MLX-LM-LoRA: Algorithms and Python API

Purpose

This is an implementation-grounded guide to the training algorithms in this repository. It is written for Python and notebook users as well as CLI users. The sections below are based on the checked-out source and tests, with emphasis on actual tokenization, masking, loss computation, dataclasses, defaults, and direct trainer calls.

Repository architecture

The public orchestration layer is `mlx_lm_lora/train.py`. It merges YAML and CLI configuration, loads models and tokenizers, installs LoRA or DoRA adapters, creates optimizers, loads datasets, and dispatches to the algorithm trainers. The implementation modules are under `mlx_lm_lora/trainer/`.

The common training flow is:

1. Load the policy model and tokenizer.
2. Optionally load a quantized base model and replace supported linear layers with LoRA or DoRA adapters.
3. Load and tokenize the dataset using the expected schema.
4. Construct masks so only intended next-token targets contribute to the loss.
5. Run the policy, and where required a frozen reference model or judge model.
6. Compute a scalar loss and diagnostic metrics.
7. Run MLX autodiff, gradient accumulation, and an optimizer update.
8. Optionally apply QAT projection.
9. Report, evaluate, save adapters, and export.

Shared configuration defaults

Important defaults in `train.py` include:

- `train_type: lora`
- `train_mode: sft`
- `batch_size: 4`
- `seed: 0`
- `max_seq_length: 2048`
- `learning_rate: 1e-5`
- `optimizer: adam`
- `val_batches: 25`

- `steps_per_report: 10`
- `steps_per_eval: 200`
- `save_every: 100`
- `adapter_path: adapters`
- `lora_parameters.rank: 8`
- `lora_parameters.dropout: 0.0`
- `lora_parameters.scale: 10.0`
- `mask_prompt: false`
- `fuse: true`

Inherited `SFTTrainingArgs` fields are:

- `batch_size=4`
- `iters=100`
- `gradient_accumulation_steps=1`
- `val_batches=25`
- `steps_per_report=10`
- `steps_per_eval=200`
- `steps_per_save=100`
- `max_seq_length=2048`
- `adapter_file="adapters.safetensors"`
- `grad_checkpoint=False`
- `seq_step_size=None`
- `qat_enable=False`
- `qat_bits=8`
- `qat_group_size=64`
- `qat_mode="affine"`
- `qat_start_step=1`
- `qat_interval=1`

CLI values override YAML values. Python and YAML use snake_case; CLI flags generally use kebab-case.

Parameterization modes

LoRA

LoRA freezes the pretrained model and replaces supported linear layers with trainable low-rank adapters. Conceptually, the adapted weight is:

```
W' = W + (scale / rank) * B @ A
```

The main controls are `rank`, `dropout`, `scale`, and `num_layers`. Use a smaller rank for narrow tasks or memory-constrained runs. Increase it only when validation shows underfitting.

DoRA

DoRA decomposes the weight magnitude and direction and applies a low-rank directional update. It uses the same general model-construction and trainer paths as LoRA, but usually adds more adapter-side state. Keep data, seed, target layers, and schedule fixed when comparing DoRA with LoRA.

Full tuning

Full tuning updates all eligible model parameters. It offers the most capacity but costs substantially more memory and is more sensitive to learning rate, overfitting, and catastrophic forgetting.

QLoRA and QAT

QLoRA-style operation loads the base model in 4-bit, 6-bit, 8-bit, or mixed FP4 form while training adapters. QAT is separate: for SFT, DPO, and ORPO, the implementation uses a straight-through fake-quantization hook and scheduled projection. Its controls are `qat_enable`, `qat_bits`, `qat_group_size` (`0` or less means per-tensor), `qat_mode` (currently `affine`), `qat_start_step`, and `qat_interval`.

SFT and notebook baseline

SFT maximizes target-token likelihood. It supports `nll`, memory-bounded `chunked_nll`, and `dft` loss types. Use `mask_prompt=True` for response-only instruction tuning. Inspect truncation and token masks before launching a long run.

```
from mlx_lm import load
from mlx_lm_lora.trainer.datasets import load_dataset
from mlx_lm_lora.trainer.sft_trainer import SFTTrainingArgs, train_sft

model, tokenizer = load("model-id")
train_set, valid_set, test_set = load_dataset("./data")
args = SFTTrainingArgs(
    loss_type="nll",
    batch_size=2,
    iters=500,
    max_seq_length=2048,
    gradient_accumulation_steps=4,
    mask_prompt=True,
)
train_sft(model, tokenizer, train_set, valid_set, args=args)
```

Best practices:

- Run a 2-5 iteration smoke test first.

- Verify the chat template, sequence lengths, padding, and masks.
- Use `chunked_nll`, `seq_step_size`, or efficient long-context processing when memory is the bottleneck.
- Keep a held-out validation set and a small manual generation set.
- Compare DFT with NLL on identical data and seeds.

Offline Preference Optimization

This guide documents the repository implementations of DPO, CPO, ORPO, and FTPO/Antidoom. It follows the trainer modules, dataset adapters, CLI dispatch, and focused tests.

Shared behavior

DPO, CPO, and ORPO consume preference pairs with chosen and rejected responses. Dataset adapters tokenize alternatives with the model tokenizer and chat template. Autoregressive scoring passes tokens[:, :-1] to the model and scores tokens[:, 1:].

DPO and CPO currently score every valid token in the concatenated prompt-plus-response sequence. ORPO uses response-only masks. FTPO scores only the single next-token distribution after a supplied context.

All preference trainers inherit SFTTrainingArgs:

Field	Default
batch_size	4
iters	100
gradient_accumulation_steps	1
val_batches	25
steps_per_report	10
steps_per_eval	200
steps_per_save	100
max_seq_length	2048
adapter_file	adapters.safetensors
grad_checkpoint	False
seq_step_size	None
qat_enable	False
qat_bits	8
qat_group_size	64
qat_mode	affine
qat_start_step	1
qat_interval	1

The CLI defaults additionally include learning rate 1e-5, Adam optimization, LoRA training, seed 0, adapter directory adapters, and maximum sequence length 2048.

Direct Preference Optimization (DPO)

DPO trains a policy directly from preference pairs without a reward model. It compares the policy's relative likelihood of chosen and rejected sequences against a frozen reference model.

Data flow

DPODataset expects:

```
{"prompt": "Question", "chosen": "Preferred answer", "rejected": "Rejected answer"}
{"system": "You are helpful", "prompt": "Question", "chosen": "Good", "rejected": "Bad"}
```

Each row becomes two independently rendered conversations, one ending with the chosen assistant response and one with the rejected response. The processed item is:

```
{"chosen": list[int], "rejected": list[int]}
```

Batches are sorted by chosen-sequence length, truncated to max_seq_length, and dynamically padded. The iterator requires the global batch size to be divisible by the MLX distributed worker count.

Objective

Let s_pi_c and s_pi_r be policy sequence scores and s_ref_c and s_ref_r the corresponding reference scores. The preference logit is:

```
z = (s_pi_c - s_pi_r) - (s_ref_c - s_ref_r)
```

Supported loss types are:

```
sigmoid: -log_sigmoid(beta * z)
hinge:   relu(1 - beta * z)
ipo:     (z - 1 / (2 * beta)) ** 2
dpop:    -log_sigmoid(beta * z)
         + delta * max(0, s_ref_c - s_pi_c)
```

For ipo, compute_score divides summed token scores by mask.sum(-1); other losses use raw sums. delta is used only by dpop.

Configuration and Python API

```
@dataclass
class DPOTrainingArgs(SFTTrainingArgs):
    beta: float = 0.1
    loss_type: str = "sigmoid"
    delta: float = 50.0
    reference_model_path: str | None = None
```

CLI options:

```
--train-mode dpo
--beta 0.1
--dpo-cpo-loss-type {sigmoid,hinge,ipo,dpop}
--delta 50.0
--reference-model-path PATH
```

efficient-long-context enables cached chunking with an internal chunk size of 512.

In the CLI workflow, the reference model is frozen automatically. If reference-model-path is omitted, a separate frozen copy is loaded from model. Direct train_dpo calls may pass ref_model=None; the implementation then substitutes zero reference scores.

```
from mlx_lm_lora.trainer.dpo_trainer import DPOTrainingArgs, train_dpo

args = DPOTrainingArgs(
    beta=0.1, loss_type="sigmoid", delta=50.0,
    batch_size=2, iters=500,
)

train_dpo(
    model=policy_model,
    ref_model=frozen_reference_model,
    optimizer=optimizer,
    train_dataset=train_dataset,
    val_dataset=valid_dataset,
    args=args,
    loss_type=args.loss_type,
)
```

The functional API is:

```
loss, reward, num_tokens, metrics = dpo_loss(
    policy_chosen_score,
    policy_rejected_score,
    reference_chosen_score,
    reference_rejected_score,
    chosen_masks,
    rejected_masks,
    beta=0.1,
    delta=50.0,
    loss_type="sigmoid",
)
```

Metrics are accuracies, margins, policy_rejected_logps, policy_chosen_logps, rejected_logits_mean, and chosen_logits_mean. Accuracy compares batch-level mean chosen and rejected rewards rather than producing one decision per sample.

Important source caveat: the CLI DPO branch constructs DPOTrainingArgs(loss_type=...) but calls train_dpo without its separate loss_type keyword. Since train_dpo defaults that keyword to sigmoid, non-sigmoid CLI selections should be verified; direct callers should pass loss_type explicitly.

Best practices:

- Use a frozen reference model from the same base checkpoint.
- Keep beta modest initially.
- Use ipo when sequence-length normalization is important.
- Tune dpop delta together with beta.
- Validate chosen/rejected ordering and chat-template rendering.
- Monitor reward margins and chosen/rejected log-probabilities.

Contrastive Preference Optimization (CPO)

CPO removes the reference-model correction and directly optimizes the policy score difference between chosen and rejected responses.

The preference logit is:

```
z = s_pi_c - s_pi_r
```

It supports the same sigmoid, hinge, IPO, and DPOP loss family. Its DPOP form uses:

```
-log_sigmoid(beta * z) + delta * max(0, s_pi_r - s_pi_c)
```

CPO uses the same DPODataset format as DPO.

CPOTrainingArgs is an alias of DPOTrainingArgs, exposing beta=0.1, loss_type=sigmoid, delta=50.0, and reference_model_path=None. The reference field is unused by CPO.

CLI options:

```
--train-mode cpo
--beta 0.1
--dpo-cpo-loss-type {sigmoid,hinge,ipo,dpop}
--delta 50.0
```

Python usage:

```
from mlx_lm_lora.trainer.cpo_trainer import CPOTrainingArgs, train_cpo

train_cpo(
    model=policy_model,
    optimizer=optimizer,
    train_dataset=train_dataset,
    val_dataset=valid_dataset,
    args=CPOTrainingArgs(
        beta=0.1, loss_type="sigmoid",
        batch_size=2, iters=500,
    ),
)
```

The functional APIs are:

```
cpo_loss_from_model(
    model, chosen, rejected, chosen_masks, rejected_masks,
    beta=0.1, delta=50.0, loss_type="sigmoid",
)

cpo_loss(
    policy_chosen_score, policy_rejected_score,
    chosen_masks, rejected_masks,
    beta=0.1, delta=50.0, loss_type="sigmoid",
)
```

Both return (loss, reward, num_tokens, metrics). Metrics are accuracies, margins, policy_rejected_logps, policy_chosen_logps, and chosen_logits_mean. The implementation reports the raw rejected score in policy_rejected_logps but normalizes the chosen score by chosen token count.

Best practices:

- Use CPO when a reference model is unavailable or intentionally undesired.
- Consider ipo when raw sequence sums would favor longer examples.
- Be conservative with learning rate because CPO has no reference anchor.
- Use cached long-context mode for long examples.

Odds Ratio Preference Optimization (ORPO)

ORPO is a reference-free monolithic objective combining chosen-response supervised loss with an odds-ratio preference term.

Data and masking

ORPODataset accepts prompt (or question as fallback), string/dictionary/message-list responses, optional system, and optional preference_score:

```
{"prompt": "Question", "chosen": "Good answer", "rejected": "Bad answer"}
{"prompt": "Question", "chosen": "Good", "rejected": "Bad", "preference_score": 0.75}
{"prompt": "Question", "chosen": {"messages": [...]}, "rejected": {"messages": [...]}}
```

Missing preference scores default to 1.0; explicit values are converted with float(). Each row stores chosen/rejected token IDs and prompt lengths. Batches are length-sorted, padded to a width rounded up to a multiple of eight, and mask only response positions. get_logps intersects adjacent masks, preventing right padding from becoming a next-token target.

Objective

get_logps clips token log-probabilities to [-1000, 0] and returns average sequence log-probabilities. With chosen score s_c, rejected score s_r, and preference score q, the implementation sets:

```
s_tilde_c = s_c * q
log_odds = (s_tilde_c - s_r)
           - (log(1 - exp(s_tilde_c)) - log(1 - exp(s_r)))
loss = -s_c - beta * log_sigmoid(log_odds)
```

This is chosen-response SFT NLL plus a beta-weighted odds-ratio term. Non-finite log-probability inputs are sanitized before calculation.

```
@dataclass
class ORPOTrainingArgs(SFTTrainingArgs):
    beta: float = 0.1
    reward_scaling: float = 1.0
```

reward_scaling is explicitly not implemented and is unused.

CLI:

```
--train-mode orpo
--beta 0.1
--reward-scaling 1.0
```

Python usage:

```
from mlx_lm_lora.trainer.orpo_trainer import ORPOTrainingArgs, train_orpo

train_orpo(
    model=policy_model,
    optimizer=optimizer,
    train_dataset=train_dataset,
    val_dataset=valid_dataset,
    args=ORPOTrainingArgs(beta=0.1, batch_size=2, iters=500),
)
```

The model-level API is:

```
orpo_loss_from_model(
    model, chosen, rejected, chosen_masks, rejected_masks,
    preference_scores, beta=0.1,
)
```

The functional API is:

```
orpo_loss(
    chosen_logps, chosen_logits_mean,
    rejected_logps, rejected_logits_mean,
    chosen_masks, rejected_masks,
    preference_scores, beta=0.1,
)
```

Both return (loss, reward, num_tokens, metrics). Metrics are accuracies, margins, policy_chosen_logps, policy_rejected_logps, chosen_logits_mean, and rejected_logits_mean.

Best practices:

- Preserve exact chat-template prompt offsets.
- Use preference scores consistently.
- Start with beta=0.1 and adjust after checking margins.
- Do not tune reward_scaling expecting behavioral changes until implemented.
- Inspect response-only metrics rather than interpreting logits means as probabilities.

Final Token Preference Optimization (FTPO / Antidoom)

FTPO is designed to repair repetition or reasoning “doom loops.” It operates on the final next-token distribution after a supplied context rather than scoring complete chosen and rejected sequences.

Dataset format

Each Antidoom row must contain:

```
{
  "context_with_chat_template": "...",
  "rejected_decoded": "...",
  "multi_chosen_decoded": ["...", "..."]
}
```

FTPODataset keeps only rows with a non-empty prompt fitting max_seq_length, a rejected string encoding to exactly one token, and at least one distinct chosen one-token alternative. It deduplicates chosen IDs and removes the rejected token. The processed row is:

```
{
    "prompt_ids": list[int],
    "chosen_ids": list[int],
    "rejected_token_id": int,
}
```

The iterator length-sorts prompts, pads prompts within the batch, pads variable chosen lists, and requires at least batch_size valid rows.

Forward pass and objective

For a prompt of length L, _last_logits selects the model output at position L-1, which predicts the token immediately following the context. Policy and frozen-reference logits have shape [batch, vocabulary].

For each chosen token c and rejected token r:

```
delta = policy_logit(c) - policy_logit(r)
weight = clip((clip_epsilon_logits - delta) /
              clip_epsilon_logits, 0, 1)
```

The preference term is the batch mean of softplus(clip_epsilon_logits - delta) * weight, normalized by chosen-token count. Once a chosen token exceeds the rejected token by at least clip_epsilon_logits, its preference weight becomes zero.

Let:

```
diff = policy_logits - stop_gradient(reference_logits)
```

The non-target MSE averages squared diff over vocabulary entries excluding chosen and rejected targets. The target-excess MSE averages:

```
max(abs(diff) - tau_mse_target, 0) ** 2
```

over chosen and rejected targets. The final objective is:

```
loss = pref_loss
       + lambda_mse * mse_elem
       + lambda_mse_target * mse_target
```

```
@dataclass
class FTPOTrainingArgs(SFTTrainingArgs):
    lambda_mse_target: float = 0.05
    tau_mse_target: float = 1.0
    lambda_mse: float = 0.4
    clip_epsilon_logits: float = 2.0
```

CLI:

```
--train-mode ftpo
--lambda-mse-target 0.05
--tau-mse-target 1.0
--lambda-mse 0.4
--clip-epsilon-logits 2.0
--reference-model-path PATH
```

train_ftpo requires a non-None frozen reference model. The CLI creates one automatically for FTPO.

```
from mlx_lm_lora.trainer.ftpo_trainer import FTPOTrainingArgs, train_ftpo

train_ftpo(
    model=policy_model,
    ref_model=frozen_reference_model,
    optimizer=optimizer,
    train_dataset=train_dataset,
    val_dataset=valid_dataset,
    args=FTPOTrainingArgs(
        lambda_mse_target=0.05,
        tau_mse_target=1.0,
        lambda_mse=0.4,
        clip_epsilon_logits=2.0,
        batch_size=4,
        iters=500,
    ),
)
```

The functional API is:

```
loss, samples, metrics = ftpo_loss(
    policy_logits,
    reference_logits,
    chosen_ids,
    chosen_mask,
    rejected_ids,
    lambda_mse_target=0.05,
    tau_mse_target=1.0,
    lambda_mse=0.4,
    clip_epsilon_logits=2.0,
)
```

Metrics are pref_loss, mse_elem, mse_tgt_tokenwise, chosen_win, margin_win, mean_delta, and active_weight.

Best practices:

- Use the exact inference chat template to create contexts.
- Verify retained row counts after Antidoom preprocessing.
- Keep the reference model frozen and tokenizer-compatible.
- Start with repository defaults.
- Increase lambda_mse when unrelated vocabulary behavior drifts.
- Inspect chosen_win, margin_win, and active_weight.

Source and test coverage

Implementation references:

- mlx_lm_lora/trainer/dpo_trainer.py
- mlx_lm_lora/trainer/cpo_trainer.py
- mlx_lm_lora/trainer/orpo_trainer.py
- mlx_lm_lora/trainer/ftpo_trainer.py
- mlx_lm_lora/trainer/datasets.py
- mlx_lm_lora/train.py

Focused tests cover supported loss variants, finite numerical behavior, token filtering, padding, response masking, preference-score handling, distributed batch constraints, and FTPO margin behavior in tests/test_training_algorithms.py and tests/test_datasets.py.

GRPO, GSPO, Dr. GRPO, and DAPO

The repository implements the GRPO family through one `train_mode: "grpo"` path. The variants are selected with additional settings:

Variant	Configuration
GRPO	`grpo_loss_type="grpo"`
GSPO-style training	`importance_sampling_level="sequence"`
Dr. GRPO	`grpo_loss_type="dr_grpo"`
DAPO-style clipping	Set `epsilon_high`

Training flow

For every prompt, the trainer samples `group_size` completions. With the default `group_size=4`, each prompt produces four candidate answers.

The rollout process is:

1. Tokenize a batch of prompts.
2. Duplicate each prompt `group_size` times.
3. Generate completions with the configured sampler.

4. Score every completion with each reward function.
5. Combine reward components using `reward_weights`.
6. Normalize rewards within each prompt group.
7. Compute policy/reference log-probability ratios.
8. Apply clipped policy optimization and the optional KL penalty.
9. Update the trainable parameters.

`batch_size` is the number of prompts, not the number of completions. A rollout therefore contains approximately `batch_size × group_size` completions. In distributed training, `batch_size` must be divisible by the number of workers.

The default generation terminator is `"</answer>"`. If the tokenizer supports it, the trainer adds this as an EOS token; otherwise it removes a trailing occurrence after generation.

Rewards and grouped advantages

For completion `i`, the trainer computes one score from every reward function:

```
r_i = sum_j(w_j * r_i,j)
```

If no weights are supplied, every reward function receives weight `1.0`. Weights are not normalized automatically, so their scale directly affects the total reward and advantage magnitude.

Rewards are normalized separately for each prompt group:

```
A_i = (r_i - mean(group_rewards)) / (std(group_rewards) + 1e-4)
```

A prompt with only one completion receives advantage `0.0`. Thus `group_size=1` provides no useful policy-gradient signal and should normally be reserved for evaluation or debugging.

Missing reward values behave as follows:

- A reward function returning `None` contributes `NaN`.
- `NaN` components are treated as zero in the weighted total.
- If every reward function returns `None` for a completion, training raises `RuntimeError`.
- The number of `reward_weights` must exactly match the number of reward functions.

The trainer reports total and grouped reward statistics, per-function mean/std/coverage, KL, completion lengths, and clipping ratios.

GRPO loss

The implementation computes a policy/reference log-ratio for every valid generated token:

```
log_ratio(i,t) = log pi_theta(y_i,t) - log pi_ref(y_i,t)
```

The ratio is clipped to:

```
[1 - epsilon, 1 + epsilon_high]
```

The loss uses the pessimistic objective:

```
-min(rho * A, clip(rho) * A)
```

When `beta` is non-zero, a reference KL penalty is also added. With `beta=0`, the KL term is omitted from the optimization objective, although KL metrics are still calculated.

GRPO-specific settings

The current `GRPOTrainingArgs` defaults are:

Setting	Default
group_size	4
beta	0.1
epsilon	1e-4
epsilon_high	None
max_completion_length	512
reference_model_path	None
temperature	0.8
top_p	0.95
top_k	20
min_p	0.0
grpo_loss_type	"grpo"
reward_weights	None
importance_sampling_level	"token"
end_answer_token	"</answer>"

Inherited training defaults include `batch_size=4`, `gradient_accumulation_steps=1`, `val_batches=25`, `steps_per_report=10`, `steps_per_eval=200`, `steps_per_save=100`, `max_seq_length=2048`, `grad_checkpoint=False`, `learning_rate=1e-5`, `optimizer="adam"`, `seed=0`, `num_layers=-1`, LoRA rank `8`, dropout `0.0`, scale `10.0`, `adapter_path="adapters"`, `test=False`, `test_batches=500`, and `fuse=True`. `GRPOTrainingArgs` defaults `iters` to `100`; the CLI default is `None` and can derive iterations from `epochs`.

GSPO-style sequence importance sampling

There is no separate `"gspo"` training mode. Use:

```
train_mode: grpo
importance_sampling_level: sequence
```

At the default token level, each generated token has its own importance ratio. At the sequence level, the trainer averages valid-token log-ratios for each completion, excludes padding, and broadcasts that sequence value across the completion before clipping.

The helper accepts only `"token"` and `"sequence"`; unsupported values raise `ValueError`. Tests verify that sequence averaging ignores masked padding and that token-level gradients are preserved.

Token-level sampling is the default. Compare sequence-level runs using the same seed, reward functions, group size, and generation settings.

Dr. GRPO

Use:

```
train_mode: grpo
grpo_loss_type: dr_grpo
```

The current implementation changes the loss denominator. Base GRPO divides by the number of valid generated tokens. Dr. GRPO divides by the number of generated samples multiplied by the configured `max_completion_length`:

```
loss = masked_loss.sum() / (
    number_of_samples * max_completion_length
)
```

This makes the denominator independent of sampled completion lengths, while the configured completion limit still affects both generation and loss scaling. The source also accepts `grpo_loss_type="bnpo"`, but its current denominator is effectively the same valid-token denominator as base GRPO with a `max(..., 1.0)` safeguard.

DAPO-style dual clipping

There is no separate DAPO mode. Configure asymmetric clipping:

```
train_mode: grpo
epsilon: 0.0001
epsilon_high: 0.01
```

`epsilon` controls the lower boundary and `epsilon_high` controls the upper boundary. If `epsilon_high` is omitted, it falls back to `epsilon`, giving symmetric clipping. The source uses a truthiness check, so an explicit value of `0` also falls back to `epsilon`.

Dataset format

GRPO expects rows containing `prompt` and `answer`, with optional `system` and `type` fields:

```
{
  "prompt": "What is 2 + 2?",
  "answer": "4",
  "system": "Solve the problem and use <answer> tags.",
  "type": "arithmetic"
}
```

`GRPODataset` converts prompt and answer values to strings, applies the tokenizer chat template, adds a system message and generation prompt, and preserves prompt, answer, and type metadata for rewards. If `system` is absent, it asks the model to place reasoning in `<think>` and the solution in `<answer>`.

Built-in reward functions

The registry currently contains:

- `r1_accuracy_reward_func`: extracts `<answer>...</answer>` and returns `2.0` for an exact match with `answer`, otherwise `0.0`.
- `r1_int_reward_func`: returns `0.5` when the extracted answer is a non-empty sequence of digits.
- `r1_strict_format_reward_func`: returns `0.5` for the strict `<think> ... </think> <answer> ... </answer>` pattern.
- `r1_soft_format_reward_func`: returns `0.5` when both sections are present, non-empty, and correctly ordered.
- `r1_count_xml`: awards `0.125` for each correctly occurring XML marker, up to `0.5`, and penalizes trailing text.

Use the exact registry names:

```
from mlx_lm_lora.trainer.grpo_reward_functions import (
    get_default_reward_functions,
    get_reward_function,
    list_available_reward_functions,
)
```

Custom rewards

A custom reward must accept the keyword arguments used by the trainer and return one value per completion:

```
from mlx_lm_lora.trainer.grpo_reward_functions import register_reward_function

@register_reward_function()
def exact_length_reward(prompts, completions, answer, types=None):
    return [
        1.0 if 1 <= len(completion.split()) <= 50 else 0.0
        for completion in completions
    ]
```

Use the parameter name `answer`, not `answers`, because the trainer calls it by keyword. Always return one value per completion, include `types=None` for compatibility, and handle empty, malformed, truncated, and adversarial completions. Keep reward scales stable and interpretable.

For CLI usage, a file can register rewards and then be selected by name:

```
mlx_lm_lora.train \
  --train \
  --train-mode grpo \
  --reward-functions-file ./my_rewards.py \
  --reward-functions "exact_length_reward"
```

The CLI parses `--reward-weights` as a string such as `"[0.8,0.2]"`; the adapter strips brackets and splits on commas. Direct Python calls should pass a normal list such as `[0.8, 0.2]`.

Python and notebook API

The direct API uses trainer modules rather than the CLI:

```
from pathlib import Path

import mlx.optimizers as optim
from datasets import load_dataset

from mlx_lm_lora.utils import from_pretrained
from mlx_lm_lora.trainer.datasets import CacheDataset, GRPODataset
from mlx_lm_lora.trainer.grpo_trainer import GRPOTrainingArgs, train_grpo
from mlx_lm_lora.trainer.grpo_reward_functions import (
    r1_accuracy_reward_func,
    r1_soft_format_reward_func,
)

model, tokenizer, adapter_file = from_pretrained(
    model="mlx-community/model",
    new_adapter_path="./grpo-adapter",
    lora_config={
        "rank": 8,
        "dropout": 0.0,
        "scale": 10.0,
        "use_dora": False,
        "num_layers": 8,
    },
)

ref_model, _, _ = from_pretrained(model="mlx-community/model")
ref_model.freeze()

dataset = load_dataset("org/grpo-dataset")
train_set = CacheDataset(GRPODataset(dataset["train"], tokenizer))
valid_set = CacheDataset(GRPODataset(dataset["valid"], tokenizer))

optimizer = optim.AdamW(learning_rate=1e-5)

args = GRPOTrainingArgs(
    batch_size=2,
    iters=100,
    max_seq_length=2048,
    max_completion_length=256,
    group_size=4,
    temperature=0.8,
    beta=0.1,
    epsilon=1e-4,
```

```
    importance_sampling_level="token",
    grpo_loss_type="grpo",
    reward_weights=[1.0, 0.25],
    adapter_file=Path("./grpo-adapter/adapters.safetensors"),
)

train_grpo(
    model=model,
    ref_model=ref_model,
    tokenizer=tokenizer,
    optimizer=optimizer,
    train_dataset=train_set,
    val_dataset=valid_set,
    reward_funcs=[r1_accuracy_reward_func, r1_soft_format_reward_func],
    args=args,
)
```

`from_pretrained()` returns three values: model, tokenizer, and an optional adapter-file path. `evaluate_grpo()` returns `(avg_loss, token_count, metrics)` and requires explicit sampling arguments such as `top_p`, `top_k`, and `min_p`.

Recommended practice

- Use `group_size >= 2`; the default `4` is a practical starting point.
- Test rewards independently on correct, incorrect, empty, malformed, and truncated outputs.
- Keep reward magnitudes comparable before assigning weights.
- Use a frozen base-model reference unless a separate reference is intentional.
- Begin with short completions and a small iteration budget.
- Track reward coverage, grouped reward standard deviation, KL, clipping ratios, completion lengths, and the fraction hitting the generation limit.
- Compare token- and sequence-level importance sampling under identical conditions.
- Use Dr. GRPO when completion-length-dependent normalization is undesirable.
- Use a modest `epsilon_high` for DAPO-style asymmetric clipping.
- Maintain a held-out set and inspect generated responses, not only scalar rewards.

Source-specific caveats

The checked-out implementation has several documentation/API discrepancies:

- The CLI parser defaults `temperature` to `1.0`, while `GRPOTrainingArgs` and `CONFIG_DEFAULTS` specify `0.8`.
- The CLI has no `--top-p`, `--top-k`, or `--min-p` options; direct trainer defaults are used during training.
- The README's custom-reward import path and function signature do not match the current source.
- YAML `reward_weights` lists are incompatible with the current CLI adapter, which expects a string and calls `.strip(...)`.

- `grpo_loss` calculates log-probabilities from generated completion tokens directly; the current implementation does not concatenate the original prompt before that calculation.
- The checked-in notebook uses stale conventions: `from_pretrained()` returns three values, and `evaluate_grpo()` requires explicit sampling arguments.

The focused test run passed all 83 tests covering datasets, reward registration and behavior, grouped advantages, loss variants, and importance sampling.

Online DPO, XPO, RLHF REINFORCE, and PPO

This draft is aligned with the checked-out repository source and tests.

Shared online-training pipeline

Online DPO, XPO, RLHF REINFORCE, and PPO use `PromptDataset` from `mlx_lm_lora/trainer/datasets.py`. Each record must contain `prompt`:

```
{"prompt": [{"role": "user", "content": "Question"}]}
```

or:

```
{"prompt": "Question"}
```

The dataset applies the tokenizer's chat template and stores tokenized `prompt` plus rendered `prompt_text`.

For each prompt, the trainer:

1. Generates two completions from the policy.
2. Sends them to a human or LLM judge.
3. Converts the judgment into chosen/rejected responses.
4. Concatenates the rendered prompt with each completion.
5. Scores the complete sequences under the policy and frozen reference.
6. Computes the mode-specific objective and updates the policy/adapters.

Online modes currently use all-one masks for the full prompt-plus-completion sequence. They do not implement response-only masking. The online batch iterator sorts prompts by length, requires `batch_size` to be divisible by the distributed world size, and drops an incomplete final batch. Its `max_seq_length` parameter is currently not used to truncate prompts.

Exact defaults

All four argument classes inherit these fields from `SFTTrainingArgs`:

Setting	Direct Python default
`batch_size`	4
`iters`	100
`gradient_accumulation_steps`	1
`val_batches`	25
`steps_per_report`	10
`steps_per_eval`	200
`steps_per_save`	100
`max_seq_length`	2048
`adapter_file`	`adapters.safetensors`
`grad_checkpoint`	`False`
`seq_step_size`	`None`

Mode-specific defaults:

Mode	Defaults
Online DPO	`beta=0.1`, `loss_type="sigmoid"`, `delta=50.0`, `temperature=0.8`, `judge="human"`, `max_completion_length=512`, `reference_model_path=None`
XPO	Online DPO defaults plus `alpha=[1e-5]`
RLHF REINFORCE	`beta=0.1`, `judge=None`, `reference_model_path=None`, `max_completion_length=128`
PPO	Online DPO defaults plus `epsilon=0.2`

The CLI has different effective defaults:

- `--temperature` defaults to `1.0`, not `0.8`.
- `--epsilon` defaults to `1e-4`, not `0.2`.
- `--max-completion-length` defaults to `512`, including RLHF.
- `--alpha` defaults to `[1e-5]`.
- `--iters` must be supplied, or derived from `--epochs`; there is no usable CLI iteration fallback.
- Online CLI modes require `--judge`.
- Without `--reference-model-path`, the CLI reloads the base model as a separate frozen reference.
- `judge_config` is expected to be a dictionary; currently `system_prompt` is the relevant key.

Online DPO

Implementation: `mlx_lm_lora/trainer/online_dpo_trainer.py`.

For chosen response `c` and rejected response `r`, the trainer computes:

```
delta = (policy_chosen - policy_rejected)
        - (reference_chosen - reference_rejected)
```

Scores are sums of token log-probabilities. For `loss_type="ipo"`, they are divided by the sequence mask count.

Supported losses are:

- `sigmoid`: `-log_sigmoid(beta * delta)`
- `hinge`: `relu(1 - beta * delta)`
- `ipo`: `(delta - 1 / (2 * beta)) ** 2`
- `dpop`: sigmoid loss plus a penalty when the policy under-scores the chosen response relative to the reference

Reported rewards are:

```
chosen_reward   = beta * (policy_chosen   - reference_chosen)
rejected_reward = beta * (policy_rejected - reference_rejected)
```

Generation produces exactly two completions per prompt with `max_completion_length`, temperature, `top_p=1.0`, `top_k=0`, `min_p=0.0`, disabled XTC, and newline/EOS special-token handling.

LLM pairwise judging randomly swaps candidate order, asks for `0` or `1`, and restores the original order. Invalid judge output returns `-1`, which the trainer's `else` branch treats as candidate 1 winning. Human judging is interactive and uses the same random swapping.

The README advertises `--judge human`, but the CLI loader currently attempts to load `human` as a model. Direct Python calls can use `judge_model="human"`.

Metrics include loss, chosen/rejected rewards, accuracy, margin, policy log-probabilities, score means, throughput, and peak memory.

XPO

Implementation: `mlx_lm_lora/trainer/xpo_trainer.py`.

XPO reuses Online DPO's generation, judging, reference scoring, and DPO loss, then adds:

```
exploration_bonus = alpha * (
    policy_chosen - reference_chosen
    + policy_rejected - reference_rejected
)
total_loss = dpo_loss - exploration_bonus
```

It reports `exploration_bonus`, `chosen_kl`, and `rejected_kl` in addition to the standard preference metrics.

`alpha` may be a schedule, for example:

```
XPOTrainingArgs(alpha=[0.0, 1e-6, 1e-5], iters=100)
```

The schedule is divided into equal iteration segments, with the final value used afterward. A schedule longer than the total iteration count can produce a zero-division error because the implementation computes `total_steps // len(alpha)`.

XPO generation currently omits `args.temperature`, so it uses the helper's default temperature of `0.8`.

RLHF REINFORCE

Implementation: `mlx_lm_lora/trainer/rlhf_reinforce_trainer.py`.

The trainer generates two completions and sends them to `LLMPPOJudge`, which expects JSON scores such as:

```
{
  "scores": [
    {"model_identifier": "0", "score": 7.5},
    {"model_identifier": "1", "score": 5.0}
  ]
}
```

Missing scores default to `0.5`; parsing errors use `[0.5, 1.0]`. RLHF always uses an LLM judge in the current source; it has no human-judge branch.

The intended loss is a scalar-reward policy gradient with KL regularization:

```
advantage = reward - beta * KL
loss = -advantage * sum(policy_log_probability)
```

The implementation concatenates prompt and completion, pads trajectories, shifts inputs and masks, and applies the reward/KL-weighted gradient.

Two source-level deviations are important:

1. `get_model_logits()` computes shifted `targets`, but `rlhf_reinforce_loss()` ignores them and labels cross-entropy with `argmax(policy_logits)`.
2. The loss uses selected policy/reference log-probability differences rather than full-vocabulary KL. The separate `compute_kl_penalty()` helper computes full KL but is not used by the loss.

`seq_step_size` is accepted but unused, and `max_seq_length` is not applied to generated trajectories.

Metrics are `rewards`, `kl_penalty`, `advantages`, `policy_logps`, and `ref_logps`.

There is also a test-path bug: `evaluate_rlhf_reinforce()` returns four values, while the RLHF test branch in `train.py` unpacks three.

PPO

Implementation: `mlx_lm_lora/trainer/ppo_trainer.py`.

The current implementation uses a pairwise judge, not a scalar reward model. It computes sequence log ratios against the frozen reference:

```
chosen_ratio   = exp(policy_chosen   - reference_chosen)
rejected_ratio = exp(policy_rejected - reference_rejected)
```

Advantages are `policy_chosen - policy_rejected`, normalized across the batch. Chosen and rejected surrogate objectives are clipped to `[1-epsilon, 1+epsilon]`. The final loss combines both clipped objectives with:

```
beta * (mean(chosen_log_ratio) + mean(rejected_log_ratio))
```

This is not a conventional old-policy PPO implementation: the frozen reference is the ratio denominator, and there is no separate old-policy snapshot or critic.

Metrics include `policy_loss`, `kl_penalty`, `advantages_mean`, `ratios_mean`, `clip_fraction`, policy/reference log-probabilities, accuracy, and margin.

Direct Python and notebook API

The repository does not ship the example notebooks; the README points to a separate notebook repository. Notebook cells can call the same Python API:

```
import mlx.core as mx
from mlx_lm import load

from mlx_lm_lora.utils import from_pretrained
from mlx_lm_lora.trainer.datasets import PromptDataset, CacheDataset
from mlx_lm_lora.trainer.online_dpo_trainer import (
    OnlineDPOTrainingArgs,
    train_online_dpo,
)

base_id = "model-id"
judge_id = "judge-model-id"

policy, tokenizer, adapter_file = from_pretrained(
    base_id,
    new_adapter_path="adapters",
    lora_config={"rank": 8, "dropout": 0.0, "scale": 10.0, "use_dora": False},
)

reference, _ = load(base_id)
reference.freeze()

judge, judge_tokenizer = load(judge_id)
judge.freeze()

rows = [
    {"prompt": [{"role": "user", "content": "Explain gravity."}]},
    {"prompt": "What is machine learning?"},
]

dataset = CacheDataset(PromptDataset(rows, tokenizer))
optimizer = mx.optimizers.Adam(learning_rate=1e-5)

args = OnlineDPOTrainingArgs(
    iters=20,
    batch_size=2,
    max_completion_length=128,
    beta=0.1,
    temperature=0.8,
    adapter_file=str(adapter_file),
)

train_online_dpo(
```

```
    model=policy,
    ref_model=reference,
    tokenizer=tokenizer,
    optimizer=optimizer,
    train_dataset=dataset,
    val_dataset=None,
    judge_config={},
    args=args,
    judge_model=judge,
    judge_tokenizer=judge_tokenizer,
)
```

Use the same setup with these mode-specific substitutions:

```
from mlx_lm_lora.trainer.xpo_trainer import XPOTrainingArgs, train_xpo
args = XPOTrainingArgs(alpha=[0.0, 1e-5], iters=20)
train_xpo(...)
```

```
from mlx_lm_lora.trainer.ppo_trainer import PPOTrainingArgs, train_ppo
args = PPOTrainingArgs(epsilon=0.2, iters=20)
train_ppo(...)
```

```
from mlx_lm_lora.trainer.rlhf_reinforce_trainer import (
    RLHFReinforceTrainingArgs,
    train_rlhf_reinforce,
)

args = RLHFReinforceTrainingArgs(
    iters=20,
    max_completion_length=128,
    beta=0.1,
)
train_rlhf_reinforce(...)
```

For direct human judging in Online DPO, XPO, or PPO, pass:

```
judge_model = "human"
judge_tokenizer = None
```

RLHF requires an actual judge model because it uses `LLMPPOJudge`.

Diagnostics and best practices

Before a long run:

- Establish a small SFT or offline DPO baseline.
- Run two to five iterations first.
- Inspect rendered prompts and both generated completions.
- Validate judge calibration and candidate-order robustness.
- Keep a held-out prompt set and evaluate generations, not only loss.
- Use a frozen reference from the same base checkpoint.
- Start with short completions and conservative learning rates.
- Track lengths because online `max_seq_length` is not enforced.
- Use gradient checkpointing or smaller batches when memory is unstable.
- Record model revisions, tokenizer, seed, judge prompt, sampling settings, objective parameters, and adapter checkpoints.

Mode-specific signals:

- Online DPO: accuracy, margin, chosen/rejected rewards, and invalid-judge rate.
- XPO: exploration bonus and both KL metrics.
- RLHF: reward, KL penalty, advantage, and generation quality.
- PPO: ratio mean and clip fraction; ratio explosion indicates excessive drift.
- All modes: inspect reward hacking, verbosity inflation, repetition, malformed answers, and general capability regression.

The relevant tests are in `tests/test_training_algorithms.py`. They cover defaults, finite losses, invalid loss types, batch preservation, XPO schedules, PPO clipping metrics, and RLHF tensor shifting. In the current environment, test collection did not complete because the Transformers import entered a Torch-loading stall.

Final operating checklist

Before a long run:

1. Inspect one raw example and one rendered/tokenized example.
2. Confirm the dataset schema matches the selected algorithm.
3. Check whether prompt tokens are included or masked.
4. Run a tiny training smoke test and save a checkpoint.
5. Inspect validation loss, algorithm-specific metrics, and generations.
6. Record model and tokenizer revisions, dataset revision, seed, all non-default hyperparameters, sampling settings, reward definitions, and the source commit.

For preference methods, monitor margins and chosen/rejected log-probabilities. For online methods, additionally monitor judge decisions, reward scale, KL, entropy, completion length, and evidence of reward hacking.

Source map

- mlx_lm_lora/train.py
- mlx_lm_lora/trainer/sft_trainer.py
- mlx_lm_lora/trainer/dpo_trainer.py
- mlx_lm_lora/trainer/cpo_trainer.py
- mlx_lm_lora/trainer/orpo_trainer.py
- mlx_lm_lora/trainer/ftpo_trainer.py
- mlx_lm_lora/trainer/grpo_trainer.py
- mlx_lm_lora/trainer/grpo_reward_functions.py
- mlx_lm_lora/trainer/online_dpo_trainer.py
- mlx_lm_lora/trainer/xpo_trainer.py
- mlx_lm_lora/trainer/rlhf_reinforce_trainer.py
- mlx_lm_lora/trainer/ppo_trainer.py
- mlx_lm_lora/trainer/datasets.py
- README.md
- tests/