 抽象模型（只有形状，没有实际数据）

**输出：**

- `graph_def`：模型的计算图（层的定义、连接关系）
- `_`：模型的参数状态（这里被忽略了，因为 `abs_gemma` 本来就没真实数据）

**为什么要这样做？** 因为我们需要**模型的结构**，但要用从检查点加载的**真实参数**。

---

## `nnx.merge` 做什么？

```
gemma = nnx.merge(graph_def, restored_params)
```

**输入：**

- `graph_def`：模型结构
- `restored_params`：从检查点恢复的真实参数（已经分布到多个设备）

**输出：** 完整的、可运行的模型

### 为什么不直接加载模型？

**传统方式（单GPU）：**

```
# 简单但不适合大模型
model = load_model(ckpt_path)  # 全部加载到一个 GPU
```

**问题：**

- 如果模型 140GB，单个 GPU 只有 80GB 显存 → **爆显存**

**这个方案的优势：**

1. **先规划后加载**：知道每个参数去哪儿，再分批加载
2. **自动分片**：不需要手动切分数据
3. **透明并行**：后续计算时，JAX 自动处理跨设备通信

```
def get_lora_model(base_model, mesh):
  lora_provider = qwix.LoraProvider(
      module_path=(
          ".*q_einsum|.*kv_einsum|.*gate_proj|.*down_proj|.*up_proj|"
          ".*attn_vec_einsum"
      ),
      rank=RANK,
      alpha=ALPHA,
  )

  model_input = base_model.get_model_input()
  lora_model = qwix.apply_lora_to_model(
      base_model, lora_provider, **model_input
  )

  with mesh:
    state = nnx.state(lora_model)
    pspecs = nnx.get_partition_spec(state)
    sharded_state = jax.lax.with_sharding_constraint(state, pspecs)
    nnx.update(lora_model, sharded_state)

  return lora_model
```

这段代码是在基础模型上添加 **LoRA（Low-Rank Adaptation）** 微调层。我来逐步解释。

# 什么是 LoRA?

# 问题：大模型微调太贵

假设 Gemma 2B 有 20 亿参数：

- 全参数微调：需要更新全部 20 亿参数 → 需要大量显存和计算
- 成本高、时间长

# LoRA 的解决方案

**核心思想：** 不直接修改原始权重，而是添加小的"调整层"

**数学原理：**

```
# 原始的全参数微调
W_new = W_original + ΔW  # ΔW 是 [1024, 1024] 的矩阵

# LoRA 的做法
ΔW ≈ A @ B  # A: [1024, 8], B: [8, 1024]
# 参数量：1024×1024 = 1M  →  1024×8 + 8×1024 = 16K
# 减少了 98% 的参数！
```

**形象比喻：**

- 全参数微调：重新装修整栋房子
- LoRA：只加几个装饰品和家具（效果接近，成本极低）

# 代码解读

## 1.创建 LoRA 配置

```
lora_provider = qwix.LoraProvider(
    module_path=(
        ".*q_einsum|.*kv_einsum|.*gate_proj|.*down_proj|.*up_proj|"
        ".*attn_vec_einsum"
    ),
    rank=RANK,
    alpha=ALPHA,
)
```

## 参数详解

### 1. `module_path`（正则表达式）

指定**哪些层**需要添加 LoRA。

```
".*q_einsum|.*kv_einsum|.*gate_proj|.*down_proj|.*up_proj|.*attn_vec_einsum"
```

这是一个正则表达式，匹配模型中特定的层名：

| 模式 | 匹配的层 | 作用 |
|---|---|---|
| `.*q_einsum` | 查询矩阵（Query）投影 | 注意力机制的一部分 |
| `.*kv_einsum` | 键值矩阵（Key/Value）投影 | 注意力机制的一部分 |
| `.*attn_vec_einsum` | 注意力输出投影 | 注意力机制的一部分 |
| `.*gate_proj` | 门控投影 | FFN（前馈网络）的门控 |
| `.*up_proj` | 上投影 | FFN 的升维操作 |
| `.*down_proj` | 下投影 | FFN 的降维操作 |

**为什么选这些层？**
- 这些是 Transformer 中**最关键的权重矩阵**
- 研究表明：只微调这些层就能达到很好效果
- 其他层（如 LayerNorm）的参数很少，不需要 LoRA

**形象理解：**
```
Transformer 层结构：
├── 自注意力模块
│   ├── q_einsum  ← 添加 LoRA
│   ├── kv_einsum ← 添加 LoRA
│   └── attn_vec_einsum ← 添加 LoRA
├── LayerNorm（不添加 LoRA）
└── FFN 模块
    ├── gate_proj ← 添加 LoRA
    ├── up_proj   ← 添加 LoRA
    └── down_proj ← 添加 LoRA
```

---

### 2. `rank=RANK`（秩）

**决定 LoRA 矩阵的"中间维度"大小。**

```
# 假设原始权重是 [1024, 1024]
# rank=8 时：
A = [1024, 8]   # 8 是 rank
B = [8, 1024]   # 8 是 rank
ΔW = A @ B  # 结果仍是 [1024, 1024]

# 参数量对比
原始全参数微调：1024 × 1024 = 1,048,576
LoRA (rank=8):  1024×8 + 8×1024 = 16,384  (减少 98.4%)
LoRA (rank=16): 1024×16 + 16×1024 = 32,768 (减少 96.9%)
```

**rank 的选择：**

- **rank 越大**：表达能力越强，但参数越多
- **rank 越小**：参数越少，但可能不够表达复杂任务
- **常用值**：4, 8, 16, 32（根据任务复杂度选择）

---

### 3. `alpha=ALPHA`（缩放因子）

控制 LoRA 调整的"强度"。

```
# 前向传播时的实际计算
output = W_original @ x + (alpha / rank) * (B @ (A @ x))
#         原始输出       +    LoRA 的贡献
```

**为什么需要 alpha?**

- 如果 `rank` 很小，LoRA 的贡献可能太弱
- `alpha` 用来放大/缩小 LoRA 的影响
- **常用值**：通常设为 `alpha = rank`（保持稳定）

**例子：**

```
rank = 8, alpha = 8   → 缩放系数 = 8/8 = 1.0
rank = 8, alpha = 16  → 缩放系数 = 16/8 = 2.0  (LoRA 影响更大)
rank = 16, alpha = 8  → 缩放系数 = 8/16 = 0.5  (LoRA 影响减半)
```

---

## 2.应用 LoRA 到模型

```
model_input = base_model.get_model_input()
lora_model = qwix.apply_lora_to_model(
    base_model, lora_provider, **model_input
)
```

### `get_model_input()` 做什么?

获取模型的输入规格（形状、数据类型等），用于初始化 LoRA 层。

```
# 可能返回类似这样的字典
model_input = {
    'input_shape': (batch_size, seq_len),
    'dtype': jnp.bfloat16,
    ...
}
```

### `apply_lora_to_model` 做什么?

遍历模型的所有层，对匹配 `module_path` 的层添加 LoRA 适配器。

**内部逻辑（伪代码）：**

```
for layer_name, layer in base_model.layers:
    if re.match(lora_provider.module_path, layer_name):
        # 匹配成功，添加 LoRA
        original_weight = layer.weight  # [1024, 1024]

        # 创建两个小矩阵
        lora_A = initialize([1024, rank])  # [1024, 8]
        lora_B = initialize([rank, 1024])  # [8, 1024]

        # 修改层的前向传播
        def new_forward(x):
            return original_weight @ x + (alpha/rank) * lora_B @ (lora_A
@ x)

        layer.forward = new_forward
```

**结果：**

- `lora_model` 和 `base_model` 结构相同
- 但在关键层添加了可训练的 LoRA 参数
- **原始参数冻结**（不更新），只训练 LoRA 部分

---

## 3. 分片 LoRA 参数

```
with mesh:
    state = nnx.state(lora_model)
    pspecs = nnx.get_partition_spec(state)
    sharded_state = jax.lax.with_sharding_constraint(state, pspecs)
    nnx.update(lora_model, sharded_state)
```

## 为什么需要这一步？

虽然 LoRA 参数很小，但仍需要正确分布到多个设备上，以便：

1. 与基础模型的分片策略一致
2. 训练时能高效并行

---

## 逐行解释

### 1. `with mesh:`

进入设备网格的上下文，后续操作会自动考虑设备分布。

```
# mesh 定义了设备布局，比如
mesh = jax.make_mesh((2, 4), ('data', 'model'))
```

---

### 2. `state = nnx.state(lora_model)`

提取 LoRA 模型的所有参数状态。

```
state = {
    'layer1.q_einsum.lora_A': Array[1024, 8],
    'layer1.q_einsum.lora_B': Array[8, 1024],
    'layer1.kv_einsum.lora_A': Array[1024, 8],
    ...
    # 注意：原始参数（如 W_original）不在这里，它们是冻结的
}
```

---

### 3. `pspecs = nnx.get_partition_spec(state)`

自动推断每个 LoRA 参数应该如何分片。

```
# 例如：
pspecs = {
    'layer1.q_einsum.lora_A': PartitionSpec('model', None),
    # [1024, 8] → 按第0维切分到 'model' 维度的 4 个设备
    # 每个设备存 [256, 8]

    'layer1.q_einsum.lora_B': PartitionSpec(None, 'model'),
    # [8, 1024] → 按第1维切分到 4 个设备
    # 每个设备存 [8, 256]
}
```

**为什么这样分片？**

- 确保 LoRA 参数的分片与原始权重对齐
- 计算 `lora_B @ (lora_A @ x)` 时，通信开销最小

---

### 4. `sharded_state = jax.lax.with_sharding_constraint(state, pspecs)`

强制将参数按照 `pspecs` 分布到设备。

```
# 之前 state 可能在单个设备上
# 执行后：
sharded_state['layer1.q_einsum.lora_A'] 被切成 4 份，分别在 GPU 0-3
```

**`with_sharding_constraint` 的作用：**

- 告诉 JAX 编译器："这个数据必须按这个方式分片"

- 如果已经符合，什么都不做
- 如果不符合，执行重新分布（类似 PyTorch 的 `tensor.to(device)`）

---

### 5. `nnx.update(lora_model, sharded_state)`

将分片后的参数更新回模型。

```
# 之前 lora_model 的参数可能未分片
# 现在替换为分片后的版本
lora_model.layers['layer1.q_einsum'].lora_A =
sharded_state['layer1.q_einsum.lora_A']
# 现在这个参数分布在多个设备上
```

---

## 完整流程总结

```
输入：base_model（已分片的基础模型）

1. 定义 LoRA 配置
   ├── 选择要修改的层（注意力 + FFN）
   ├── 设置 rank（低秩维度）
   └── 设置 alpha（缩放因子）

2. 添加 LoRA 适配器
   ├── 识别匹配的层
   ├── 为每层添加 A, B 两个小矩阵
   └── 修改前向传播：output = W@x + (alpha/rank)*B@(A@x)

3. 分片 LoRA 参数
   ├── 提取 LoRA 参数
   ├── 推断分片策略（与基础模型对齐）
   ├── 执行分片
   └── 更新回模型

输出：lora_model（基础模型 + LoRA 层，全部参数已分片）
```

---

## 实际例子

假设原始模型某层权重 `[1024, 1024]`，在 4 个 GPU 上按行切分：
```
原始权重（冻结，不训练）：
GPU 0: W[0:256, :]
GPU 1: W[256:512, :]
GPU 2: W[512:768, :]
GPU 3: W[768:1024, :]

添加 LoRA (rank=8):
GPU 0: lora_A[0:256, :] (256×8)    lora_B[:, 0:256] (8×256)
GPU 1: lora_A[256:512, :] (256×8)  lora_B[:, 256:512] (8×256)
GPU 2: lora_A[512:768, :] (256×8)  lora_B[:, 512:768] (8×256)
GPU 3: lora_A[768:1024, :] (256×8) lora_B[:, 768:1024] (8×256)

前向传播时：
每个 GPU 计算：W_local @ x_local + (alpha/rank) * lora_B_local @
(lora_A_local @ x_local)
```

```
# Reference model
if model_family == "gemma2":
  ref_model, mesh, model_config = get_gemma_ref_model(
      ckpt_path=os.path.join(INTERMEDIATE_CKPT_DIR, "state")
  )

# Policy model
lora_policy = get_lora_model(ref_model, mesh=mesh)
nnx.display(lora_policy)

if model_family == "gemma2":
  tokenizer = tokenizer_lib.Tokenizer(
      tokenizer_path=os.path.join(kaggle_ckpt_path, "tokenizer.model")
  )
```

# Define Reward Functions

We define four reward functions:

- reward if the format of the output exactly matches the instruction given in `TEMPLATE`;
- reward if the format of the output approximately matches the instruction given in `TEMPLATE`;
- reward if the answer is correct/partially correct;
- Sometimes, the text between `<answer>`, `</answer>` might not be one number. So, we extract the number, and reward the model if the answer is correct.

The reward functions are inspired from here.

我来详细解释 **Reward Function（奖励函数）** 是什么，以及这段代码中为什么需要它。

# 什么是 Reward Function?

## 背景：强化学习训练语言模型

在用 **RLHF（Reinforcement Learning from Human Feedback）** 或类似方法训练语言模型时，需要告诉模型"什么样的输出是好的"。

**类比：训练宠物狗**

- 狗做对了动作 → 给零食（正奖励）
- 狗做错了动作 → 不给零食或轻微惩罚（负奖励或零奖励）
- 通过反复奖励，狗学会正确的行为

**训练语言模型也类似：**

- 模型生成好的输出 → 给高分（正奖励）
- 模型生成差的输出 → 给低分（负奖励）
- 通过调整参数，让模型倾向于生成高奖励的输出

---

# Reward Function 的作用

**Reward Function（奖励函数）** 就是一个"评分器"，用来评估模型输出的质量。

```
def reward_function(model_output, ground_truth):
    """
    输入：
    - model_output：模型生成的文本
    - ground_truth：正确答案（如果有的话）

    输出：
    - score：一个数值，表示这个输出有多好
    """
    score = calculate_score(model_output, ground_truth)
    return score  # 例如：0.0（很差）到 1.0（完美）
```

---

## 这段代码中的 Reward Functions

代码定义了 **4 个不同的奖励函数**，从不同角度评估模型输出：

### **情境说明**

假设你在训练一个模型做数学题，要求输出格式如下：

```
TEMPLATE（期望格式）：
<reasoning>
Let me think step by step...
</reasoning>

<answer>
42
</answer>
```

---

## Reward Function 1: 精确格式匹配

```
# 奖励：输出格式**完全**符合 TEMPLATE
```

**评判标准：**

- 必须有 `<reasoning>` 和 `</reasoning>` 标签
- 必须有 `<answer>` 和 `</answer>` 标签
- 标签的顺序、位置都要正确

**例子：**

| 模型输出 | 奖励 | 原因 |
| --- | --- | --- |
| `<reasoning>...</reasoning>\n<answer>42</answer>` | ✅ 高分 | 格式完全正确 |
| `<answer>42</answer>\n<reasoning>...</reasoning>` | ❌ 低分 | 顺序错了 |
| `The answer is 42` | ❌ 零分 | 没有标签 |

**为什么需要这个?**

- 确保模型学会遵循指定的格式
- 在实际应用中，解析器可能依赖固定格式

---

## Reward Function 2: 近似格式匹配

```
# 奖励：输出格式**大致**符合 TEMPLATE
```

**评判标准：**

- 允许一些小的格式偏差
- 比如：标签顺序可能不同，但主要结构存在

**例子：**

| 模型输出 | 精确匹配 | 近似匹配 |
| --- | --- | --- |
| `<reasoning>...</reasoning>\n<answer>42</answer>` | ✅ | ✅ |
| `<answer>42</answer>\n<reasoning>...</reasoning>` | ❌ | ✅（顺序不同但都有） |
| `I think the answer is <answer>42</answer>` | ❌ | ✅（缺 reasoning 但有 answer） |
| `The answer is 42` | ❌ | ❌（完全没标签） |

**为什么需要这个?**

- 太严格的要求可能让训练困难
- 近似匹配可以作为"部分正确"的鼓励信号

---

## Reward Function 3: 答案正确性

```
# 奖励：答案是正确的或部分正确
```

**评判标准：**

- 提取 `<answer>` 标签中的内容
- 与标准答案比较

**例子（假设正确答案是 42）：**

| 模型输出 | 奖励 |
| --- | --- |
| `<answer>42</answer>` | ✅ 满分（完全正确） |
| `<answer>40</answer>` | 🟡 部分分（接近但不对） |
| `<answer>100</answer>` | ❌ 零分（完全错误） |

**部分正确的例子：**

- 数学题答案是 `42`，模型输出 `40` → 可能给 0.5 分（很接近）

- 多选题正确答案是 `A, B, C`，模型输出 `A, B` → 给 0.67 分（2/3 正确）

---

## Reward Function 4: 提取数字后再判断

```
# 奖励：即使格式不完美，只要能提取出正确的数字就给分
```

**为什么需要这个？** 模型可能在 `<answer>` 标签中写了额外的文字：

```
<!-- 期望的输出 -->
<answer>42</answer>

<!-- 实际的输出 -->
<answer>The final answer is 42.</answer>
<!-- 或者 -->
<answer>42 (rounded to the nearest integer)</answer>
```

**这个奖励函数的策略：**

1. 从 `<answer>` 标签中提取文本
2. 使用正则表达式找出所有数字
3. 判断是否包含正确答案

**例子（正确答案是 42）：**

| `<answer>` 标签内容 | 提取结果 | 奖励 |
|---|---|---|
| `42` | `42` | ✅ 满分 |
| `The answer is 42.` | `42` | ✅ 满分 |
| `42 (approximately)` | `42` | ✅ 满分 |
| `Forty-two` | 无法提取 | ❌ 零分 |
| `The answer is 100` | `100` | ❌ 零分 |

---

## 为什么需要多个 Reward Functions?

## 多维度评估更全面

训练时可以组合这些奖励：

```
# 伪代码
total_reward = (
    0.3 * format_exact_match_reward +      # 30% 权重给格式
    0.2 * format_approx_match_reward +     # 20% 权重给近似格式
    0.4 * answer_correctness_reward +      # 40% 权重给答案正确性
    0.1 * extracted_number_reward          # 10% 权重给提取后的答案
)
```

###

## 具体Reward Function的实现如下：

First off, let's define a RegEx for checking whether the format matches.... 见程序

Give the model a reward of 3 points if the format matches exactly.

```
def match_format_exactly(prompts, completions, **kwargs):
  return [
      0 if match_format.search(response) is None else 3.0
      for response in completions
  ]
```

We also reward the model if the format of the output matches partially.

```
def match_format_approximately(prompts, completions, **kwargs):
  scores = []

  for completion in completions:
    score = 0
    response = completion
    # Count how many keywords are seen - we penalize if too many!
    # If we see 1, then plus some points!
    score += 0.5 if response.count(reasoning_start) == 1 else -0.5
    score += 0.5 if response.count(reasoning_end) == 1 else -0.5
    score += 0.5 if response.count(solution_start) == 1 else -0.5
    score += 0.5 if response.count(solution_end) == 1 else -0.5
    scores.append(score)
  return scores
```

Reward the model if the answer is correct. A reward is also given if the answer does not match exactly, i.e., based on how close the answer is to the correct value.

```
def check_answer(prompts, completions, answer, **kwargs):
  responses = completions

  extracted_responses = [
      guess.group(1) if (guess := match_format.search(r)) is not None
else None
      for r in responses
  ]

  scores = []
  assert len(extracted_responses) == len(
      answer
  ), f"{extracted_responses} and {answer} have mismatching length"
  for guess, true_answer in zip(extracted_responses, answer):
    score = 0
    if guess is None:
      scores.append(0)
      continue
    # Correct answer gets 3 points!
    if guess == true_answer:
      score += 3.0
    # Match if spaces are seen
    elif guess.strip() == true_answer.strip():
      score += 1.5
    else:
      # We also reward it if the answer is close via ratios!
      # Ie if the answer is within some range, reward it!
      try:
        ratio = float(guess) / float(true_answer)
        if ratio >= 0.9 and ratio <= 1.1:
          score += 0.5
        elif ratio >= 0.8 and ratio <= 1.2:
          score += 0.25
        else:
          score -= 1.0  # Penalize wrong answers
      except:
        score -= 0.5  # Penalize
```

```
    scores.append(score)
  return scores
```

Sometimes, the text between `<answer>` and `</answer>` might not be one number; it can be a sentence. So, we extract the number and compare the answer.... 见程序

# Evaluate

Before we train the model, let's evaluate the model on the test set so we can see the improvement post training.

We evaluate it in two ways:

**Quantitative**

- **Answer Accuracy**: percentage of samples for which the model predicts the correct final numerical answer
- **Answer (Partial) Accuracy**: percentage of samples for which the model predicts a final numerical answer such that the `model answer / answer` ratio lies between 0.9 and 1.1.
- **Format Accuracy**: percentage of samples for which the model outputs the correct format, i.e., reasoning between the reasoning special tokens, and the final answer between the `<start_answer>`, `<end_answer>` tokens.

**Qualitative**

We'll also print outputs for a few given questions so that we can compare the generated output later.

## 设置 GRPO 训练器（Setting Up the GRPO Trainer）

现在我们开始初始化训练系统。首先，我们创建一个 `RLCluster` 实例，用于整合**策略模型**（`actor`）、**参考模型**（`reference`）以及 **tokenizer**。其中，`actor` 是一个可训练的 LoRA 模型，而 `reference` 是一个参数固定的基础模型，用于在训练过程中对策略模型起到约束与引导作用。

随后，我们创建 `GRPOLearner`，这是一个专门用于 GRPO 的训练器。该训练器通过一组**奖励函数（reward functions）**来评估模型生成结果，并据此优化模型参数，从而完成强化学习训练流程的搭建。

Tunix 训练器与 Weights & Biases 深度集成，可用于可视化训练过程。你可以根据需求选择不同的使用方式：

**选项 1（Type 1）**：适合快速实验或功能测试。该方式会在浏览器中创建一个临时的私有仪表盘，无需登录或创建账号。

**选项 2（Type 2）**：适合已有 W&B 账号、希望长期保存实验记录的用户。选择该方式后，你需要登录 W&B 或输入 API Key，训练过程和结果将被保存到你的个人项目仪表盘中。

# 一、第一部分：`RLCluster` —— 把“模型体系”组织起来

```
rl_cluster = rl_cluster_lib.RLCluster(
    actor=lora_policy,
    reference=ref_model,
    tokenizer=tokenizer,
    cluster_config=cluster_config,
)
```

## 👉 核心一句话

**`RLCluster` 是一个“强化学习用的模型运行环境”**，它把：

- 可训练模型
- 冻结参考模型
- tokenizer
- 并行 / 分布式配置

统一管理起来，供后面的 GRPO Trainer 使用。

---

## 1️⃣ `actor=lora_policy`（策略模型，真的会被训练）

- `lora_policy` = **加了 LoRA 的模型**
- 这是：
    - 当前策略 πθ

    - **参数会被 GRPO 更新**
- 它负责：
    - 给定 prompt → 生成多个回答（group）

📌 直觉比喻：
👉 **actor = 学生（你要教的人）**

---

## 2️⃣ `reference=ref_model`（参考模型，不会被训练）

- `ref_model` = **原始 base model**
- 参数是：
    - 完全冻结
- 用途是：
    - 计算 **KL divergence**
    - 防止策略模型跑偏

📌 数学上对应：

```
KL( π_actor || π_reference )
```

📌 直觉比喻：
👉 **reference = 老师 / 教科书 / 原本的语言风格**

---

## 3️⃣ `tokenizer=tokenizer`

- actor 和 reference **必须共用 tokenizer**
- 否则：
    - token id 对不上
    - logprob / KL 会全错

这是 RLHF/GRPO 的**硬性要求**。

---

## 4️⃣ `cluster_config=cluster_config`

这个是**工程层面的配置**，通常包括：

- FSDP / TP / DP
- mesh 结构
- device 数量
- batch / micro-batch 划分

👉 它决定的是：
**“这些模型怎么在多卡 / 多设备上跑”**
而不是“学什么”。

---

## ✅ 到这里为止，`RLCluster` 做完了一件事：

> **把“谁在学、谁不学、怎么跑模型”全部定下来**

---

## 二、第二部分：`GRPOLearner` —— 真正的 GRPO 训练器

```
grpo_trainer = GRPOLearner(
    rl_cluster=rl_cluster,
    reward_fns=[
        match_format_exactly,
        match_format_approximately,
        check_answer,
        check_numbers,
    ],
    grpo_config=grpo_config,
)
```

## 👉 核心一句话

**`GRPOLearner` 是一个“会执行 GRPO 算法的 trainer”**，它负责：

- 采样
- 计算 reward
- 算优势
- 更新 LoRA 参数

---

## 1️⃣ `rl_cluster=rl_cluster`

- 把刚才那个 **模型运行环境** 交给 GRPO
- GRPO learner 会从这里拿到：
  - actor
  - reference
  - tokenizer
  - 并行策略

👉 没有 `RLCluster`，GRPO trainer 根本不知道：

- 用哪个模型生成
- 用哪个模型算 KL

---

## 2️⃣ `reward_fns=[...]`（非常关键）

这里定义了：**“什么叫一个好回答”**

你给了 **4 个 reward function**：

```
reward_fns = [
    match_format_exactly,
    match_format_approximately,
    check_answer,
    check_numbers,
]
```

## 3️⃣ `grpo_config=grpo_config`

这里定义的是 **GRPO 算法细节**，比如：

- group size（一次生成多少个回答）
- KL 系数 β
- learning rate
- clip / normalization 方式
- 每 batch 做几次 update

📌 简单说：

- `reward_fns`：**学什么**
- `grpo_config`：**怎么学**