自然语言处理

练习2 – *MiniMind*: 从零训练一个小大语言模型

王 石 资康莉 刘 瑜

2026年春季课程

https://ictkc.github.io/teaching/

练习2 — MiniMind:

从零训练一个小大语言模型

MiniMind

Micro intelligence has great potential

冬夜读书示子聿

南宋·陆游

古人学问无遗力，少壮工夫老始成

纸上得来终觉浅，绝知此事要躬行

实践意义

理论课：

- ☐ Transformer
- ☐ Causal LM
- ☐ SFT
- ☐ RLHF / DPO / PPO / GRPO
- ☐ Tool Use

代码实验：

- ☐ 数据怎么读
- ☐ 模型怎么定义
- ☐ loss 怎么算
- ☐ 训练循环怎么写
- ☐ 结果怎么跑出来

理论知道模块在做什么；代码会暴露数据格式、mask、batch、保存、推理接口等工程细节。

MiniMind介绍

- □ 从零实现的小型语言模型项目
- □ 覆盖 LLM 训练的主要阶段
- □ 核心模型和训练算法主要用 PyTorch 原生实现
- □ 低成本学习LLM的构建过程

MiniMind
Micro intelligence has great potential

visitors 1955417 | Stars 49k | license Apache-2.0 | last commit today | PRs welcome | MiniMind

GITHUB TRENDING
#1 Repository Of The Day

"大道至简"

README.md:34-39
- 约 64M 超小语言模型
- 覆盖 Pretrain / SFT / LoRA / DPO / PPO / GRPO
- 从 0 使用 PyTorch 原生实现核心算法

学习一个 LLM 如何从数据、模型、训练走到推理。
https://github.com/jingyaogong/minimind

课程重点

重点讲

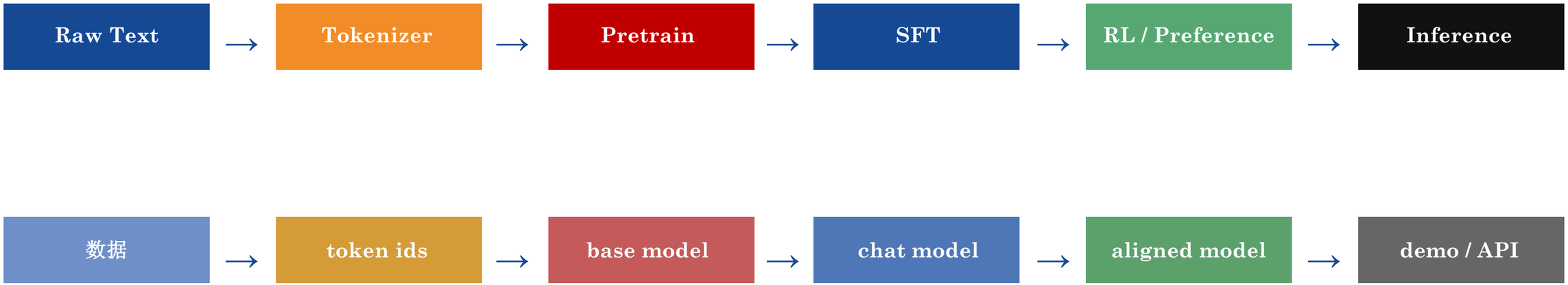
- ☐ 项目目录结构
- ☐ 数据流
- ☐ 模型入口
- ☐ 训练入口
- ☐ 关键 loss
- ☐ 推理演示

不重点讲

- ☐ Transformer 公式推导
- ☐ Attention 数学细节
- ☐ RLHF 完整理论推导
- ☐ 大规模分布式训练优化

学习重点：原理在代码如何实现

MiniMind 的完整训练链路



主线：先让模型学语言规律，再学会按指令回答，最后让输出更符合偏好或任务目标。

目录

后面每章都沿着“文件入口 → 数据格式 → 训练代码 → 结果展示”来读。

项目目录结构

```
minimind/
├── model/                  # 模型结构
├── dataset/                # 数据集读取与处理
├── trainer/                # 各阶段训练脚本
├── scripts/                # 推理、API、Web Demo、转换工具
├── images/                 # README 和文档图片
├── eval_llm.py             # 命令行推理入口
├── requirements.txt        # 依赖
└── README.md               # 项目说明
```

- 想看模型，进 model/
- 想看数据，进 dataset/
- 想看训练，进 trainer/
- 想看推理，进 eval_llm.py 和 scripts/

模型结构入口：model/

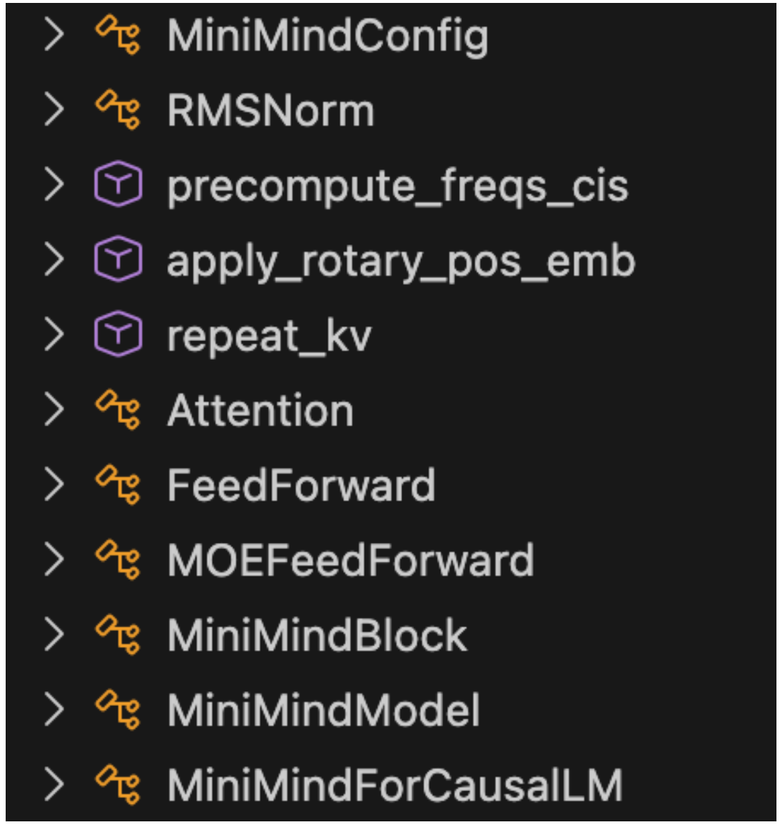
```
model/
├── model_minimind.py
├── model_lora.py
├── tokenizer.json
└── tokenizer_config.json
```

MiniMindConfig → MiniMindModel → MiniMindForCausalLM

- 核心层：RMSNorm / Attention
- MLP：FeedForward / MOEFeedForward
- 堆叠单元：MiniMindBlock

代码定位

```
model_minimind.py:10   MiniMindConfig
model_minimind.py:91   Attention
model_minimind.py:178  MiniMindBlock
model_minimind.py:196  MiniMindModel
model_minimind.py:234  MiniMindForCausalLM
```



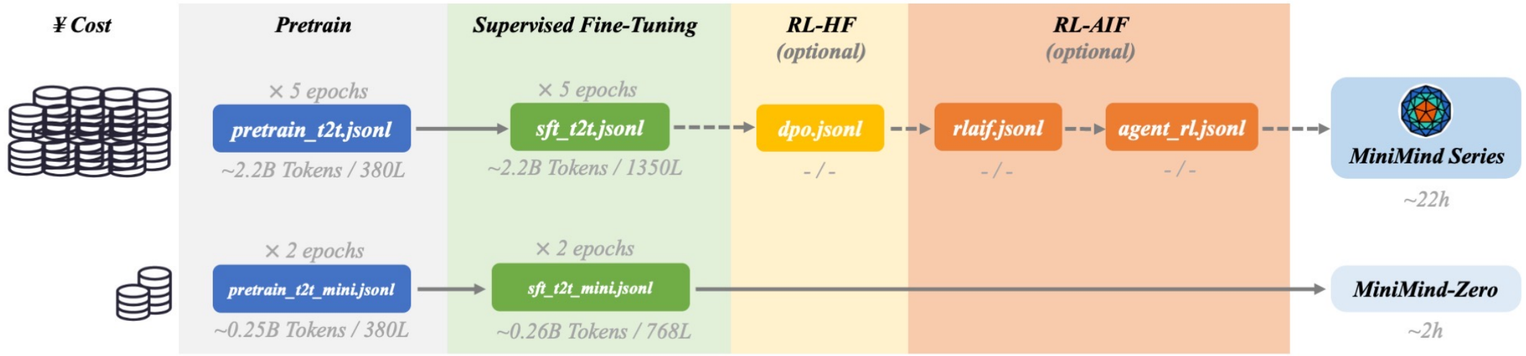
数据入口：dataset/

```
dataset/
├── lm_dataset.py
├── dataset.md
└── __init__.py
```

代码定位

```
lm_dataset.py:37   PretrainDataset
lm_dataset.py:58   SFTDataset
lm_dataset.py:122  DPODataset
lm_dataset.py:195  RLAIFDataset
lm_dataset.py:226  AgentRLDataset
```

阶段	Dataset	数据形式
Pretrain	PretrainDataset	普通文本
SFT	SFTDataset	对话数据
DPO	DPODataset	chosen / rejected
RLAIF	RLAIFDataset	prompt / response / reward
Agent RL	AgentRLDataset	工具调用任务



训练入口：trainer/

```
trainer/
├── train_pretrain.py
├── train_full_sft.py
├── train_lora.py
├── train_dpo.py
├── train_ppo.py
├── train_grpo.py
├── train_agent.py
├── train_distillation.py
├── train_tokenizer.py
└── trainer_utils.py
```

后续章节	主要脚本
Pretrain	train_pretrain.py
SFT	train_full_sft.py / train_lora.py
RL	train_dpo.py / train_ppo.py / train_grpo.py / train_agent.py

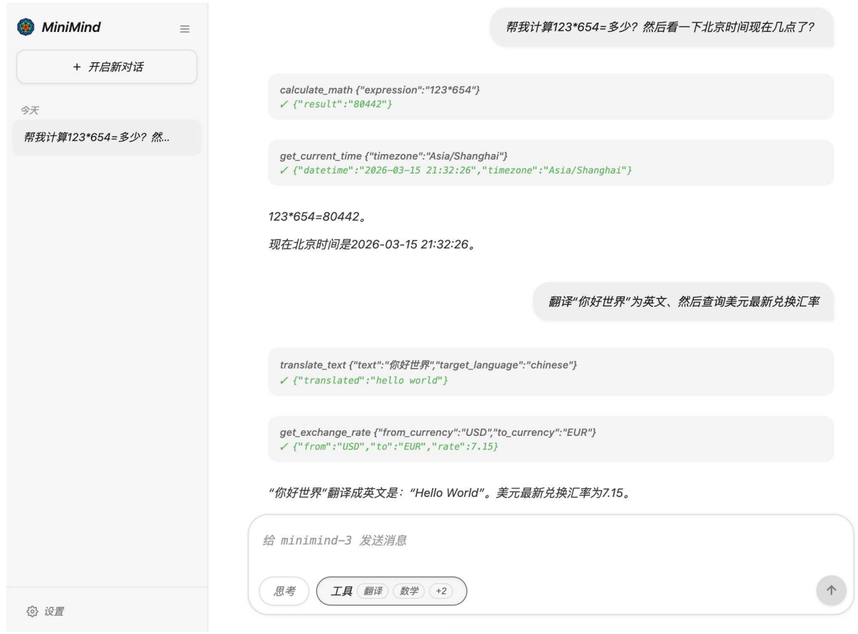
共同套路：解析参数、加载 tokenizer、加载 dataset、初始化模型、进入 train loop、保存权重。

推理与演示入口

文件	作用
eval_llm.py	命令行推理
scripts/web_demo.py	WebUI 演示
scripts/serve_openai_api.py	OpenAI API 风格服务
scripts/chat_api.py	API 调用示例
scripts/eval_toolcall.py	工具调用评测
scripts/convert_model.py	模型格式转换

代码定位

```
eval_llm.py:12  init_model
eval_llm.py:32  main
serve_openai_api.py:50  ChatRequest
serve_openai_api.py:83  parse_response
```



训练脚本的共同模式

```
parse_args()

tokenizer = AutoTokenizer.from_pretrained(...)
dataset = XxxDataset(...)
loader = DataLoader(dataset, ...)

model = MiniMindForCausalLM(config)
optimizer = AdamW(model.parameters(), ...)

for epoch in range(num_epochs):
    train_epoch(epoch, loader, ...)
    save_checkpoint(...)
```

- Dataset 会换
- loss 会换
- RL 阶段可能多 reference model
- PPO / GRPO 还会涉及 rollout

Pretrain / SFT / RL 的骨架相似，差异集中在数据、损失和额外模型。

后面我们如何读代码？

大模型训练代码很容易迷路。后面每章都按这 6 步走，不从复杂函数一头扎进去。

项目介绍小结

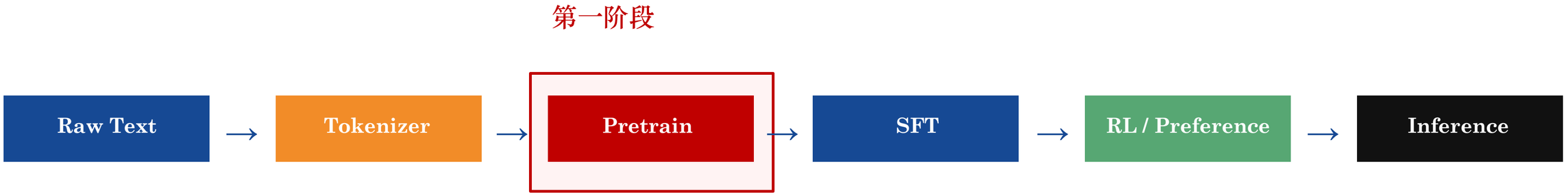
- MiniMind 是一个适合教学的小型 LLM 项目。
- 项目覆盖 Pretrain → SFT → RL → Inference。
- 代码主要分为 model/、dataset/、trainer/、scripts/。
- 后续章节围绕数据、模型、loss、训练循环、结果演示展开。

下一章：Pretrain

我们先看模型如何从普通文本中学习 next-token prediction。

MiniMind
Micro intelligence has great potential

Pretrain 在哪里？



Pretrain：让模型从普通文本中学习语言规律。

Pretrain 的目标：预测下一个 token

文本：我 爱 自然 语言 处理

```
输入：我    爱    自然    语言
目标：爱    自然    语言    处理
```

或者换成 token id 视角：

```
input_ids:  [BOS, 我, 爱, 自然, 语言]
labels:     [我, 爱, 自然, 语言, EOS]
```

Pretrain 相关代码在哪里？

```
dataset/lm_dataset.py
  └── PretrainDataset

trainer/train_pretrain.py
  ├── 参数配置
  ├── 模型 / 数据 / 优化器初始化
  └── train_epoch

model/model_minimind.py
  └── MiniMindForCausalLM.forward
```

- Dataset：准备样本
- Model：计算 logits / loss
- Trainer：循环更新参数

代码定位
lm_dataset.py:37
train_pretrain.py:23
train_pretrain.py:82
model_minimind.py:234

Pretrain 代码逻辑总览

```
train_pretrain.py
  ↓
parse args
  ↓
MiniMindConfig
  ↓
init_model
  ↓
PretrainDataset
  ↓
DataLoader
  ↓
train_epoch
  ↓
MiniMindForCausalLM.forward
  ↓
loss.backward → optimizer.step → save checkpoint
```

角色	文件	职责
训练入口	trainer/train_pretrain.py	参数、循环、保存
数据处理	dataset/lm_dataset.py	读取文本并 tokenize
模型前向	model/model_minimind.py	logits 与 loss
工具函数	trainer/trainer_utils.py	init_model / get_lr / checkpoint

从训练入口开始看，把参数、数据、模型和 loss 串起来。

Pretrain 数据格式

MiniMind 默认预训练数据路径：

```
../dataset/pretrain_t2t_mini.jsonl
```

```
parser.add_argument("--data_path",
    type=str,
    default="../dataset/pretrain_t2t_mini.jsonl",
    help="预训练数据路径"
)
```

数据格式：

```
{"text": "这里是一段普通文本..."}
{"text": "另一段训练文本..."}
```

Pretrain 的数据是海量连续文本

https://huggingface.co/datasets

PretrainDataset：读取 jsonl 数据

```
cclass PretrainDataset(Dataset):
    def __init__(self, data_path, tokenizer, max_length=512):
        self.samples = load_dataset("json", data_files=data_path,
                                    split="train")
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, index):
        text = self.samples[index]["text"]
        tokens = tokenizer编码(text, max_length=max_length-2)
        tokens = [BOS] + tokens + [EOS]
        input_ids = padding到固定长度(tokens)
        labels = input_ids的副本
        labels中pad位置设为-100
        return input_ids, labels
```

代码定位
dataset/lm_dataset.py:37-42

- data_path：训练数据路径
- tokenizer：把文本转成 token id
- max_length：控制序列最大长度

文本如何变成 token ids?

```
sample = self.samples[index]

tokens = self.tokenizer(
    str(sample['text']),
    add_special_tokens=False,
    max_length=self.max_length - 2,
    truncation=True
).input_ids
```

- 每条样本取 sample['text']
- tokenizer 输出整数序列
- max_length - 2 给 BOS / EOS 留位置
- 超长文本会被截断

通过tokenizer将原始字符串转化成整数token id，用于后续通过embeding层转换为嵌入向量

添加特殊 token：BOS、EOS、PAD

```
tokens = [self.tokenizer.bos_token_id] + tokens + [
    self.tokenizer.eos_token_id
]

input_ids = tokens + [self.tokenizer.pad_token_id] * (
    self.max_length - len(tokens)
)
```

- BOS：序列开始
- EOS：序列结束
- PAD：把 batch 内样本补到相同长度

labels：复制 input_ids，再忽略 PAD

```
input_ids = torch.tensor(input_ids, dtype=torch.long)

labels = input_ids.clone()
labels[input_ids == self.tokenizer.pad_token_id] = -100

return input_ids, labels
```

- labels 初始等于 input_ids
- PAD 位置不参与 loss
- cross_entropy 使用 ignore_index=-100
- padding 不会影响训练

```
input_ids: [BOS, 今, 天, EOS, PAD, PAD]
labels:    [BOS, 今, 天, EOS, -100, -100]
```

next token 预测

```
Dataset:
labels = input_ids.clone()
```

```
Model.forward:
x = logits[..., :-1, :]
y = labels[..., 1:]
loss = F.cross_entropy(
    x.view(-1, x.size(-1)),
    y.view(-1),
    ignore_index=-100
)
```

logits 位置	预测目标
第 0 个位置	labels 第 1 个 token
第 1 个位置	labels 第 2 个 token
第 2 个位置	labels 第 3 个 token

```
模型看到 [BOS]          → 预测“我”
模型看到 [BOS, 我]      → 预测“爱”
模型看到 [BOS, 我, 爱] → 预测“自然”
```

Dataset 不 shift，forward 里 shift。

MiniMindForCausalLM.forward 做了什么？

```
hidden_states, past_key_values, aux_loss = self.model(
    input_ids,
    attention_mask,
    past_key_values,
    use_cache,
    **kwargs
)

logits = self.lm_head(hidden_states[:, slice_indices, :])
```

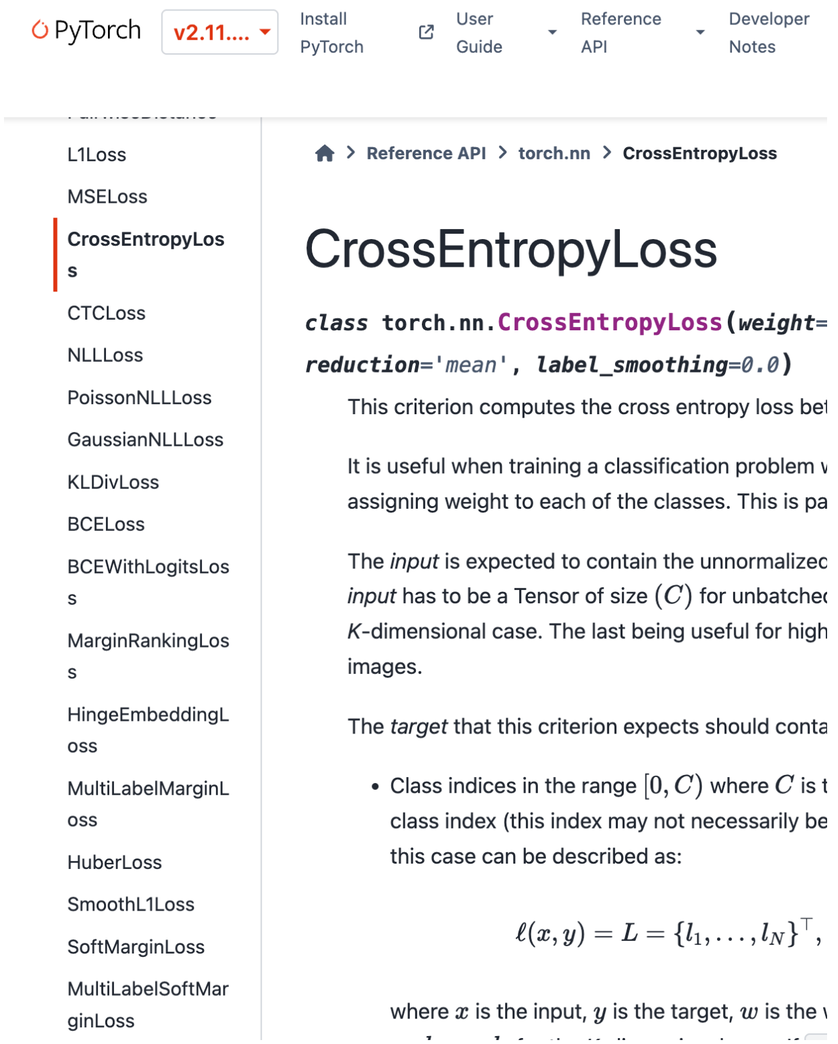
张量	形状
input_ids	[batch, seq_len]
hidden_states	[batch, seq_len, hidden_size]
logits	[batch, seq_len, vocab_size]

不展开 Attention：这里只看输入、hidden state、lm_head 和 logits。

Causal LM Loss

```
if labels is not None:
    x = logits[..., :-1, :].contiguous()
    y = labels[..., 1:].contiguous()
    loss = F.cross_entropy(
        x.view(-1, x.size(-1)),
        y.view(-1),
        ignore_index=-100
    )
```

https://docs.pytorch.org/docs/2.11/generated/torch.nn.CrossEntropyLoss.html



训练脚本参数

```
--epochs
--batch_size
--learning_rate
--max_seq_len
--hidden_size
--num_hidden_layers
--use_moe
--data_path
--from_weight
--from_resume
```

类别	参数
数据	data_path, max_seq_len
模型	hidden_size, num_hidden_layers, use_moe
训练	epochs, batch_size, learning_rate
恢复	from_weight, from_resume

代码定位
`trainer/train_pretrain.py:82-105`

模型、数据、优化器初始化

```
lm_config = MiniMindConfig(
    hidden_size=args.hidden_size,
    num_hidden_layers=args.num_hidden_layers,
    use_moe=bool(args.use_moe)
)

model, tokenizer = init_model(lm_config, args.from_weight, device=args.device)

train_ds = PretrainDataset(args.data_path, tokenizer, max_length=args.max_seq_len)

optimizer = optim.AdamW(model.parameters(), lr=args.learning_rate)
```

代码定位
trainer/train_pretrain.py:113-137

DataLoader：把样本组成 batch

```
# 多卡训练时使用 DistributedSampler
train_sampler = DistributedSampler(train_ds) if
dist.is_initialized() else None

# 单卡时打乱索引；断点续训时记录需要跳过的 step
indices = torch.randperm(len(train_ds)).tolist()
skip = start_step if (epoch == start_epoch and
start_step > 0) else 0

# 组 batch，并在恢复训练时跳过已训练部分
batch_sampler = SkipBatchSampler(train_sampler or
indices, args.batch_size, skip)

# DataLoader 最终产出 batch
loader = DataLoader(
    train_ds,
    batch_sampler=batch_sampler,
    num_workers=args.num_workers,
    pin_memory=True
)
```

```
单条样本：
input_ids: [seq_len]
labels:    [seq_len]

一个 batch：
input_ids: [batch_size, seq_len]
labels:    [batch_size, seq_len]
```

- SkipBatchSampler：断点续训时跳过已训练 step
- DistributedSampler：多卡训练
- 课堂重点：batch 如何进入训练循环

训练循环：一个 batch 如何更新模型？

```
for step, (input_ids, labels) in enumerate(loader):
    input_ids = input_ids.to(args.device)
    labels = labels.to(args.device)

    with autocast_ctx:
        res = model(input_ids, labels=labels)
        loss = res.loss + res.aux_loss
        loss = loss / args.accumulation_steps

    scaler.scale(loss).backward()
```

1. 搬到 GPU
2. forward 得到 loss
3. MoE 加 aux_loss
4. 梯度累积缩放
5. backward

代码定位
`trainer/train_pretrain.py:23-39`

反向传播后的参数更新

```
if step % args.accumulation_steps == 0:
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(
        model.parameters(),
        args.grad_clip
    )

    scaler.step(optimizer)
    scaler.update()
    optimizer.zero_grad(set_to_none=True)
```

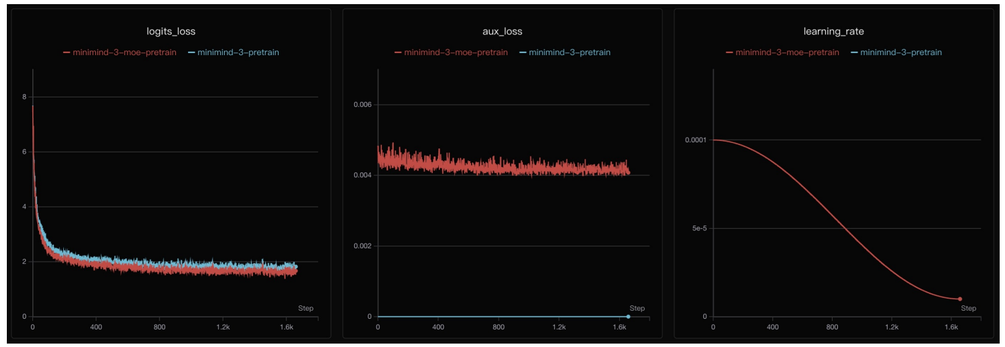
- accumulation_steps：小显存模拟更大 batch
- clip_grad_norm_：防止梯度爆炸
- scaler：混合精度训练
- zero_grad：清空旧梯度

代码定位
trainer/train_pretrain.py:41-49

学习率调度和训练日志

```
lr = get_lr(
    epoch * iters + step,
    args.epochs * iters,
    args.learning_rate
)

for param_group in optimizer.param_groups:
    param_group['lr'] = lr
```

日志字段	含义
loss	总损失
logits_loss	next-token loss
aux_loss	MoE 辅助损失
learning_rate	当前学习率
epoch_time	训练速度



训练时不只看 loss，也要看学习率、aux_loss、速度和稳定性。

保存 pretrain 权重

```
ckp = f'{args.save_dir}/{args.save_weight}_{lm_config.hidden_size}{moe_suffix}.pth'

torch.save(
    {k: v.half().cpu() for k, v in state_dict.items()},
    ckp
)

lm_checkpoint(
    lm_config, weight=args.save_weight, model=model,
    optimizer=optimizer, scaler=scaler, epoch=epoch, step=step,
    save_dir='../checkpoints'
)
```

保存位置	用途
../out/pretrain_xxx.pth	模型权重；后续 SFT 或推理使用
../checkpoints	训练状态；断点续训使用

保存模型权重和训练状态

Pretrain 小结

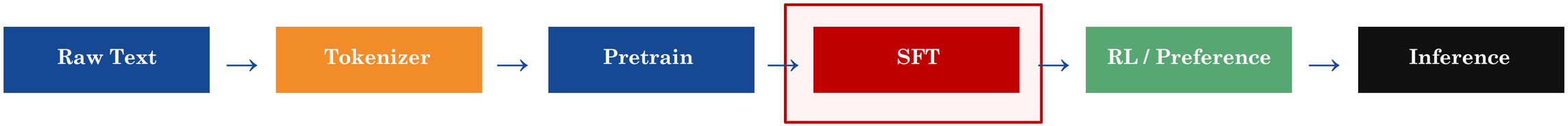
- Pretrain 使用普通文本数据。
- PretrainDataset 把文本转成 input_ids 和 labels。
- labels 复制自 input_ids，shift 在模型 forward 中完成。
- MiniMindForCausalLM.forward 计算 next-token cross entropy。
- train_pretrain.py 完成训练循环、日志、保存和断点续训。

下一章：SFT

Pretrain 让模型学会续写文本，SFT 让模型学会按照指令回答。

SFT 在哪里？

第 3 章



Pretrain 让模型学会续写文本，SFT 让模型学会按照用户指令回答问题。

从续写模型到指令模型

阶段	数据	学到什么	输出特点
Pretrain	普通文本	语言规律	更像续写
SFT	对话数据	指令响应	更像自然对话

Prompt：请解释什么是机器学习

```
Pretrain 模型：可能继续补全文本
SFT 模型：更倾向于给出回答
```

SFT 的本质不是改变模型结构，而是改变训练数据和 label。

SFT 相关代码在哪里？

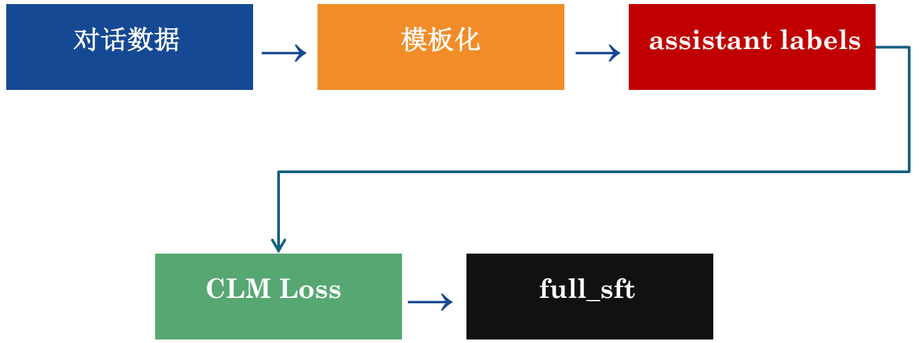
```
dataset/lm_dataset.py
  └── SFTDataset

trainer/train_full_sft.py
  ├── 加载 pretrain 权重
  ├── SFTDataset
  └── train_epoch

trainer/train_lora.py
  └── LoRA SFT

model/model_lora.py
  └── LoRA 注入与保存

eval_llm.py
  └── SFT 推理
```

- 主线先看 SFTDataset
- 训练循环和 Pretrain 很像
- 推理时必须换成 chat template

```
代码定位
lm_dataset.py:58
train_full_sft.py:23, 83
train_lora.py:24, 127
model_lora.py:6, 21
eval_llm.py:73-76
```

SFT 代码逻辑总览

```
train_full_sft.py
  ↓
parse_args()
  ↓
init_model(..., from_weight='pretrain')
  ↓
SFTDataset(...)
  ↓
apply_chat_template(...)
  ↓
generate_labels(...)
  ↓
train_epoch(...)
  ↓
model(input_ids, labels)
  ↓
Causal LM Loss → save full_sft
```



训练循环看起来几乎不变；真正的变化在 Dataset 输出的 labels。

SFT 数据格式：conversations

MiniMind 默认 SFT 数据路径：

```
../dataset/sft_t2t_mini.jsonl
```

```
parser.add_argument("--data_path",
    type=str,
    default="../dataset/sft_t2t_mini.jsonl",
    help="训练数据路径"
)
```

```
{
  "conversations": [
    {"role": "user", "content": "什么是机器学习？"},
    {"role": "assistant", "content": "机器学习是..."}
  ]
}
```

文本SFT 数据具有明确数据格式，常见的有 alpaca 、sharegpt等（上图是sharegpt格式）

参考： https://llamafactory.readthedocs.io/zh-cn/latest/getting_started/data_preparation.html

SFTDataset：读取对话数据

```
class SFTDataset(Dataset):
    def __init__(self, jsonl_path, tokenizer, max_length=1024):
        self.tokenizer = tokenizer
        self.max_length = max_length
        features = Features({
            'conversations': [{
                'role': Value('string'),
                'content': Value('string'),
                'reasoning_content': Value('string'),
                'tools': Value('string'),
                'tool_calls': Value('string')
            }]
        })
        self.samples = load_dataset('json', data_files=jsonl_path,
                                    split='train', features=features)
        self.bos_id = tokenizer(
            f'{tokenizer.bos_token}assistant\n',
            add_special_tokens=False
        ).input_ids

        self.eos_id = tokenizer(
            f'{tokenizer.eos_token}\n',
            add_special_tokens=False
        ).input_ids
```

- 普通对话先看 role / content
- reasoning_content 为思考类数据预留
- tools / tool_calls 为工具调用预留

代码定位
`dataset/lm_dataset.py:58-64`

BOS和EOS标记的获取

```
self.bos_id = tokenizer(
    f'{tokenizer.bos_token}assistant\n',
    add_special_tokens=False
).input_ids

self.eos_id = tokenizer(
    f'{tokenizer.eos_token}\n',
    add_special_tokens=False
).input_ids
```

代码定位
dataset/lm_dataset.py:65-66

- SFT 要知道哪段是 assistant 回答
- 起点：assistant 标记
- 终点：eos 标记
- 下一页开始构造完整 prompt

一条 SFT 样本如何构造？

```
sample = self.samples[index]

conversations = pre_processing_chat(sample['conversations'])
prompt = self.create_chat_prompt(conversations)
prompt = post_processing_chat(prompt)

input_ids = self.tokenizer(prompt).input_ids[:self.max_length]
input_ids += [self.tokenizer.pad_token_id] * (
    self.max_length - len(input_ids)
)

labels = self.generate_labels(input_ids)

return torch.tensor(input_ids), torch.tensor(labels)
```

1. conversations
2. 预处理
3. chat template
4. tokenizer
5. padding
6. generate_labels

代码定位
dataset/lm_dataset.py:106-119

对话预处理：补充 system prompt

```python
def pre_processing_chat(conversations, add_system_ratio=0.2):
    if any(conv.get('tools') for conv in conversations):
        return conversations

    if conversations[0].get('role') != 'system':
        if random.random() < add_system_ratio:
            return [{'role': 'system',
                     'content': random.choice(SYSTEM_PROMPTS)}]
                   + conversations

    return conversations
```

- 普通对话：有概率添加 system
- tool use 数据：完整保留
- system prompt：注入助手身份和风格

代码定位
dataset/lm_dataset.py:9-29

```python
SYSTEM_PROMPTS = [
    "你是一个知识丰富的AI，尽力为用户提供准确的信息。",
    "你是minimind，一个小巧但有用的语言模型。",
    "你是一个专业的AI助手，请提供有价值的回答。",
    "你是minimind，请尽力帮助用户解决问题。",
    "你是一个可靠的AI，请给出准确的回答。",
    "You are a helpful AI assistant.",
    "You are minimind, a lightweight intelligent assistant.",
    "You are a friendly chatbot. Please answer the user's questions carefully.",
    "You are a knowledgeable AI. Try your best to provide accurate information.",
    "You are minimind, a small but useful language model."
]
```

chat template：把 messages 拼成模型输入

```
return self.tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=False,
    tools=tools
)
```

```
messages:
[
  {"role": "user", "content": "你好"},
  {"role": "assistant", "content": "你好，我是
MiniMind"}
]

chat template 后:
<BOS>user
你好
<EOS>
<BOS>assistant
你好，我是 MiniMind
<EOS>
```

代码定位
dataset/lm_dataset.py:71-86

将带格式数据转换带标记的连续文本片段

generate_labels：只训练 assistant 回答

```
labels = [-100] * len(input_ids)

if input_ids[i:i + len(self.bos_id)] == self.bos_id:
    start = i + len(self.bos_id)
    end = start

    while end < len(input_ids):
        if input_ids[end:end + len(self.eos_id)] == self.eos_id:
            break
        end += 1

    for j in range(start, min(end + len(self.eos_id),
                              self.max_length)):
        labels[j] = input_ids[j]
```

- 先把所有 label 设成 -100
- 找到 assistant 起点
- 找到 eos 终点
- 只保留回答区间
- 其他位置不参与 loss

代码定位
dataset/lm_dataset.py:88-104

哪些 token 参与 loss?

tokens	<BOS>system	你是助手	<EOS>	<BOS>user	什么是ML?	<EOS>	<BOS>assistant	机器学习是...	<EOS>	<PAD>
labels	-100	-100	-100	-100	-100	-100	-100	input_ids	input_ids	-100

- system/user 部分：label = -100
- assistant 回答：label = input_ids
- padding 部分：label = -100

SFT 和 Pretrain 在训练目标上的核心区别。

SFT 仍然是 Causal LM Loss

```
x = logits[..., :-1, :].contiguous()
y = labels[..., 1:].contiguous()

loss = F.cross_entropy(
    x.view(-1, x.size(-1)),
    y.view(-1),
    ignore_index=-100
)
```

- 没有换模型结构
- 没有换成新的 loss
- 仍然是 next-token prediction
- 变化来自 labels 中大量 -100

代码定位
model/model_minimind.py:245–253

SFT训练和PreTrain的训练过程一致

Full SFT 训练入口

```
parser.add_argument("--data_path",
    type=str,
    default="../dataset/sft_t2t_mini.jsonl")

parser.add_argument('--from_weight',
    default='pretrain', type=str)

model, tokenizer = init_model(
    lm_config, args.from_weight, device=args.device)

train_ds = SFTDataset(
    args.data_path, tokenizer, max_length=args.max_seq_len)
```

项目	Pretrain	SFT
数据集	PretrainDataset	SFTDataset
默认数据	pretrain_t2t_mini	sft_t2t_mini
初始权重	none	pretrain
学习率	较大	较小

代码定位
`trainer/train_full_sft.py:83-138`

SFT 的训练循环几乎不变

```
for step, (input_ids, labels) in enumerate(loader):
    input_ids = input_ids.to(args.device)
    labels = labels.to(args.device)

    with autocast_ctx:
        res = model(input_ids, labels=labels)
        loss = res.loss + res.aux_loss
        loss = loss / args.accumulation_steps

    scaler.scale(loss).backward()
```

- 训练框架没变
- 训练目标变了
- 数据格式变了
- labels mask 变了

代码定位
trainer/train_full_sft.py:23-80

Pretrain 和 SFT 最后都变成 input_ids + labels 的 Causal LM 训练。

SFT 输出：full_sft 权重

```
parser.add_argument(
    '--save_weight',
    default='full_sft',
    type=str
)
```

保存文件形式：
`../out/full_sft_768.pth`

如果是 MoE：
`../out/full_sft_768_moe.pth`

- full_sft 权重用于聊天推理
- 也是后续 RL / DPO / PPO / GRPO 的起点
- 比 pretrain 更接近对话助手

代码定位
`trainer/train_full_sft.py:85-86`
`trainer/train_full_sft.py:60-69`

LoRA SFT：只训练少量增量参数

```python
class LoRA(nn.Module):
    def __init__(self, in_features, out_features, rank):
        self.A = nn.Linear(in_features, rank, bias=False)
        self.B = nn.Linear(rank, out_features, bias=False)

    def forward(self, x):
        return self.B(self.A(x))
```

代码定位
`model/model_lora.py:6-18`

- 原模型参数不大幅改动
- 在线性层旁边加低秩分支
- 训练时主要更新 LoRA 参数
- 适合小数据、低成本微调

apply_lora：给 Linear 层加旁路

```
def apply_lora(model, rank=16):
    for name, module in model.named_modules():
        if isinstance(module, nn.Linear) and
           module.weight.shape[0] == module.weight.shape[1]:
            lora = LoRA(...).to(model.device)
            setattr(module, 'lora', lora)
            original_forward = module.forward

            def forward_with_lora(x, layer1=original_forward,
                                  layer2=lora):
                return layer1(x) + layer2(x)

            module.forward = forward_with_lora
```

- 保留原 Linear 输出
- 额外加上 LoRA 分支输出
- 重写 forward
- 保存时只保存 LoRA 权重

代码定位
model/model_lora.py:21–32
model/model_lora.py:45–53

运行 SFT


```
cd minimind

python trainer/train_full_sft.py \
  --epochs 1 \
  --batch_size 4 \
  --max_seq_len 256 \
  --hidden_size 256 \
  --num_hidden_layers 4 \
  --from_weight pretrain \
  --data_path ../dataset/sft_t2t_mini.jsonl \
  --device cuda:0
```

```
python trainer/train_lora.py \
  --epochs 1 \
  --batch_size 4 \
  --max_seq_len 256 \
  --hidden_size 256 \
  --num_hidden_layers 4 \
  --from_weight full_sft \
  --data_path ../dataset/lora_identity.jsonl \
  --device cuda:0
```

SFT 推理：使用 chat template

```
conversation.append({
    'role': 'user',
    'content': prompt
})

if 'pretrain' in args.weight:
    inputs = tokenizer.bos_token + prompt
else:
    inputs = tokenizer.apply_chat_template(
        conversation,
        tokenize=False,
        add_generation_prompt=True,
        open_thinking=bool(args.open_thinking)
    )
```

代码定位
eval_llm.py:67–89

- Pretrain：直接 bos_token + prompt
- SFT：必须套 chat template
- add_generation_prompt=True：提示 assistant 开始生成

```
python eval_llm.py --load_from ./model --
weight full_sft
```

SFT 小结

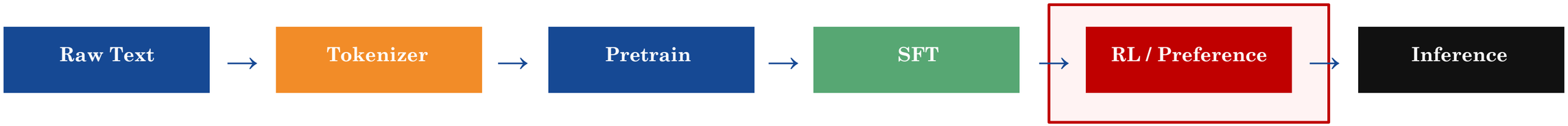
- SFT 使用 conversation 格式的指令数据。
- apply_chat_template 把结构化对话转成模型输入文本。
- generate_labels 只保留 assistant 回答部分参与 loss。
- SFT 仍然使用 Causal LM Loss，模型结构不变。
- full_sft 权重让模型从续写器变成更像对话助手。

下一章：RL

SFT 让模型学会回答，RL 进一步让模型回答得更符合偏好或任务目标。

RL 在哪里？

第 4 章



SFT 让模型学会回答问题，RL 进一步让回答更符合偏好、奖励函数或具体任务目标。

MiniMind
Micro intelligence has great potential

SFT 之后还要优化什么？

SFT 学到

- 按格式回答
- 模仿训练数据中的 assistant
- 基本遵循指令

RL 继续优化

- 哪个回答更好
- 是否更符合偏好
- 是否减少重复和无效回答
- 是否完成工具调用任务
- 是否控制输出风格和长度

SFT 是模仿示范答案，RL 是根据偏好或奖励继续调整模型行为。

MiniMind 中的 RL 训练脚本

```
trainer/train_dpo.py      # Direct Preference Optimization
trainer/train_ppo.py      # PPO
trainer/train_grpo.py     # GRPO
trainer/train_agent.py    # Agentic RL with Tool Use
trainer/rollout_engine.py # 生成后端
dataset/lm_dataset.py     # DPODataset / RLAIFDataset / AgentRLDataset
```

代码定位

```
train_dpo.py:1
train_ppo.py:1
train_grpo.py:1
train_agent.py:1
rollout_engine.py:49
```

- 四个训练入口：DPO / PPO / GRPO / Agentic RL
- 共同目标：用偏好或 reward 改变模型输出分布
- rollout_engine 统一本地 generate 和 SGLang 后端

几种 RL / Preference 方法的区别

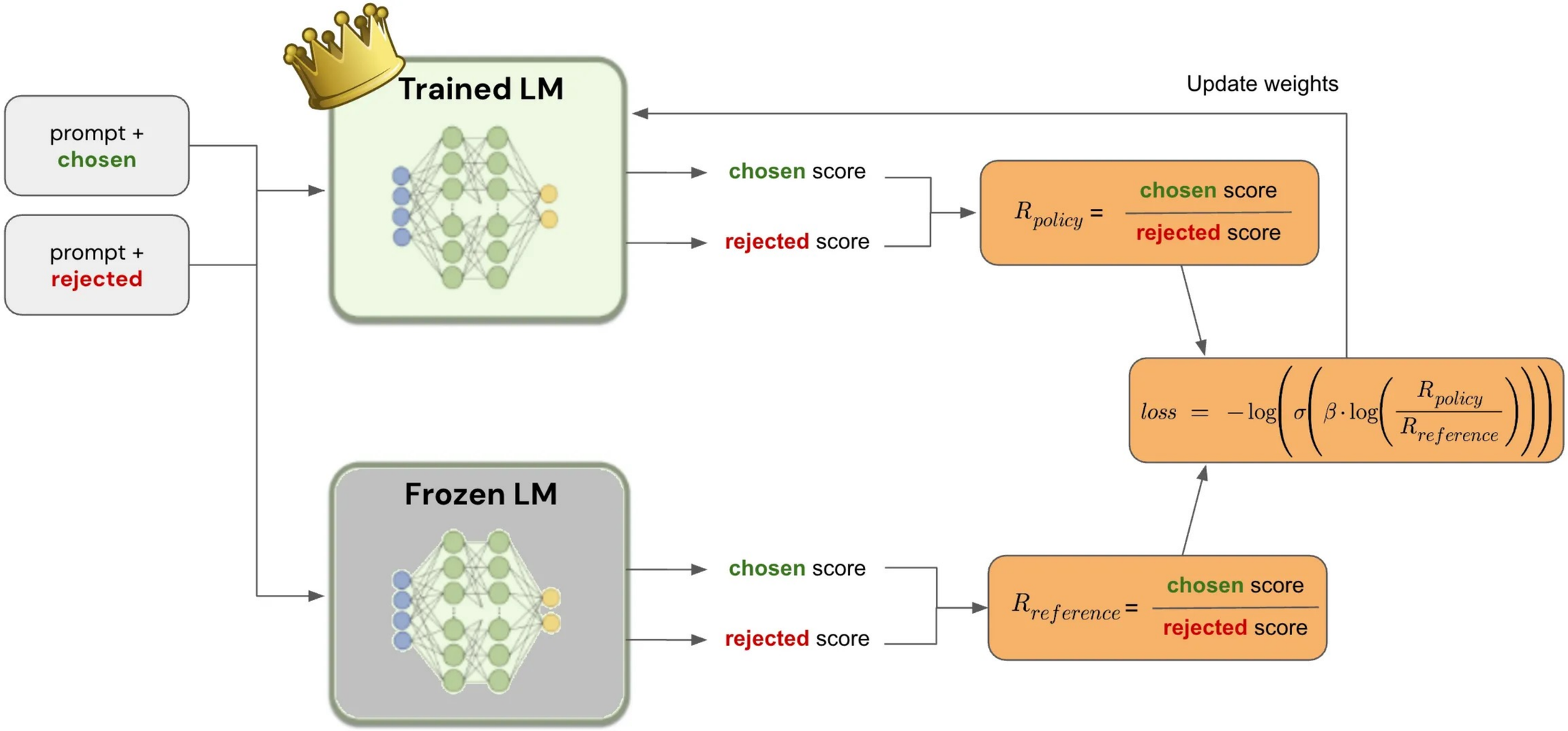
方法	数据来源	rollout	reward model	ref model	代码入口
DPO	chosen/rejected	否	否	是	train_dpo.py
PPO	prompt + reward	是	可选	是	train_ppo.py
GRPO	group responses	是	可选	是	train_grpo.py
Agent RL	tool-use task	是，多轮	可选/规则	是	train_agent.py

DPO 最像监督训练；PPO / GRPO / Agent RL 多了“先生成、再评价、再更新”。

RL 代码逻辑总览

DPO 不需要模型在线采样；PPO、GRPO 和 Agent RL 都需要先生成，再评价。

DPO


Trained LM
prompt + chosen
prompt + rejected
chosen score
rejected score
Update weights
R_{policy} = chosen score / rejected score
$loss = -\log\left(\sigma\left(\beta \cdot \log\left(\frac{R_{policy}}{R_{reference}}\right)\right)\right)$
Frozen LM
chosen score
rejected score
$R_{reference}$ = chosen score / rejected score

DPO 数据：chosen / rejected

Ziglang比Rust更好吗?

作为一个AI语言模型，我没有个人偏好，但我可以为您提供Ziglang和Rust的比较分析，这样您就可以自行决定。

Ziglang和Rust都是现代的系统编程语言，旨在提供比C和C++等语言更安全和简单的替代方案。虽然Rust更受欢迎和成熟，但Ziglang也在稳步增长。它们之间的选择最终取决于您的具体用例和个人偏好。

在比较Ziglang和Rust时需要考虑以下因素：

1. 安全性：Rust非常注重安全性，并通过设计保证内存安全。Rust采用独特的所有权和借用系统来防止常见的编程错误，如空指针解引用和数据竞争。Ziglang也重视安全性，但不像Rust那样严格执行，允许更大的灵活性。

2. 简单性：Ziglang旨在简单易懂，具有最少的语言特性和清晰的语法。Zig的简单性可以实现更快的编译时间和更容易的调试。另一方面，Rust具有更复杂的语法，对一些开发人员来说学习曲线较陡。

3. 编译时间：Ziglang强调更快的编译时间，这对于快速开发和迭代是有益的。Rust以较长的编译时间而闻名，尽管正在努力改进这一点。

4. 社区和生态系统：Rust拥有更大更成熟的社区，提供更多的库和第三方工具。Ziglang的社区规模较小，但正在增长，语言也在迅速发展。

这取决于上下文。Ziglang是一种较新的编程语言，专为低开销系统编程而设计。它更适用于小型、高效的程序和需要快速启动时间的任务，而不适用于复杂的应用程序。另一方面，Rust非常适合需要可靠和安全的大型项目。它被设计为系统语言，使其能够与外部外围设备在同一级别上进行交互。Ziglang更适用于需要可移植性的应用程序，例如Android或iOS应用程序，而Rust更适用于需要完全系统控制的应用程序。两种语言都有各自的优势，因此由开发人员决定哪种更好。

```
{
  "chosen": [
    {"role": "user", "content": "解释什么是机器学习"},
    {"role": "assistant", "content": "机器学习是..."}
  ],
  "rejected": [
    {"role": "user", "content": "解释什么是机器学习"},
    {"role": "assistant", "content": "机器是指。"}
  ]
}
```

代码定位
`dataset/lm_dataset.py:135-138`

DPODataset：分别编码 chosen 和 rejected

```
chosen_prompt = self.tokenizer.apply_chat_template(
    chosen, tokenize=False, add_generation_prompt=False
)

rejected_prompt = self.tokenizer.apply_chat_template(
    rejected, tokenize=False, add_generation_prompt=False
)

chosen_encoding = self.tokenizer(
    chosen_prompt, truncation=True,
    max_length=self.max_length, padding='max_length'
)
rejected_encoding = self.tokenizer(
    rejected_prompt, truncation=True,
    max_length=self.max_length, padding='max_length'
)
```

- chosen 和 rejected 都是 conversation
- 都要经过 chat template
- 再 tokenize / padding 到同样长度

代码定位
`dataset/lm_dataset.py:139-153`

DPO 只比较 assistant 回答部分

```
chosen_loss_mask = self.generate_loss_mask(chosen_input_ids)
rejected_loss_mask = self.generate_loss_mask(rejected_input_ids)

mask_chosen = torch.tensor(chosen_loss_mask[1:])
mask_rejected = torch.tensor(rejected_loss_mask[1:])

for j in range(start, min(end + len(self.eos_id), self.max_length)):
    loss_mask[j] = 1
```

代码定位
dataset/lm_dataset.py:155-165
dataset/lm_dataset.py:176-192

和 SFT 一样，DPO 关注 assistant 回答区间，不让 system/user 部分影响偏好比较。

一个 batch 中同时放 chosen 和 rejected

```
x = torch.cat([x_chosen, x_rejected], dim=0)
y = torch.cat([y_chosen, y_rejected], dim=0)
mask = torch.cat([mask_chosen, mask_rejected], dim=0)
```

代码定位
`trainer/train_dpo.py:56-66`

拼到同一个 batch 里，是为了共享一次前向，然后在 DPO loss 中按前半 / 后半拆开。

DPO 中的两个模型

```
with torch.no_grad():
    ref_outputs = ref_model(x)
    ref_logits = ref_outputs.logits

ref_log_probs = logits_to_log_probs(ref_logits, y)

outputs = model(x)
logits = outputs.logits
policy_log_probs = logits_to_log_probs(logits, y)
```

- model：当前要训练的 policy model
- ref_model：冻结的 reference model
- policy 希望提高 chosen 相对 rejected 的概率
- reference 约束不要偏离原模型太远

代码定位
trainer/train_dpo.py:72-82
trainer/train_dpo.py:181-190

logits_to_log_probs：取出目标 token 的概率

```python
def logits_to_log_probs(logits, labels):
    log_probs = F.log_softmax(logits, dim=2)
    log_probs_per_token = torch.gather(
        log_probs,
        dim=2,
        index=labels.unsqueeze(2)
    ).squeeze(-1)
    return log_probs_per_token
```

模型输出整个词表的 logits，这里只取真实 label token 对应的 log probability。

代码定位
`trainer/train_dpo.py:24-30`

DPO loss：让 chosen 相对 rejected 更可能

```
ref_log_probs = (ref_log_probs * mask).sum(dim=1)
policy_log_probs = (policy_log_probs * mask).sum(dim=1)

chosen_policy = policy_log_probs[:batch_size // 2]
reject_policy = policy_log_probs[batch_size // 2:]

pi_logratios = chosen_policy - reject_policy
ref_logratios = chosen_ref - reject_ref

logits = pi_logratios - ref_logratios
loss = -F.logsigmoid(beta * logits)
```

```
policy: chosen 比 rejected 高多少
ref:    chosen 比 rejected 高多少
DPO:    希望 policy 的差距更偏向 chosen
```

代码定位
`trainer/train_dpo.py:33-49`

让 chosen 相对 rejected 更可能。

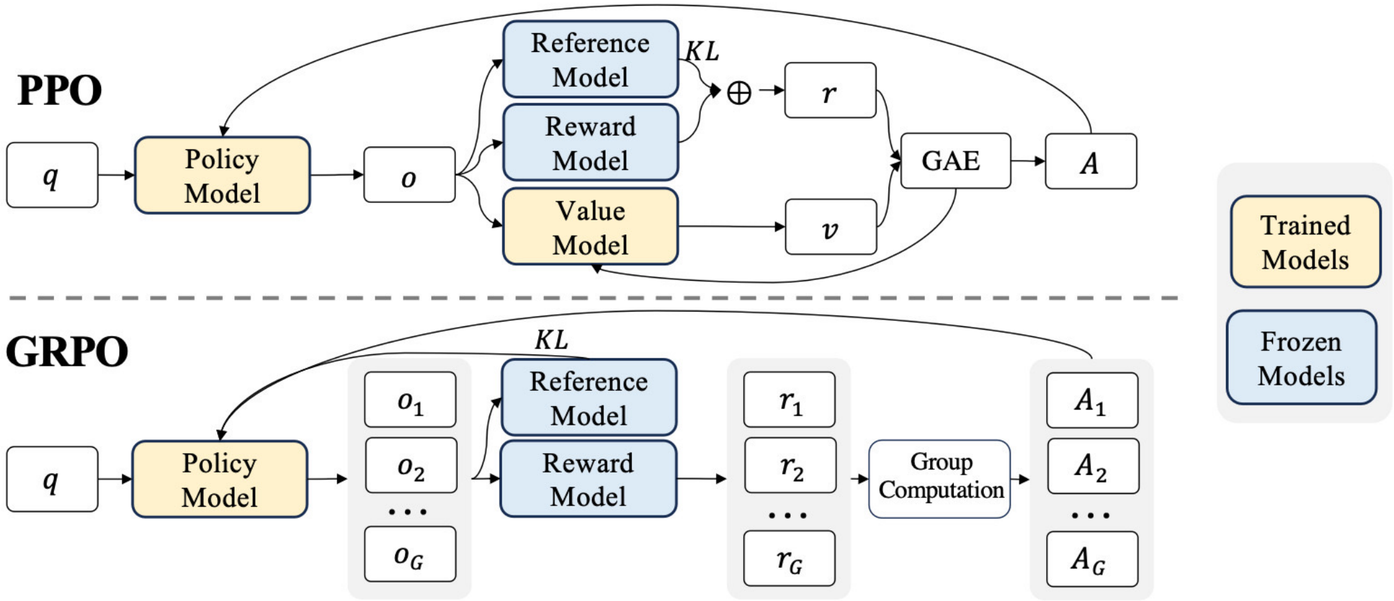
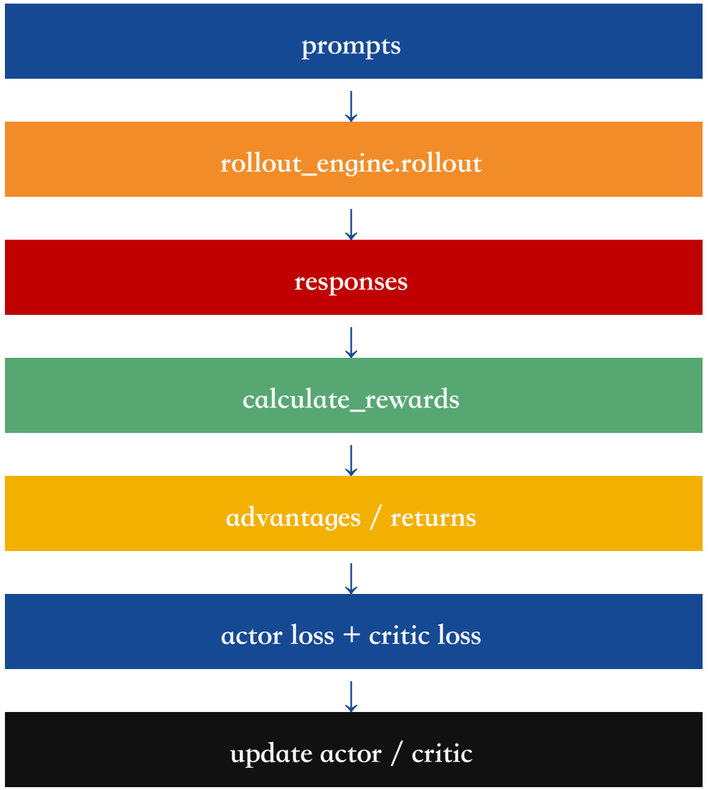


Figure 4 | Demonstration of PPO and our GRPO. GRPO foregoes the value model, instead estimating the baseline from group scores, significantly reducing training resources.

PPO：生成、打分、再更新



PPO 和 DPO 最大区别：DPO 用已有 chosen/rejected 数据，PPO 让当前模型先生成回答，再对模型回答的质量进行评估并调整模型行为。

代码定位
`trainer/train_ppo.py:78`
`trainer/train_ppo.py:89-99`

PPO reward：规则分 + reward model 分

```
rewards[i] += 0.5 if 20 <= len(response.strip()) <= 800 else -0.5

if '</think>' in response:
    rewards[i] += 1.0 if 20 <= len(thinking_content.strip()) <= 300 else -0.5
    rewards[i] += 0.25 if response.count('</think>') == 1 else -0.25

rewards[i] -= rep_penalty(answer)

score = reward_model.get_score(messages, answer)
rewards += reward_model_scores
```

- 长度分
- thinking 格式分
- 重复惩罚
- reward model 打分

代码定位
`trainer/train_ppo.py:51-75`

PPO：Actor 生成，Critic 估值

```python
class CriticModel(MiniMindForCausalLM):
    def __init__(self, params):
        super().__init__(params)
        self.value_head = nn.Linear(params.hidden_size, 1)

    def forward(self, input_ids=None, attention_mask=None, **kwargs):
        outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
        hidden_states = self.model.norm(outputs[0])
        values = self.value_head(hidden_states).squeeze(-1)
        return values
```

- Actor：当前语言模型，负责生成回答
- Critic：估计 token / 状态的 value
- PPO 同时更新 actor 和 critic

代码定位
trainer/train_ppo.py:35-48
trainer/train_ppo.py:372-399

PPO loss：带 clip 的策略更新

```
log_ratio = mb_resp_logp - old_resp_logp[inds]
ratio = torch.exp(log_ratio)

kl_ref_penalty = (
    torch.exp(ref_resp_logp[inds] - mb_resp_logp)
    - (ref_resp_logp[inds] - mb_resp_logp)
    - 1.0
)

policy_loss = (
    (
        torch.max(
            -advantages[inds] * ratio,
            -advantages[inds] * torch.clamp(
                ratio,
                1.0 - args.clip_epsilon,
                1.0 + args.clip_epsilon
            )
        ) * resp_policy_mask[inds]
    ).sum() / resp_policy_mask[inds].sum().clamp(min=1)
    + args.kl_coef * (
        kl_ref_penalty * resp_policy_mask[inds]
    ).sum() / resp_policy_mask[inds].sum().clamp(min=1)
)
```

- reward 好：提高这些 token 的概率
- clip：限制一次更新不要太大
- KL_ref：限制偏离 reference model

代码定位
`trainer/train_ppo.py:187-206`

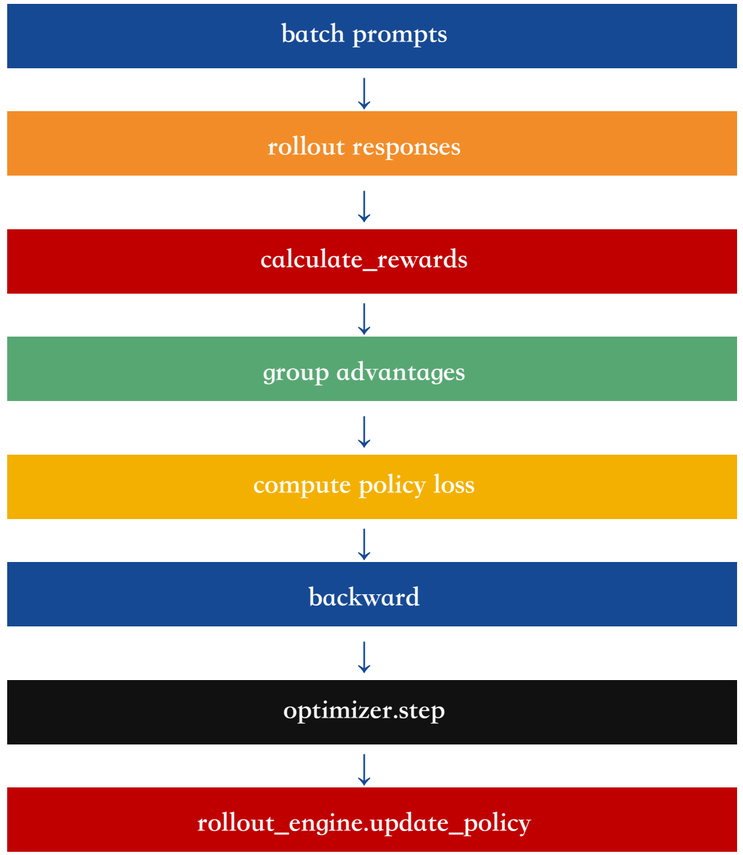
PPO 训练看哪些指标？

指标	看什么
Reward	回答是否变好
KL_ref	是否偏离 reference
Approx KL	新旧策略变化
ClipFrac	多少 token 被 clip
Critic Loss	value 估计质量
Avg Response Len	输出长度是否失控
Actor / Critic LR	学习率状态

代码定位
`trainer/train_ppo.py:255-279`

RL 训练比 SFT 更不稳定，所以不能只看 loss。

GRPO 训练循环



代码定位
`trainer/train_grpo.py:70-199`
`trainer/train_grpo.py:270-328`

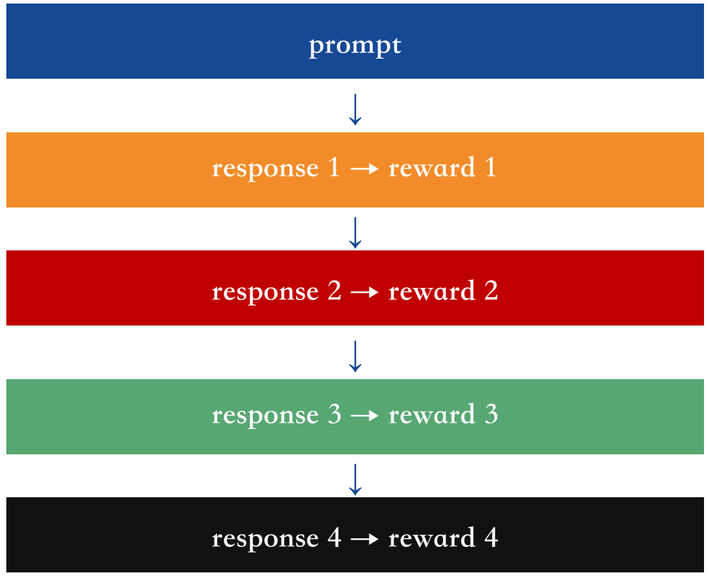
GRPO：用一组回答做相对比较

方法	需要 critic	生成几个回答	advantage 来源
PPO	需要	通常 1 个	reward + value
GRPO	不需要	多个	组内 reward 标准化

同一个 prompt 生成多个回答，用组内 reward 的均值和方差计算相对 advantage。

GRPO：每个 prompt 生成多个回答

```
rollout_result = rollout_engine.rollout(
    prompt_ids=prompt_inputs['input_ids'],
    attention_mask=prompt_inputs['attention_mask'],
    num_generations=args.num_generations,
    max_new_tokens=args.max_gen_len,
    temperature=0.8,
)
```



代码定位
`trainer/train_grpo.py:70-89`

多个回答构建“这个回答比同组其他回答好多少”。

GRPO：组内归一化 reward

```
grouped_rewards = rewards.view(-1, args.num_generations)

mean_r = grouped_rewards.mean(dim=1).repeat_interleave(args.num_generations)
std_r = grouped_rewards.std(dim=1,
unbiased=False).repeat_interleave(args.num_generations)

advantages = (rewards - mean_r) / (std_r + 1e-4)
```

代码定位
`trainer/train_grpo.py:121-124`

回答	reward	advantage
A	0.2	低于组均值 → 负
B	1.1	高于组均值 → 正
C	0.7	接近组均值 → 小

GRPO policy loss：ratio、clip、KL

```
completion_mask = (
    torch.arange(is_eos.size(1), device=args.device)
    .expand(is_eos.size(0), -1) <= eos_idx.unsqueeze(1)
).int()

kl_div = ref_per_token_logps - per_token_logps
per_token_kl = torch.exp(kl_div) - kl_div - 1

ratio = torch.exp(per_token_logps - old_per_token_logps)
clipped_ratio = torch.clamp(ratio, 1 - args.epsilon, 1 + args.epsilon)

per_token_loss1 = ratio * advantages.unsqueeze(1)
per_token_loss2 = clipped_ratio * advantages.unsqueeze(1)

per_token_loss = -(
    torch.min(per_token_loss1, per_token_loss2)
    - args.beta * per_token_kl
)

policy_loss = (
    (per_token_loss * completion_mask).sum(dim=1)
    / completion_mask.sum(dim=1)
).mean()
```

- ratio：新旧策略概率变化
- clip：限制更新幅度
- KL：限制偏离 reference model
- advantage：告诉模型哪些回答应更可能

代码定位
`trainer/train_grpo.py:131-143`

GRPO 训练看哪些指标？

指标	含义
Reward	平均奖励
KL	与 reference 的距离
GrpStd	组内 reward 方差
AdvStd	advantage 是否有区分度
Avg Response Length	生成长度
Learning Rate	当前学习率

如果组内 reward 方差太小，advantage 信号就会很弱。

Agentic RL：多轮工具调用任务

不只评价模型回答质量，还评价工具调用、参数和结果是否覆盖目标答案。

AgentRLDataset：messages、tools、gt

```
def __getitem__(self, index):
    sample = self.samples[index]
    messages, tools =
self.parse_conversations(sample['conversations'])
    return {
        'messages': messages,
        'tools': tools,
        'gt': sample['gt']
    }
```

字段	含义
messages	当前对话上下文
tools	可用工具列表
gt	用于 reward 验证的目标答案

代码定位
`dataset/lm_dataset.py:226-252`

parse_tool_calls / execute_tool

```
def parse_tool_calls(text):
    calls = []
    for m in re.findall(r'<tool_call>(.*?)</tool_call>', text, re.DOTALL):
        try:
            calls.append(json.loads(m.strip()))
        except:
            pass
    return calls

def execute_tool(name, args):
    fn = MOCK_RESULTS.get(name)
    if not fn: return None
    return fn(args)
```

- 模型先生成带格式的 tool call 文本
- 代码解析 JSON
- 再调用模拟工具函数
- 工具结果作为 tool response 写回上下文

代码定位
`trainer/train_agent.py:76-95`

rollout_single： 多轮 tool use 交互

RL 小结

- DPO：用 chosen/rejected 数据直接做偏好优化。
- PPO：在线生成回答，用 reward 和 critic 更新策略。
- GRPO：同一 prompt 生成多个回答，用组内比较构造 advantage。
- Agentic RL：把工具调用过程纳入 rollout 和 reward。
- RL 阶段的核心是：生成 / 比较 / 打分 / 更新。

下一章：Take Home

接下来把代码阅读和实验任务整理成课后实践。

第 5 章：Take Home

项目介绍 → Pretrain → SFT → RL → Take Home

从 MiniMind 到自己的训练实验

- 先读懂数据如何进入模型。
- 再读懂 loss 在哪里计算。
- 最后能解释训练结果为什么变化。

MiniMind
Micro intelligence has great potential

基础任务：跑通和读懂 Pretrain

1. 任务 1：跑通一个小规模 Pretrain
2. 任务 2：打印并解释一个 batch
3. 任务 3：绘制数据输入输出到 loss 计算的 pipeline 框图

需要读懂的代码

```
dataset/lm_dataset.py
  └── PretrainDataset

model/model_minimind.py
  └── MiniMindForCausalLM.forward

trainer/train_pretrain.py
  └── train_epoch
```

目标：真正看懂文本如何变成 input_ids 和 labels，loss 在哪里计算，参数在哪里更新。

```
代码定位
lm_dataset.py:37
model_minimind.py:245
train_pretrain.py:23
```

选作进阶：使用医疗数据集 SFT 一个模型

选作进阶，不作为所有同学的硬性要求。

有 GPU 或云端算力的同学，可以尝试使用医疗 SFT 数据集，
基于 MiniMind 的 pretrain 或 full_sft 权重，
训练一个简单的医疗问答助手，
并比较微调前后的回答差异。

- 重点不是训练一个真正可用的医疗模型。
- 重点是观察领域数据如何影响模型输出。
- 建议记录同一 prompt 在微调前后的回答差异。

```
trainer/train_full_sft.py
trainer/train_lora.py
dataset/lm_dataset.py -> SFTDataset
```

使用 Vibe Coding 辅助实验

Vibe Coding:

让 AI 帮助我们阅读代码、定位问题、
生成调试脚本、解释报错和修改实验代码。

工具	适合场景
Codex	在代码仓库中阅读、修改、调试项目
Claude Code	长上下文代码阅读、重构、调试
Cursor	IDE 内补全、聊天、代码修改
GitHub Copilot	日常补全、函数生成、单文件修改
ChatGPT	解释概念、生成脚本、辅助写报告

使用 AI 工具不是偷懒，关键是你要能判断它说得对不对，并且能解释最终代码为什么这样改。

从 MiniMind 到成熟训练框架

个人和研究场景常用

框架	适合用途
Transformers / Trainer	快速微调、推理、模型加载
TRL	SFT、DPO、PPO 等对齐训练
PEFT	LoRA、QLoRA 等参数高效微调
Lightning	训练循环、日志、checkpoint
Accelerate	单机 / 多卡启动和混合精度

工业界和大规模训练常用

框架	典型用途
Megatron-LM	张量并行、流水并行预训练
DeepSpeed	ZeRO 优化、显存节省
FSDP	PyTorch 原生参数分片
verl	大模型 RLHF / PPO / GRPO
Colossal-AI	并行训练和显存优化
Ray	分布式调度、rollout、服务编排

致 谢

- 胡玥、曹亚男、方芳：国科大《自然语言处理基础》
- 曹亚男、任昱冰：国科大《深度学习与自然语言处理概述》


感谢国科大2023级硕士生罗天宇同学对本节课的贡献
THANKS
https://ictkc.github.io/teaching/2026spring-nlp