Package: dragonfarm (via r-universe)

November 17, 2026

Title Fine-Tune Small Language Models with LoRA from R

Version 0.3.4

Description Fine-tune small (100M to 3B parameter) causal language models with LoRA (Low-Rank Adaptation) from R. Datasets are mapped to chat-format prompts and responses, training runs in a background 'Python' process built on Hugging Face 'transformers' and 'peft', and a 'shiny' app offers drag-and-drop dataset upload and column mapping. 'Python' dependencies are declared through 'reticulate' and resolved automatically on first use.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1)

Imports bslib (>= 0.6.0), cli, glue, jsonlite, plotly, processx, ps, reticulate (>= 1.41.0), rlang, shiny (>= 1.8.0), sortable, stats, tools, utils, withr, zip

Suggests arrow, ellmer, httr2, knitr, pkgload, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

SystemRequirements 'Python' (>= 3.10). The 'uv' tool is installed automatically by 'reticulate' to build the 'Python' environment.

URL https://github.com/tejas4patel/dragon-farm

BugReports https://github.com/tejas4patel/dragon-farm/issues

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Tejas Patel [aut, cre]

Maintainer Tejas Patel <algocrat@gmail.com>

Config/pak/sysreqs cmake make libicu-dev libpng-dev libuv1-dev libssl-dev python3 zlib1g-dev

Repository https://algocrat.r-universe.dev

Date/Publication 2026-10-12 13:31:00 UTC

RemoteUrl https://github.com/cran/dragonfarm

RemoteRef HEAD

RemoteSha 40b84c89b9f07061fae5825eca2cdec8751aa2f4

Contents

dragon_app . . . 3
dragon_archive_run . . . 4
dragon_archived_runs . . . 5
dragon_backend . . . 5
dragon_bundle . . . 6
dragon_cancel . . . 8
dragon_chat . . . 8
dragon_check . . . 10
dragon_code . . . 10
dragon_compare . . . 11
dragon_conversations . . . 12
dragon_dataset . . . 13
dragon_evaluate . . . 13
dragon_example_data . . . 14
dragon_export_gguf . . . 15
dragon_feedback . . . 15
dragon_generate . . . 16
dragon_hardware . . . 17
dragon_import . . . 18
dragon_judge . . . 18
dragon_llm_anthropic . . . 20
dragon_lora . . . 21
dragon_map . . . 22
dragon_map_pairs . . . 22
dragon_map_prompts . . . 23
dragon_merge . . . 24
dragon_metrics . . . 24
dragon_pipeline . . . 25
dragon_pipeline_cancel . . . 27
dragon_pipeline_status . . . 27
dragon_prefer . . . 28
dragon_presets . . . 30
dragon_preview . . . 30
dragon_prompts . . . 31
dragon_publish . . . 31
dragon_python_requirements . . . 32
dragon_reinforce . . . 33
dragon_remote . . . 35

dragon_remote_providers 35
dragon_resume 36
dragon_reward 36
dragon_run 38
dragon_runs 39
dragon_runs_dir 39
dragon_serve_ollama 40
dragon_split 41
dragon_status 41
dragon_step 42
dragon_synthesize 44
dragon_synthesize_pairs 46
dragon_train 48
dragon_train_args 49
dragon_wait 50
dragon_worker_stop 51

Index **52**

`dragon_app` *Launch the dragon-farm app*

Description

A Shiny app that walks through the same steps as the R API: drop in a dataset, drag its columns into prompt and response slots, pick a model, train in the background, watch the loss curve, try the result, talk to it with context, and run the whole post-training loop as a pipeline. Every run started here is a normal run directory, and the Monitor panel shows the R code that reproduces it.

Usage

```
dragon_app(runs_dir = dragon_runs_dir(), ...)
```

Arguments

`runs_dir`	Directory where runs are stored and listed.
`...`	Passed to `shiny::shinyApp()` as `options`.

Value

A Shiny app object. Printing it runs the app.

Examples

```
if (interactive()) {
dragon_app()
}
```

dragon_archive_run *Archive, restore, or delete a run*

Description

Archiving moves a run's directory under `archived/` in `runs_dir`. It disappears from `dragon_runs()` and the app's Runs list, but every file is kept; `dragon_unarchive_run()` moves it back exactly as it was. Deleting removes the run directory for good. Both refuse a run that is queued or running (cancel it first) and a run that another run continues from, unless `force = TRUE`.

Usage

```
dragon_archive_run(run, runs_dir = dragon_runs_dir(), force = FALSE)

dragon_unarchive_run(id, runs_dir = dragon_runs_dir())

dragon_delete_run(run, runs_dir = dragon_runs_dir(), force = FALSE)
```

Arguments

`run`	A `dragon_run`, run directory, or run id (with `runs_dir`).
`runs_dir`	Where runs live. Needed only when `run`/`id` is a bare id.
`force`	Archive or delete even if another run continues from this one.
`id`	An archived run's id, for `dragon_unarchive_run()`.

Value

The run id, invisibly.

Examples

```
if (interactive()) {
dragon_archive_run(run)
dragon_archived_runs()
dragon_unarchive_run(run$id)
dragon_delete_run("20260101-000000-old-experiment")
}
```

dragon_archived_runs *Archived runs*

Description

Runs that dragon_archive_run() moved out of the way. Same columns as dragon_runs().

Usage

```
dragon_archived_runs(runs_dir = dragon_runs_dir())
```

Arguments

runs_dir	Where runs live.

Value

A data frame, one row per archived run.

dragon_backend *Inference backends*

Description

Every function that generates text, dragon_generate(), dragon_chat(), the judge, teacher, and synthesis helpers, takes a backend. The default is the local worker; set options(dragonfarm.backend = ...) to change it for a session.

Usage

```
dragon_backend_local(keep_loaded = TRUE, device = "auto", dtype = "auto")

dragon_backend_server(url, model, api_key = NULL, headers = NULL)

dragon_backend_ollama(model, url = "http://localhost:11434")

dragon_backend()
```

Arguments

keep_loaded	Keep the worker and its models alive between calls.
device, dtype	Device and precision for the local worker. See dragon_hardware().
url	Base URL of the server, ending in /v1.
model	Model name as the server knows it.
api_key	Bearer token, if the server needs one. Defaults to DRAGONFARM_API_KEY, then OPENAI_API_KEY.
headers	Extra HTTP headers as a named character vector.

Details

- `dragon_backend_local()`: a Python worker on this machine that keeps the last two models loaded, so repeated calls pay the model load once. `keep_loaded = FALSE` stops it after every call.
- `dragon_backend_server()`: any server that speaks the OpenAI chat completions protocol. `url` is the base that ends in `/v1`. The model must already be available on that server; see `dragon_serve_ollama()` for the local case.
- `dragon_backend_ollama()`: a server backend for a model registered with Ollama on this machine.
- `dragon_backend()`: the session default.

Value

A `dragon_backend` object.

Examples

```
if (interactive()) {
options(dragonfarm.backend = dragon_backend_ollama("support-0.5b"))
dragon_generate(run, "My thermostat keeps dropping off Wi-Fi.")

vllm <- dragon_backend_server("https://my-pod.example.com/v1", model = "tejas/support-0.5b")
dragon_generate(run, "Hello", backend = vllm)
}
```

---

dragon_bundle *Package a run for a cloud GPU*

Description

Writes a run directory exactly as `dragon_train()` would, but instead of launching the trainer it zips everything a GPU machine needs: the data in chat format, the configuration, the trainer's Python code, and a notebook that runs it. Nothing is trained locally and Python is not needed.

Usage

```
dragon_bundle(
  dataset,
  model,
  lora = dragon_lora(),
  args = dragon_train_args(),
  name = NULL,
  run_dir = NULL,
  runs_dir = dragon_runs_dir(),
  n_samples = 10,
  revision = NULL,
```

```
  trust_remote_code = FALSE,
  method = NULL,
  beta = NULL,
  rewards = NULL,
  group_size = 4,
  temperature = 1,
  max_new_tokens = 128
)
```

Arguments

dataset	A mapped dragon_dataset (see dragon_map()).
model	A Hugging Face model id such as "HuggingFaceTB/SmolLM2-135M-Instruct", a local model directory, or a finished dragon_run to continue from. In the last case the earlier run's adapters are folded into the weights before this run adds its own, so stages chain: fine-tune, then dragon_prefer(), and so on. See dragon_presets() for model ids.
lora	LoRA settings from dragon_lora().
args	Training settings from dragon_train_args().
name	Short label used in the run id. Defaults to the model name.
run_dir	Exact directory to use. Defaults to a timestamped directory under runs_dir.
runs_dir	Parent directory for runs. See dragon_runs_dir().
n_samples	Number of held-out rows to generate sample replies for at the end of training.
revision	Model revision (branch, tag, or commit) on the Hub.
trust_remote_code	Allow the model repository to run custom code.
method	For a dataset mapped with dragon_map_pairs(), the preference method, "dpo" or "orpo". Defaults to "dpo". Ignored for prompt and response data.
beta	Preference strength for method, or the KL weight for an RL run. Defaults to 0.1 and 0.04 respectively.
rewards	For a dataset mapped with dragon_map_prompts(), the dragon_reward() list an RL run needs.
group_size, temperature, max_new_tokens	RL sampling settings. See dragon_reinforce().

Details

Continue with dragon_remote() to open a provider and see the steps, and dragon_import() to bring the results back into this run directory.

Value

A dragon_run whose state is "bundled".

Examples

```
if (interactive()) {
run <- dragon_dataset(dragon_example_data()) |>
  dragon_map(prompt = "{subject}\n\n{body}", response = "reply") |>
  dragon_bundle("Qwen/Qwen2.5-0.5B-Instruct")
dragon_remote(run, "colab")
# ... train in the browser, download the results zip ...
dragon_import(run, "~/Downloads/dragonfarm-results-<run id>.zip")
dragon_generate(run, "My thermostat keeps dropping off Wi-Fi.")
}
```

---

dragon_cancel *Cancel a run*

Description

Asks the trainer to stop after the current step and save a checkpoint. Falls back to killing the process if it does not stop in time.

Usage

```
dragon_cancel(run, timeout = 120)
```

Arguments

run	A dragon_run.
timeout	Seconds to wait for a clean stop.

Value

The run, invisibly.

---

dragon_chat *Talk to a model, with memory of the conversation*

Description

A conversation object that keeps the message history and sends all of it on every turn, so the model has context. Works with any backend: the local worker, Ollama, or a remote server. Transcripts use the same message format as training data, so a good conversation can become an example.

Usage

```
dragon_chat(
  x = NULL,
  system = NULL,
  backend = dragon_backend(),
  max_new_tokens = 256,
  temperature = 0.7,
  top_p = 0.9,
  base = FALSE,
  runs_dir = dragon_runs_dir(),
  context_window = NULL
)

dragon_chat_load(path, x = NULL, backend = dragon_backend())
```

Arguments

x	What answers: a dragon_run, adapter or model directory, or model id. Ignored by server backends, which serve a fixed model.
system	Optional system prompt, kept at the top of every turn.
backend	Where to run inference. See dragon_backend.
max_new_tokens, temperature, top_p	Generation settings.
base	Talk to what the run started from instead of the run.
runs_dir	Where feedback from $rate() and $edit() is recorded (under feedback/). See dragon_feedback().
context_window	Approximate token budget for the conversation (system prompt plus history), such as a small model's 2K to 8K context. When the next turn would go over it, $say() drops the oldest user/assistant pair and tries again until it fits, keeping the system prompt and the most recent turns. NULL (the default) never drops turns. The token count is a rough estimate (about 4 characters per token), not the model's own tokenizer.
path	A transcript written by $save().

Value

A dragon_chat object with methods: $say(text, on_token = NULL) sends a user turn and returns the reply (streaming pieces to on_token when the backend supports it); $history() returns the messages; $reset() clears them; $undo() drops the last exchange; $regenerate() asks again for the last reply; $rate("up") or $rate("down") records a verdict on the last reply and $edit(text) replaces it with a better one, both saved as feedback that dragon_feedback() turns into training data; $context_usage() reports the estimated tokens used, the window, and how many turns have been dropped to stay under it (NULL when context_window is not set); $save(path) and dragon_chat_load(path) write and read a transcript; $as_example() returns the conversation as one training row.

Examples

```
if (interactive()) {
chat <- dragon_chat(run, system = "You are a concise support agent.")
chat$say("My thermostat keeps dropping off Wi-Fi.")
chat$say("I tried that. What else?")   # the model sees the first exchange
chat$history()
chat$save("good-conversation.json")
}
```

---

dragon_check *Check the Python environment and hardware*

Description

Prepares the Python environment if needed, then reports the interpreter, library versions, the compute device that training will use, available GPU memory, and whether a Hugging Face token is configured. Run this first on a new machine.

Usage

```
dragon_check()
```

Value

Invisibly, a list of the collected facts.

Examples

```
if (interactive()) {
dragon_check()
}
```

---

dragon_code *R code that reproduces a run*

Description

Every run, including ones started from the Shiny app, can be replayed as a script. The dataset path is the original source when it was a file. A run that continued from an earlier run refers to that run by directory.

Usage

```
dragon_code(run)
```

Arguments

run	A dragon_run or run directory.

Value

A single string of R code.

---

dragon_compare *Compare runs side by side*

Description

One row per run with its stage, what it started from, and every number the package knows about it: held-out loss, perplexity, preference accuracy, task metrics from dragon_evaluate(), and the latest judge result from dragon_judge(). Runs that lack a measurement show NA.

Usage

```
dragon_compare(..., runs_dir = dragon_runs_dir())
```

Arguments

...	Runs, run directories, a single dragon_pipeline (its runs are compared), or nothing to list every run in runs_dir.
runs_dir	Directory scanned when no runs are given.

Value

A data frame of class dragon_comparison.

Examples

```
if (interactive()) {
dragon_compare()                # everything in the runs directory
dragon_compare(sft, dpo)        # two specific runs
}
```

`dragon_conversations` *Multi-turn conversations as training data*

Description

`dragon_map()` builds one user turn and one assistant turn per row. When the examples are whole conversations, chat transcripts for instance, use this instead. Each conversation is a list of messages with `role` and `content`; it must contain at least one user turn and end with an assistant turn, which is the turn the model learns to produce. Earlier turns are context.

Usage

```
dragon_conversations(x, name = NULL)
```

Arguments

`x`	A list of conversations (each a list of messages), a path to a JSONL file with one `{"messages": [...]}` object per line, or a data frame with a `messages` list column.
`name`	Display name. Defaults to the file name.

Value

A `dragon_dataset` mapped as conversations, usable wherever a prompt and response dataset is: `dragon_train()`, `dragon_bundle()`, `dragon_step_train()`.

Examples

```
convs <- list(
  list(
    list(role = "system", content = "You are a support agent."),
    list(role = "user", content = "My thermostat drops off Wi-Fi."),
    list(role = "assistant", content = "Which router do you use?"),
    list(role = "user", content = "An Eero."),
    list(role = "assistant",
        content = "Eero often band-steers 2.4 GHz devices. Make a 2.4 GHz-only network.")
  )
)
ds <- dragon_conversations(convs)
dragon_preview(ds)
```

dragon_dataset *Create a dataset for fine-tuning*

Description

Reads a file or wraps a data frame. Use `dragon_map()` afterwards to say which columns hold the prompt and the response.

Usage

```
dragon_dataset(x, ...)

## S3 method for class 'character'
dragon_dataset(x, format = NULL, name = NULL, ...)

## S3 method for class 'data.frame'
dragon_dataset(x, name = NULL, ...)
```

Arguments

`x`	A file path (CSV, TSV, JSONL, JSON array, or Parquet) or a data frame.
`...`	Passed to methods.
`format`	One of `"csv"`, `"tsv"`, `"jsonl"`, `"json"`, or `"parquet"`. Inferred from the file extension when `NULL`.
`name`	Display name. Defaults to the file name.

Value

A `dragon_dataset` object.

Examples

```
ds <- dragon_dataset(dragon_example_data())
ds
```

dragon_evaluate *Evaluate a finished run*

Description

Training already evaluates on the held-out rows and writes the result into the run directory. This function reads that result, or recomputes it with the saved adapter when `recompute = TRUE` or nothing was saved.

Usage

```
dragon_evaluate(run, n_samples = 10, recompute = FALSE, metrics = NULL)
```

Arguments

run	A dragon_run or run directory.
n_samples	Number of held-out prompts to generate replies for. Inf generates for every held-out row.
recompute	Reload the model and evaluate again.
metrics	Task metrics to compute on the generated replies: TRUE for every built-in metric, a character vector of names from dragon_metrics(), or a named list of functions. Metrics need a reply for every held-out row, so replies are generated for all of them when fewer are on disk. Results are saved to metrics.json in the run and show up in dragon_compare().

Value

A list with eval_loss, perplexity, eval_tokens, and a samples data frame with columns prompt, reference, generated. Preference runs report pref_accuracy (how often the model scores the chosen reply above the rejected one), reward_margin, and eval_pairs instead of perplexity, and their samples also carry rejected.

---

dragon_example_data *Path to the bundled example dataset*

Description

Two hundred synthetic customer-support tickets with subject, body, product, and reply columns. Small enough to train on a CPU in minutes.

Usage

```
dragon_example_data()
```

Value

A file path.

Examples

```
dragon_example_data()
```

dragon_export_gguf *Export a merged model to GGUF*

Description

Converts a merged model directory to GGUF for use with llama.cpp, Ollama, and similar runtimes. Requires a llama.cpp checkout; point the `LLAMA_CPP_DIR` environment variable at it.

Usage

```
dragon_export_gguf(
  merged_dir,
  out_file = NULL,
  quant = "q8_0",
  llama_cpp_dir = Sys.getenv("LLAMA_CPP_DIR", unset = "")
)
```

Arguments

`merged_dir`	A merged model directory from `dragon_merge()`.
`out_file`	Output path. Defaults to `model-<quant>.gguf` next to the model.
`quant`	Output type passed to the converter, such as `"q8_0"`, `"f16"`, or `"bf16"`.
`llama_cpp_dir`	Path to a llama.cpp checkout containing `convert_hf_to_gguf.py`.

Value

The output path, invisibly.

dragon_feedback *Training data from chat feedback*

Description

Ratings and edits made in `dragon_chat()` or the app's Chat panel are saved under `feedback/` in the runs directory. This turns them into data for the next stage:

Usage

```
dragon_feedback(runs_dir = dragon_runs_dir(), label = NULL)
```

Arguments

`runs_dir`	Runs directory holding `feedback/feedback.jsonl`.
`label`	Optional filter: only feedback given to this run id or backend model.

Details

- `sft`: a conversations dataset (`dragon_conversations()`) of replies the user liked or edited, each with the turns that led to it. Edited text replaces the model's reply.
- `pairs`: a preference dataset (`dragon_map_pairs()`) for prompts that received both a liked (or edited) reply and a disliked one.

Both are written as JSONL under `feedback/` so runs trained on them stay reproducible through `dragon_code()`.

Value

A list with `records` (a data frame of every feedback event), `sft` (dataset or `NULL`), and `pairs` (dataset or `NULL`).

Examples

```
if (interactive()) {
fb <- dragon_feedback()
fb$records
better <- dragon_train(fb$sft, dpo, wait = TRUE)     # continue from the run people chatted with
dpo2 <- dragon_prefer(fb$pairs, better, wait = TRUE)
}
```

---

dragon_generate *Generate replies from a fine-tuned model*

Description

Runs through the session's inference backend: by default a local Python worker that keeps the last models loaded, so only the first call pays the load. Pass a server backend to generate from Ollama or any OpenAI-compatible endpoint instead. See dragon_backend.

Usage

```
dragon_generate(
  x,
  prompt,
  system = NULL,
  max_new_tokens = 256,
  temperature = 0.7,
  top_p = 0.9,
  base = FALSE,
  backend = dragon_backend()
)
```

Arguments

x	A dragon_run, a run directory, an adapter directory, a merged model directory, or a Hugging Face model id.
prompt	One or more user prompts.
system	Optional system prompt.
max_new_tokens	Maximum tokens to generate per reply.
temperature	Sampling temperature. 0 means greedy decoding.
top_p	Nucleus sampling threshold.
base	Ignore this run's adapter and generate from what it started with: the base model, or the earlier run it continued from. Useful for before-and-after comparisons.
backend	Where to run inference. See dragon_backend. Server backends serve a fixed model and ignore x.

Value

A character vector, one reply per prompt.

---

dragon_hardware *Hardware settings*

Description

Hardware settings

Usage

```
dragon_hardware(
  device = c("auto", "cuda", "mps", "cpu"),
  dtype = c("auto", "bfloat16", "float16", "float32"),
  load_in_4bit = FALSE
)
```

Arguments

device	"auto" picks CUDA, then Apple MPS, then CPU. Or name one.
dtype	"auto" picks bfloat16 on GPUs that support it, float16 on older GPUs, and float32 elsewhere.
load_in_4bit	Load the base model in 4-bit through bitsandbytes. Only available on Linux with CUDA.

Value

A dragon_hardware object.

dragon_import *Import results trained on another machine*

Description

Copies the outputs of a run that trained elsewhere (through `dragon_remote()`) back into the local run directory: the adapter, the status, the progress log, the evaluation, and the sample generations. Afterwards `dragon_status()`, `dragon_progress()`, `dragon_evaluate()`, `dragon_generate()`, and `dragon_merge()` work as if the run had trained locally.

Usage

```
dragon_import(run, results)
```

Arguments

`run`	A `dragon_run` or run directory.
`results`	Path to the `dragonfarm-results-<run id>.zip` the notebook produced, or to a directory holding its unpacked contents.

Value

The run, invisibly.

Examples

```
if (interactive()) {
dragon_import(run, "~/Downloads/dragonfarm-results-20260914-101500-qwen2-5-0-5b-instruct.zip")
}
```

dragon_judge *Judge a run's replies with a language model*

Description

Two modes. With `against = NULL`, each reply is scored from 1 to 10 against a rubric, using the held-out reference answer as ground truth when there is one. With `against` set, the run's replies are compared pairwise with another model's replies to the same prompts, and the judge picks a winner. Pairwise judging asks each question twice with the two replies swapped, so a judge that favours whichever answer comes first cannot bias the result.

Usage

```
dragon_judge(
  x,
  against = NULL,
  prompts = NULL,
  n = 20,
  judge = NULL,
  rubric = NULL,
  system = NULL,
  max_new_tokens = 256,
  seed = 42
)
```

Arguments

x	A dragon_run or run directory whose replies are judged.
against	What to compare with: NULL for scoring alone, "base" for the model the run started from (its base model, or the run it continued), or another run, model id, or model directory.
prompts	Prompts to use. Defaults to n prompts from the run's held-out rows.
n	How many held-out prompts to use when prompts is NULL.
judge	The judge: a function taking a character vector of prompts and returning a character vector of replies, an ellmer chat object, or a Hugging Face model id or local model directory to use as a local judge. See dragon_judge_anthropic() for the Claude API.
rubric	What the judge should value. A sentence or two; a sensible default covers correctness, helpfulness, and following instructions.
system	System prompt used when generating the replies being judged.
max_new_tokens	Length cap for the generated replies.
seed	Seed for sampling the held-out prompts.

Details

Prompts default to the run's held-out set, so scores are comparable across runs that share a dataset. Results are written to judge.json in the run directory and the summary is recorded in the run's status, where dragon_compare() picks it up.

Value

A dragon_judgement object: a list with mode, summary, and details (one row per prompt).

Examples

```
if (interactive()) {
# Did preference optimization help? Compare the DPO run with the SFT run it started from.
j <- dragon_judge(dpo, against = "base", judge = dragon_judge_anthropic())
j$summary
```

```
# Absolute scores with a task-specific rubric.
dragon_judge(sft, rubric = "Reward replies that give concrete next steps and stay under 120 words.",
             judge = dragon_judge_anthropic(model = "claude-sonnet-5"))

# A local judge: any model dragon_generate() can load.
dragon_judge(sft, against = "base", judge = "Qwen/Qwen2.5-1.5B-Instruct")
}
```

`dragon_llm_anthropic` *Language models as functions: the Claude API and ellmer*

Description

Judges, teachers, and students in dragonfarm are plain functions from a character vector of prompts to a character vector of replies. These helpers build such functions.

Usage

```
dragon_llm_anthropic(
  model = "claude-opus-5",
  system = NULL,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  max_tokens = 1024,
  max_active = 4,
  temperature = NULL
)

dragon_judge_anthropic(
  model = "claude-opus-5",
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  max_tokens = 1024,
  max_active = 4
)

dragon_llm_ellmer(chat, system = NULL)

dragon_judge_ellmer(chat)
```

Arguments

`model`	Claude model id. The default is the most capable general model; `"claude-sonnet-5"` or `"claude-haiku-4-5"` are cheaper choices for large prompt sets.
`system`	Optional system prompt.
`api_key`	Anthropic API key.
`max_tokens`	Reply length cap.
`max_active`	How many requests to run at once.

temperature	Sampling temperature, or NULL for the API default.
chat	An ellmer chat object, for example ellmer::chat_anthropic().

Details

dragon_llm_anthropic() calls the Claude API directly over HTTP with refusal fallbacks enabled, reading the key from ANTHROPIC_API_KEY. dragon_llm_ellmer() wraps any ellmer chat, so every provider ellmer supports works; each prompt gets a fresh copy of the chat so no history leaks between questions. The dragon_judge_*() variants are the same with a system prompt that asks for JSON-only answers, which dragon_judge() and dragon_synthesize_pairs() need.

Value

A function suitable for the judge, teacher, or student arguments of dragon_judge(), dragon_synthesize(), and dragon_synthesize_pairs().

Examples

```
if (interactive()) {
teacher <- dragon_llm_anthropic(system = "You are a concise support agent.")
teacher(c("My thermostat drops off Wi-Fi.", "Invoice total looks wrong."))
}
```

---

dragon_lora *LoRA settings*

Description

LoRA settings

Usage

```
dragon_lora(r = 16, alpha = 32, dropout = 0.05, target_modules = "auto")
```

Arguments

r	Rank of the adapter matrices. Higher learns more, costs more memory.
alpha	Scaling factor. A common rule is alpha = 2 * r.
dropout	Dropout applied to the adapter input.
target_modules	"auto" adapts every linear layer except the output head (peft's "all-linear"). Or a character vector of module names such as c("q_proj", "v_proj").

Value

A dragon_lora object.

Examples

```
dragon_lora(r = 8, alpha = 16)
```

dragon_map *Map dataset columns to prompt, response, and system text*

Description

Each argument is either a column name or a `glue::glue()` template that combines several columns, such as `"{subject}\n\n{body}"`. Templates are rendered per row when the training files are written.

Usage

```
dragon_map(dataset, prompt, response, system = NULL)
```

Arguments

`dataset`	A `dragon_dataset`.
`prompt`	Column name or template for the user turn.
`response`	Column name or template for the assistant turn.
`system`	Optional column name or template for the system prompt. A template with no braces and no matching column is used as a constant system prompt for every row.

Value

The dataset with the mapping attached.

Examples

```
ds <- dragon_dataset(dragon_example_data())
ds <- dragon_map(ds, prompt = "{subject}\n\n{body}", response = "reply")
dragon_preview(ds, n = 1)
```

dragon_map_pairs *Map columns for preference optimization*

Description

For `dragon_prefer()`. Each row holds one prompt and two candidate replies: the one you prefer and the one you want the model to move away from. As in `dragon_map()`, each argument is a column name or a `glue::glue()` template combining several columns.

Usage

```
dragon_map_pairs(dataset, prompt, chosen, rejected, system = NULL)
```

Arguments

dataset	A dragon_dataset.
prompt	Column name or template for the user turn.
chosen	Column name or template for the preferred reply.
rejected	Column name or template for the reply to move away from.
system	Optional column name or template for the system prompt. A template with no braces and no matching column is used as a constant system prompt for every row.

Value

The dataset with the mapping attached.

Examples

```
df <- data.frame(
  q = c("What is 2 + 2?", "Capital of France?"),
  good = c("4", "Paris"),
  bad = c("5", "Lyon")
)
ds <- dragon_map_pairs(dragon_dataset(df), prompt = "q", chosen = "good", rejected = "bad")
dragon_preview(ds, n = 1)
```

---

dragon_map_prompts *Map columns for reinforcement learning*

Description

For dragon_reinforce(). Each row is a prompt the model will practise on, with an optional reference answer that reward functions such as "exact" and "numeric" compare against. Every other column travels along as fields, available to custom reward functions.

Usage

```
dragon_map_prompts(dataset, prompt, reference = NULL, system = NULL)
```

Arguments

dataset	A dragon_dataset.
prompt	Column name or template for the user turn.
reference	Optional column name or template for the reference answer.
system	Optional column name or template for the system prompt. A template with no braces and no matching column is used as a constant system prompt for every row.

Value

The dataset with the mapping attached.

Examples

```
df <- data.frame(question = c("12 * 12?", "Capital of Peru?"), answer = c("144", "Lima"))
ds <- dragon_map_prompts(dragon_dataset(df), prompt = "question", reference = "answer")
dragon_preview(ds, n = 1)
```

---

dragon_merge *Merge the adapter into the base model*

Description

Produces a standalone model directory that loads with plain transformers and needs neither peft nor dragonfarm.

Usage

```
dragon_merge(run, out_dir = NULL)
```

Arguments

run	A dragon_run or run directory.
out_dir	Where to write the merged model. Defaults to merged/ inside the run directory.

Value

The output path, invisibly.

---

dragon_metrics *Task metrics for generated replies*

Description

Deterministic checks that need no judge model. Each metric is a function of three character vectors, generated, reference, and prompt, and returns one number per row (0 or 1 for pass/fail metrics). Pass names from this list, or your own functions, to dragon_evaluate().

Usage

```
dragon_metrics()

dragon_metric_regex(pattern, ignore_case = TRUE)
```

Arguments

pattern	A regular expression the reply must match.
ignore_case	Case-insensitive match.

Details

- exact: normalized exact match (case, whitespace, and trailing punctuation ignored).
- contains: the normalized reference appears inside the reply.
- token_f1: token overlap F1 between reply and reference, the SQuAD style partial-credit score.
- json_valid: the reply parses as JSON (a fenced code block is unwrapped first).
- numeric: the last number in the reply equals the last number in the reference.
- length_ratio: characters in the reply divided by characters in the reference. Useful for spotting rambling or truncation.

dragon_metric_regex() builds a metric that passes when the reply matches a pattern, for format checks such as "starts with a ticket id".

Value

dragon_metrics(): a named list of metric functions.

dragon_metric_regex(): a metric function.

Examples

```
m <- dragon_metrics()
m$exact("Paris.", "paris", "Capital of France?")
m$token_f1("the cat sat on the mat", "a cat sat on a mat", "")
m$json_valid('```json\n{"a": 1}\n```', "", "")
```

---

dragon_pipeline *Run several post-training stages as one pipeline*

Description

Executes the steps in order, threading the result of each into the next: every training stage starts from the previous stage's run, synthesized pairs feed the next preference stage, and judge and evaluate steps score the latest run. Progress is written to pipelines/<id>.json under runs_dir after every step, so a pipeline can be watched from the app or another session with dragon_pipeline_status().

Usage

```
dragon_pipeline(
  model,
  steps,
  runs_dir = dragon_runs_dir(),
  name = "pipeline",
  background = FALSE
)
```

Arguments

model	Where the first training stage starts: a model id, a model directory, or a finished dragon_run.
steps	A list of dragon_step objects.
runs_dir	Where runs and the pipeline record are written.
name	Label used in the pipeline id.
background	Run in a separate R process.

Details

With background = TRUE the pipeline runs in a separate R process and the call returns at once. Steps must then be serializable: judges and teachers made with dragon_llm_anthropic() or plain functions are fine, ellmer chat objects are not.

Value

A dragon_pipeline object. In the foreground it holds the runs and every step's summary; in the background it is a handle whose progress dragon_pipeline_status() reads.

Examples

```
if (interactive()) {
p <- dragon_pipeline("Qwen/Qwen2.5-0.5B-Instruct", list(
  dragon_step_train(tickets),
 dragon_step_synthesize_pairs(judge = dragon_judge_anthropic(model = "claude-sonnet-5")),
  dragon_step_prefer(),
  dragon_step_judge(judge = dragon_judge_anthropic())
), background = TRUE)
dragon_pipeline_status(p)
}
```

`dragon_pipeline_cancel` *Cancel a pipeline*

Description

Writes a cancel request next to the pipeline record. The runner checks it between steps and stops there. A training step that is under way is asked to stop as well, the way `dragon_cancel()` does, so it saves a checkpoint first; a judging, synthesis, or merge step finishes before the pipeline stops. With `wait = TRUE` the call returns once the pipeline has stopped, killing the pipeline process if it is still going after `timeout` seconds.

Usage

```
dragon_pipeline_cancel(
  x,
  runs_dir = dragon_runs_dir(),
  wait = TRUE,
  timeout = 120
)
```

Arguments

`x`	A `dragon_pipeline` handle or a pipeline id.
`runs_dir`	Where the pipeline record lives, when `x` is an id.
`wait`	Wait for the pipeline to stop.
`timeout`	Seconds to wait before killing the pipeline process.

Value

The pipeline status, invisibly.

`dragon_pipeline_status` *Progress of a pipeline*

Description

Progress of a pipeline

Usage

```
dragon_pipeline_status(x, runs_dir = dragon_runs_dir())
```

Arguments

x	A dragon_pipeline object or a pipeline id.
runs_dir	Where the pipeline record lives, when x is an id.

Value

The pipeline record as a list: status, steps (each with a state and, when finished, a summary), and timestamps.

---

dragon_prefer *Preference optimization with DPO or ORPO*

Description

The second stage of post-training. Where dragon_train() teaches a model what a good reply looks like, this teaches it which of two replies is better, from a dataset mapped with dragon_map_pairs().

Usage

```
dragon_prefer(
  dataset,
  model,
  method = c("dpo", "orpo"),
  beta = 0.1,
  lora = dragon_lora(),
  args = dragon_train_args(learning_rate = 5e-05, epochs = 2),
  hardware = dragon_hardware(),
  name = NULL,
  run_dir = NULL,
  runs_dir = dragon_runs_dir(),
  n_samples = 10,
  revision = NULL,
  trust_remote_code = FALSE,
  wait = FALSE
)
```

Arguments

dataset	A dataset mapped with dragon_map_pairs().
model	A Hugging Face model id such as "HuggingFaceTB/SmolLM2-135M-Instruct", a local model directory, or a finished dragon_run to continue from. In the last case the earlier run's adapters are folded into the weights before this run adds its own, so stages chain: fine-tune, then dragon_prefer(), and so on. See dragon_presets() for model ids.
method	"dpo" or "orpo".
beta	Preference strength. Typical values are 0.05 to 0.5.

lora	LoRA settings from dragon_lora().
args	Training settings. The defaults use a lower learning rate than dragon_train(), which preference methods need.
hardware	Hardware settings from dragon_hardware().
name	Short label used in the run id. Defaults to the model name.
run_dir	Exact directory to use. Defaults to a timestamped directory under runs_dir.
runs_dir	Parent directory for runs. See dragon_runs_dir().
n_samples	Number of held-out rows to generate sample replies for at the end of training.
revision	Model revision (branch, tag, or commit) on the Hub.
trust_remote_code	Allow the model repository to run custom code.
wait	Block until training finishes.

Details

Two methods are available:

- "dpo", Direct Preference Optimization. Pushes the model's implicit reward for the chosen reply above the rejected one, relative to a reference model. With LoRA the reference is the same model with the adapter switched off, so it costs no extra memory.
- "orpo", Odds Ratio Preference Optimization. Combines the supervised loss on the chosen reply with an odds-ratio penalty on the rejected one. Needs no reference model and works from a base model that has not been fine-tuned yet.

beta controls how hard the model is pushed: the KL strength for DPO, the odds-ratio weight for ORPO. 0.1 is a sensible start for both.

Pass a finished run as model to continue from it. Its adapters are folded into the weights before this stage adds its own, which is the usual sequence: dragon_train() first, then dragon_prefer() on top.

Value

A dragon_run object.

Examples

```
if (interactive()) {
sft <- dragon_dataset("tickets.csv") |>
  dragon_map(prompt = "question", response = "answer") |>
  dragon_train("Qwen/Qwen2.5-0.5B-Instruct", wait = TRUE)

dpo <- dragon_dataset("preferences.csv") |>
  dragon_map_pairs(prompt = "question", chosen = "better", rejected = "worse") |>
  dragon_prefer(sft, method = "dpo", beta = 0.1, wait = TRUE)

dragon_evaluate(dpo)
dragon_generate(dpo, "My thermostat keeps dropping off Wi-Fi.")
}
```

dragon_presets *Recommended small models*

Description

A table of small instruction-tuned models that work well with LoRA on a single consumer GPU or, for the smallest, a CPU. Any Hugging Face causal language model id can be passed to `dragon_train()`; these are just good starting points.

Usage

```
dragon_presets()
```

Value

A data frame with columns `id`, `params`, `license`, `gated`, `rank`, `min_vram_gb`, and `notes`.

Examples

```
dragon_presets()
```

dragon_preview *Preview mapped rows as chat turns*

Description

Preview mapped rows as chat turns

Usage

```
dragon_preview(dataset, n = 3)
```

Arguments

`dataset`	A mapped `dragon_dataset`.
`n`	Number of rows to show.

Value

Invisibly, a list of message lists.

dragon_prompts *Prompts from a run's data files*

Description

The user turns of a run's training or held-out rows, for feeding `dragon_synthesize_pairs()` or `dragon_judge()`. Works for prompt/response and preference-pair runs alike.

Usage

```
dragon_prompts(run, split = c("eval", "train"), n = Inf, seed = 42)
```

Arguments

`run`	A `dragon_run` or run directory.
`split`	`"eval"` for the held-out rows, `"train"` for the rest.
`n`	Maximum number of prompts. `Inf` for all.
`seed`	Seed used when sampling down to `n`.

Value

A character vector.

dragon_publish *Push a run's model to the Hugging Face Hub*

Description

Uploads a run's model as a repo on the Hugging Face Hub, ready for any hosted inference: a dedicated Inference Endpoint behind `dragon_backend_server()`, `transformers-cli`, or any tool that loads a model by repo id. The repo is created if it does not exist.

Usage

```
dragon_publish(
  run,
  repo,
  what = c("merged", "adapter"),
  private = FALSE,
  commit_message = NULL
)
```

Arguments

run	A dragon_run or run directory.
repo	A repo id, "username/name".
what	What to push. "merged" folds the adapter into the base model first (merging it if that has not been done yet), so the repo loads with plain transformers and needs neither peft nor dragonfarm; this is the usual choice for hosted inference. "adapter" pushes just the LoRA weights: a small, fast upload, but the base model and peft are needed to load it.
private	Create the repo as private.
commit_message	Defaults to a message naming the run and the stage.

Value

The repo URL, invisibly.

Examples

```
if (interactive()) {
dragon_publish(run, "yourname/support-agent-0.5b")
dragon_publish(run, "yourname/support-agent-0.5b-adapter", what = "adapter")
}
```

---

dragon_python_requirements

Python requirements used by dragonfarm

Description

The package declares these with reticulate::py_require() when it is loaded. You normally never call this yourself.

Usage

```
dragon_python_requirements()
```

Value

A list with packages (pip requirement strings) and python_version.

Examples

```
dragon_python_requirements()
```

dragon_reinforce *Reinforcement learning with verifiable rewards (GRPO)*

Description

The third stage of post-training. For every prompt the model writes several completions, each is scored by the rewards, and the model is nudged towards the completions that beat their group's average. A KL penalty against the model it started from keeps it from drifting. This is Group Relative Policy Optimization, the method behind recent reasoning models, in its plain on-policy form.

Usage

```
dragon_reinforce(
  dataset,
  model,
  rewards,
  group_size = 4,
  beta = 0.04,
  temperature = 1,
  max_new_tokens = 128,
  lora = dragon_lora(),
 args = dragon_train_args(learning_rate = 1e-05, epochs = 1, batch_size = 4, grad_accum
    = 1, save_steps = 20),
  hardware = dragon_hardware(),
  name = NULL,
  run_dir = NULL,
  runs_dir = dragon_runs_dir(),
  n_samples = 10,
  revision = NULL,
  trust_remote_code = FALSE,
  wait = FALSE
)
```

Arguments

dataset	A dataset mapped with dragon_map_prompts().
model	A Hugging Face model id such as "HuggingFaceTB/SmolLM2-135M-Instruct", a local model directory, or a finished dragon_run to continue from. In the last case the earlier run's adapters are folded into the weights before this run adds its own, so stages chain: fine-tune, then dragon_prefer(), and so on. See dragon_presets() for model ids.
rewards	A dragon_reward() or a list of them.
group_size	Completions sampled per prompt. 4 to 8 is typical; more gives a better baseline at more cost per step.

beta	Weight of the KL penalty towards the starting model. Higher is more conservative.
temperature	Sampling temperature for the completions. Must be above zero so the group varies.
max_new_tokens	Length cap for each sampled completion.
lora	LoRA settings from dragon_lora().
args	Training settings. batch_size is prompts per step (so batch_size * group_size completions), epochs or max_steps set the length, and save_steps the checkpoint interval. The defaults use a small learning rate, which RL needs.
hardware	Hardware settings from dragon_hardware().
name	Short label used in the run id. Defaults to the model name.
run_dir	Exact directory to use. Defaults to a timestamped directory under runs_dir.
runs_dir	Parent directory for runs. See dragon_runs_dir().
n_samples	Number of held-out rows to generate sample replies for at the end of training.
revision	Model revision (branch, tag, or commit) on the Hub.
trust_remote_code	Allow the model repository to run custom code.
wait	Block until training finishes.

Details

It works well when the reward is something you can check: a correct number, valid JSON with the right keys, a required format, a length budget, a test that passes. It works poorly as a substitute for preference data on vague goals such as "be more helpful"; use dragon_prefer() for those.

Pass a finished run as model to continue from it, which is the usual order: dragon_train(), optionally dragon_prefer(), then this.

Value

A dragon_run object.

Examples

```
if (interactive()) {
math <- dragon_dataset("arithmetic.csv") |>
  dragon_map_prompts(prompt = "question", reference = "answer")
rl <- dragon_reinforce(
  math, sft,
 rewards = list(dragon_reward("numeric"), dragon_reward("length", max_chars = 300, weight = 0.2)),
  group_size = 6, wait = TRUE
)
dragon_evaluate(rl)     # mean reward on held-out prompts, per reward
}
```

dragon_remote *Open a cloud GPU provider for a bundled run*

Description

Prints the steps for running a bundled run on the chosen provider and, by default in an interactive session, opens the provider in the browser with the dragon-farm notebook loaded. Runs that were not bundled yet are bundled first, so a run that failed locally for lack of memory can be sent to the cloud as is.

Usage

```
dragon_remote(
  run,
  provider = c("colab", "kaggle", "lightning", "runpod"),
  open = interactive()
)
```

Arguments

run	A dragon_run or run directory.
provider	One of "colab", "kaggle", "lightning", "runpod". See dragon_remote_providers().
open	Open the provider link in the browser.

Value

Invisibly, a list with provider, url, bundle (path to the zip), and steps (a character vector).

Examples

```
if (interactive()) {
dragon_remote(run, "kaggle")
}
```

dragon_remote_providers *Cloud GPU providers*

Description

Machines without a GPU can still fine-tune: dragon_bundle() packages a run as a zip, dragon_remote() opens one of these providers with the dragon-farm notebook, and dragon_import() brings the trained adapter back. Google Colab and Kaggle have free GPU tiers. Lightning AI gives free monthly credits. RunPod is pay per hour.

Usage

```
dragon_remote_providers()
```

Value

A data frame with one row per provider: `provider` (the id to pass to `dragon_remote()`), `name`, `cost`, and `opens` (what the link opens).

Examples

```
dragon_remote_providers()
```

---

dragon_resume *Resume a run from its latest checkpoint*

Description

Useful after a cancel or a crash. Continues with the same configuration.

Usage

```
dragon_resume(run, wait = FALSE)
```

Arguments

`run`	A `dragon_run`.
`wait`	Block until training finishes.

Value

A `dragon_run` object.

---

dragon_reward *Verifiable rewards for reinforcement learning*

Description

A reward scores one completion between 0 and 1. `dragon_reinforce()` takes one or more, sums them by `weight`, and pushes the model towards higher totals. These built-ins are verifiable: they check facts about the text rather than asking a model's opinion, which is what makes reinforcement learning work on small models.

Usage

```
dragon_reward(
  type = c("exact", "contains", "numeric", "regex", "json", "length", "keyword",
    "command", "custom"),
  weight = 1,
  name = NULL,
  pattern = NULL,
  case_sensitive = FALSE,
  keys = NULL,
  min_chars = NULL,
  max_chars = NULL,
  words = NULL,
  mode = c("any", "all"),
  command = NULL,
  input = c("stdin", "file"),
  score_from = c("exit_code", "stdout"),
  min_score = 0,
  max_score = 1,
  timeout = 30,
  file = NULL,
  fn = "reward"
)
```

Arguments

type	Which reward.
weight	Multiplier when rewards are summed.
name	Label used in progress rows and evaluation. Defaults to the type.
pattern	Regular expression, for "regex".
case_sensitive	Whether pattern is case-sensitive.
keys	Required top-level keys, for "json".
min_chars, max_chars	Bounds, for "length". Give at least one.
words	Words to look for, for "keyword".
mode	"any" or "all" of words.
command	A character vector: the command and its arguments (run directly, not through a shell), for "command".
input	"stdin" or "file", for "command".
score_from	"exit_code" (0 means 1.0, anything else 0.0) or "stdout" (the last number the command prints, rescaled from min_score/max_score and clamped to 0 to 1), for "command".
min_score, max_score	Range that a "stdout" score is rescaled from, for "command".
timeout	Seconds before a "command" reward gives up and scores 0.
file	Python file, for "custom".
fn	Name of the function inside file.

Details

- "exact": normalized exact match with the row's reference.
- "contains": the reference appears in the completion.
- "numeric": the last number in the completion equals the reference's.
- "regex": the completion matches pattern.
- "json": the completion is valid JSON, with keys present if given (partial credit per key).
- "length": 1 within min_chars to max_chars, falling to 0 beyond.
- "keyword": any (or all) of words appear.
- "command": runs command and scores the completion by its exit code or its stdout. Common for code tasks: command a test suite or a linter. Never runs through a shell, so the completion's own text cannot inject anything into the command line: with input = "stdin" (the default) the completion is piped to the command's stdin; with input = "file" it is written to a temp file whose path replaces every "{completion_file}" token in command. This reward runs on whichever machine trains the run, so a cloud notebook needs command to be available there too.
- "custom": a Python file defining reward(prompt, completion, reference, row) that returns a number. The file is copied into the run so the run stays self-contained.

Value

A dragon_reward object.

Examples

```
dragon_reward("exact")
dragon_reward("regex", pattern = "^T-\\d{4}", weight = 2)
dragon_reward("length", max_chars = 400, weight = 0.5)
dragon_reward("json", keys = c("id", "status"))
if (interactive()) {
dragon_reward("command", command = c("pytest", "-q", "--tb=no"), input = "file")
}
```

---

dragon_run *Reopen an existing run*

Description

Runs live entirely on disk, so any run can be picked up from a new R session by its directory.

Usage

```
dragon_run(dir)
```

Arguments

dir	Path to a run directory (one containing config.json).

Value

A dragon_run object.

---

dragon_runs *List runs*

Description

List runs

Usage

```
dragon_runs(runs_dir = dragon_runs_dir())
```

Arguments

runs_dir	Directory holding run directories.

Value

A data frame with one row per run, newest first.

---

dragon_runs_dir *Directory where runs are stored*

Description

Every function that writes a run (dragon_train(), dragon_bundle(), the app, and so on) takes a runs_dir argument that defaults to this. Without configuration it resolves to a dragonfarm_runs folder under a session temp directory, so a fresh R session never writes to your working directory or home filespace by default; that folder disappears once the session ends. For runs you want to keep, set a real location once with options(dragonfarm.runs_dir = "path/to/dragonfarm_runs") or the DRAGONFARM_RUNS_DIR environment variable, or pass runs_dir= to the function you're calling.

Usage

```
dragon_runs_dir()
```

Value

A path.

Examples

```
dragon_runs_dir()
withr::with_options(list(dragonfarm.runs_dir = "~/dragonfarm_runs"), dragon_runs_dir())
```

`dragon_serve_ollama` *Serve a run's model with Ollama*

Description

Merges the adapter into the base model (if not done already) and registers the result with Ollama, which imports safetensors directly for the Llama, Qwen2, Gemma, and related families. Ollama then serves it quickly on the CPU or GPU of whatever machine it runs on, and the returned backend points `dragon_generate()` and `dragon_chat()` at it.

Usage

```
dragon_serve_ollama(
  run,
  name = NULL,
  quantize = NULL,
  ollama = Sys.which("ollama")
)
```

Arguments

`run`	A `dragon_run`, or a merged model directory.
`name`	Model name to register. Defaults to `dragonfarm-<run id>`.
`quantize`	Optional Ollama quantization such as `"q8_0"` or `"q4_K_M"` to shrink the served model.
`ollama`	Path to the `ollama` executable.

Details

Needs the `ollama` command on the PATH and the Ollama service running.

Value

A `dragon_backend` for the served model.

Examples

```
if (interactive()) {
backend <- dragon_serve_ollama(run)
options(dragonfarm.backend = backend)
dragon_chat(run)$say("Hello")
}
```

dragon_split *Hold out rows for evaluation*

Description

If you do not call this, `dragon_train()` holds out 5 percent of rows (and none when the dataset has fewer than 20 rows).

Usage

```
dragon_split(dataset, eval_frac = 0.05, seed = 42)
```

Arguments

`dataset`	A `dragon_dataset`.
`eval_frac`	Fraction of rows to hold out.
`seed`	Random seed for the split.

Value

The dataset with the split attached.

dragon_status *Inspect a run*

Description

`dragon_status()` reads the run's state. `dragon_progress()` returns one row per logged step. `dragon_logs()` returns the tail of the trainer log.

Usage

```
dragon_status(run)

dragon_progress(run)

dragon_logs(run, n = 50)
```

Arguments

`run`	A `dragon_run`.
`n`	Number of log lines to return.

Value

`dragon_status()`: a list with at least `state`, one of `"queued"`, `"running"`, `"succeeded"`, `"failed"`, `"cancelled"`.

`dragon_progress()`: a data frame with columns `step`, `epoch`, `loss`, `eval_loss`, `lr`, `grad_norm`, `elapsed_s`, `eta_s`, and for preference runs `pref_acc`, `reward_margin`, `eval_pref_acc`, `eval_reward_margin`.

`dragon_logs()`: a character vector.

`dragon_step` *Steps of a post-training pipeline*

Description

Each step becomes one stage of `dragon_pipeline()`. Stages chain: a training step's run is the starting point of the next training step, a synthesis step's pairs feed the next preference step, judge and evaluate steps measure the most recent run, and a merge step writes it out as a standalone model.

Usage

```
dragon_step_train(dataset, lora = dragon_lora(), args = NULL, n_samples = 10)

dragon_step_prefer(
  dataset = NULL,
  method = c("dpo", "orpo"),
  beta = 0.1,
  lora = dragon_lora(),
  args = NULL,
  n_samples = 10
)

dragon_step_synthesize_pairs(
  prompts = "train",
  n = 100,
  judge = NULL,
  n_samples_per_prompt = 4,
  min_gap = 2,
  rubric = NULL,
  temperature = 0.8,
  max_new_tokens = 256
)

dragon_step_reinforce(
  dataset,
  rewards,
  group_size = 4,
  beta = 0.04,
  temperature = 1,
```

```
  max_new_tokens = 128,
  lora = dragon_lora(),
  args = NULL,
  n_samples = 10
)

dragon_step_judge(against = "base", judge = NULL, n = 20, rubric = NULL)

dragon_step_evaluate(metrics = TRUE)

dragon_step_merge(out = NULL)

dragon_step_publish(
  repo,
  what = c("merged", "adapter"),
  private = FALSE,
  commit_message = NULL
)
```

Arguments

dataset	A mapped dataset for the stage. dragon_step_prefer() may leave it NULL to use the pairs made by the preceding dragon_step_synthesize_pairs().
lora, args, n_samples	As in dragon_train(). args = NULL uses the stage's own defaults.
method, beta	As in dragon_prefer().
prompts	"train" or "eval" to take prompts from the current run's data, or a character vector or mapped dataset.
n	How many prompts to use.
judge, rubric	As in dragon_judge().
n_samples_per_prompt, min_gap, temperature, max_new_tokens	As in dragon_synthesize_pairs().
rewards, group_size	As in dragon_reinforce().
against	As in dragon_judge().
metrics	As in dragon_evaluate().
out	As out_dir in dragon_merge(): where the merged model goes; NULL means merged/ inside the run.
repo, what, private, commit_message	As in dragon_publish().

Value

A dragon_step object.

Examples

```
if (interactive()) {
steps <- list(
  dragon_step_train(tickets),
 dragon_step_synthesize_pairs(prompts = "train", n = 150, judge = dragon_judge_anthropic()),
  dragon_step_prefer(),
  dragon_step_judge(against = "base", judge = dragon_judge_anthropic()),
  dragon_step_evaluate(metrics = c("token_f1", "length_ratio"))
)
p <- dragon_pipeline("Qwen/Qwen2.5-0.5B-Instruct", steps)
dragon_compare(p)
}
```

`dragon_synthesize` *Write fine-tuning data with a teacher model*

Description

Sends each prompt to a stronger model and keeps its replies as the responses to train on. This is the fastest way to get good training data for a small model: a few hundred prompts from your domain, answered the way you want them answered. A quality pass then drops rows that are too short or too long, look garbled, repeat an earlier prompt, or repeat (or nearly repeat) an earlier response, so a teacher's stock phrases don't dominate the dataset. Passing a `judge` adds distillation with a quality gate: it scores every surviving reply and keeps only the ones at or above `min_score`, the way you would with a teacher answering a student's own prompts (see `dragon_prompts()`) and filtering out its weaker answers. The result is saved as JSONL and returned as a mapped dataset ready for `dragon_train()`.

Usage

```
dragon_synthesize(
  prompts,
  teacher,
  system = NULL,
  max_new_tokens = 512,
  temperature = 0.7,
  judge = NULL,
  min_score = 7,
  rubric = NULL,
  dedupe = TRUE,
  near_dup_threshold = 0.92,
  min_chars = 1,
  max_chars = Inf,
  min_alpha_ratio = 0,
  file = NULL,
  runs_dir = dragon_runs_dir(),
  name = "synthetic"
)
```

Arguments

prompts	A character vector of prompts, or a mapped dragon_dataset whose prompt template is rendered for every row. See dragon_prompts() for prompts from an existing run, which is how distillation from "the model's own prompts" is done: pass dragon_prompts(run, "train").
teacher	The model that writes the replies: dragon_llm_anthropic(), an ellmer chat, a model id or directory, a finished run, or any function from prompts to replies.
system	System prompt for the teacher. Also stored with every row so the student trains with the same instruction.
max_new_tokens	Reply length cap for local teachers.
temperature	Sampling temperature for local teachers.
judge	Optional. Scores every surviving reply from 1 to 10 and drops the ones below min_score. See dragon_judge() for what is accepted.
min_score	Minimum judge score to keep a reply. Only used when judge is given.
rubric	What the judge should value. See dragon_judge().
dedupe	Drop rows whose prompt repeats an earlier one, and rows whose response exactly or nearly repeats an earlier response (see near_dup_threshold).
near_dup_threshold	Word-overlap similarity (0 to 1) above which two responses count as near-duplicates. Lower catches more; 1 disables near-duplicate detection while leaving exact-duplicate detection on.
min_chars, max_chars	Keep only replies whose length in characters falls in this range.
min_alpha_ratio	Minimum share of printable ASCII characters in a reply; a crude filter for garbled output or a reply in the wrong script. 0 (the default) disables it; non-English replies need it left off or set low.
file	Where to write the JSONL. Defaults to a timestamped file under synth/ in runs_dir.
runs_dir	Parent directory for the default file.
name	Label used in the default file name.

Value

A dragon_dataset mapped with prompt and response columns (and system, when given). The file path is its source, so dragon_code() reproduces runs trained on it. attr(ds, "synthesis")$dropped breaks down what the quality pass (and the judge, if used) removed.

Examples

```
if (interactive()) {
tickets <- dragon_dataset("tickets.csv") |>
  dragon_map(prompt = "{subject}\n\n{body}", response = "reply")
persona <- "You are a concise, warm support agent for a smart-home company."
teacher <- dragon_llm_anthropic(system = persona)
```

```
synth <- dragon_synthesize(tickets, teacher, system = persona)
run <- dragon_train(synth, "Qwen/Qwen2.5-0.5B-Instruct", wait = TRUE)

# Distill from the student's own prompts, keeping only replies a judge likes.
distilled <- dragon_synthesize(dragon_prompts(run, "train"), teacher,
                               judge = dragon_judge_anthropic(), min_score = 7)
}
```

`dragon_synthesize_pairs` *Build preference pairs from a model's own samples*

Description

Preference optimization needs, for each prompt, a better and a worse reply. This function makes them without hand labelling, in one of two ways:

Usage

```
dragon_synthesize_pairs(
  prompts,
  student,
  judge = NULL,
  teacher = NULL,
  n_samples = 4,
  min_gap = 2,
  rubric = NULL,
  system = NULL,
  temperature = 0.8,
  max_new_tokens = 256,
  file = NULL,
  runs_dir = dragon_runs_dir(),
  name = "preferences"
)
```

Arguments

`prompts`	A character vector of prompts, or a mapped `dragon_dataset` whose prompt template is rendered for every row. See `dragon_prompts()` for prompts from an existing run, which is how distillation from "the model's own prompts" is done: pass `dragon_prompts(run, "train")`.
`student`	The model whose replies are being improved: normally the `dragon_run` you will continue from. Also a model id, ellmer chat, or function.
`judge`	Scores the student's samples. See `dragon_judge()` for what is accepted. Required unless `teacher` is given.
`teacher`	Writes the chosen reply instead of judging. See `dragon_synthesize()`.

n_samples	Samples per prompt in judge mode. At least 2.
min_gap	Minimum score difference between chosen and rejected in judge mode. Pairs below it are dropped.
rubric	What the judge should value. See dragon_judge().
system	System prompt for the teacher. Also stored with every row so the student trains with the same instruction.
temperature	Sampling temperature for the student. Needs to be above zero in judge mode, or every sample is the same.
max_new_tokens	Reply length cap for local teachers.
file	Where to write the JSONL. Defaults to a timestamped file under synth/ in runs_dir.
runs_dir	Parent directory for the default file.
name	Label used in the default file name.

Details

- With a judge: the student answers each prompt n_samples times at a non-zero temperature, the judge scores every sample from 1 to 10, and the best and worst become the chosen and rejected replies. Prompts whose samples are too close (min_gap) or identical are dropped. This is the loop behind RLAIF-style training: the model improves on its own outputs under a judge's preferences.
- With a teacher: the teacher's reply is chosen and the student's is rejected. Cheap and effective when the teacher is clearly stronger.

The result is saved as JSONL and returned as a dataset mapped with dragon_map_pairs(), ready for dragon_prefer(), usually continuing from the student run itself.

Value

A dragon_dataset mapped as preference pairs. In judge mode the file also records chosen_score and rejected_score.

Examples

```
if (interactive()) {
# Close the loop: sample from the fine-tuned run, let a judge rank, train DPO on the result.
pairs <- dragon_synthesize_pairs(dragon_prompts(sft, "train", n = 200), student = sft,
                                 judge = dragon_judge_anthropic(model = "claude-sonnet-5"))
dpo <- dragon_prefer(pairs, sft, wait = TRUE)
dragon_judge(dpo, against = "base", judge = dragon_judge_anthropic())
}
```

dragon_train *Fine-tune a model with LoRA*

Description

Writes a run directory, then launches the trainer as a background Python process. Returns immediately unless `wait = TRUE`. The run survives the R session; reopen it later with `dragon_run()`.

Usage

```
dragon_train(
  dataset,
  model,
  lora = dragon_lora(),
  args = dragon_train_args(),
  hardware = dragon_hardware(),
  name = NULL,
  run_dir = NULL,
  runs_dir = dragon_runs_dir(),
  n_samples = 10,
  revision = NULL,
  trust_remote_code = FALSE,
  wait = FALSE
)
```

Arguments

`dataset`	A mapped `dragon_dataset` (see `dragon_map()`).
`model`	A Hugging Face model id such as `"HuggingFaceTB/SmolLM2-135M-Instruct"`, a local model directory, or a finished `dragon_run` to continue from. In the last case the earlier run's adapters are folded into the weights before this run adds its own, so stages chain: fine-tune, then `dragon_prefer()`, and so on. See `dragon_presets()` for model ids.
`lora`	LoRA settings from `dragon_lora()`.
`args`	Training settings from `dragon_train_args()`.
`hardware`	Hardware settings from `dragon_hardware()`.
`name`	Short label used in the run id. Defaults to the model name.
`run_dir`	Exact directory to use. Defaults to a timestamped directory under `runs_dir`.
`runs_dir`	Parent directory for runs. See `dragon_runs_dir()`.
`n_samples`	Number of held-out rows to generate sample replies for at the end of training.
`revision`	Model revision (branch, tag, or commit) on the Hub.
`trust_remote_code`	Allow the model repository to run custom code.
`wait`	Block until training finishes.

Value

A dragon_run object.

Examples

```
if (interactive()) {
run <- dragon_dataset(dragon_example_data()) |>
  dragon_map(prompt = "{subject}\n\n{body}", response = "reply") |>
  dragon_train("HuggingFaceTB/SmolLM2-135M-Instruct", wait = TRUE)
dragon_generate(run, "My order arrived damaged.")
}
```

---

dragon_train_args *Training settings*

Description

Training settings

Usage

```
dragon_train_args(
  epochs = 3,
  learning_rate = 2e-04,
  batch_size = 4,
  grad_accum = 4,
  max_seq_len = 1024,
  max_steps = NULL,
  warmup_ratio = 0.03,
  weight_decay = 0,
  logging_steps = 5,
  save_steps = 100,
  gradient_checkpointing = FALSE,
  seed = 42,
  ...
)
```

Arguments

epochs	Passes over the training data. Ignored when max_steps is set.
learning_rate	Peak learning rate. 2e-4 is a good LoRA default.
batch_size	Examples per device per step.
grad_accum	Steps to accumulate before an optimizer update. Effective batch size is batch_size * grad_accum.
max_seq_len	Maximum tokens per example. Longer examples are truncated.
max_steps	Stop after this many optimizer steps. NULL trains for epochs.

warmup_ratio	Fraction of steps used to warm up the learning rate.
weight_decay	Weight decay for the optimizer.
logging_steps	Steps between progress rows.
save_steps	Steps between checkpoints.
gradient_checkpointing	Trade compute for memory. Turn on if you run out of GPU memory.
seed	Random seed.
...	Extra named arguments passed straight to TrainingArguments.

Value

A dragon_train_args object.

Examples

```
dragon_train_args(epochs = 1, learning_rate = 1e-4)
```

---

dragon_wait *Wait for a run to finish*

Description

Blocks with a progress bar until the run reaches a terminal state.

Usage

```
dragon_wait(run, timeout = Inf, poll = 2)
```

Arguments

run	A dragon_run.
timeout	Seconds to wait before giving up (the run keeps going).
poll	Seconds between checks.

Value

The run, invisibly. Errors if the run failed.

dragon_worker_stop *Stop the local inference worker*

Description

The local backend keeps a Python process alive with the last models loaded. Call this to free the memory (GPU included). It restarts on the next generation call.

Usage

```
dragon_worker_stop()
```

Value

TRUE invisibly.

Index

dragon_app, 3
dragon_archive_run, 4
dragon_archive_run(), *5*
dragon_archived_runs, 5
dragon_backend, 5, *9*, *16, 17*
dragon_backend_local *(*dragon_backend*)*, 5
dragon_backend_ollama *(*dragon_backend*)*, 5
dragon_backend_server *(*dragon_backend*)*, 5
dragon_backend_server(), *31*
dragon_bundle, 6
dragon_bundle(), *12*, *35*
dragon_cancel, 8
dragon_cancel(), *27*
dragon_chat, 8
dragon_chat(), *5*, *15*, *40*
dragon_chat_load *(*dragon_chat*)*, 8
dragon_check, 10
dragon_code, 10
dragon_code(), *16*, *45*
dragon_compare, 11
dragon_compare(), *14*, *19*
dragon_conversations, 12
dragon_conversations(), *16*
dragon_dataset, 13
dragon_delete_run *(*dragon_archive_run*)*, 4
dragon_evaluate, 13
dragon_evaluate(), *11*, *18*, *24*, *43*
dragon_example_data, 14
dragon_export_gguf, 15
dragon_feedback, 15
dragon_feedback(), *9*
dragon_generate, 16
dragon_generate(), *5*, *18*, *40*
dragon_hardware, 17
dragon_hardware(), *5*, *29*, *34*, *48*
dragon_import, 18
dragon_import(), *7*, *35*
dragon_judge, 18
dragon_judge(), *11*, *21*, *31*, *43*, *45–47*
dragon_judge_anthropic *(*dragon_llm_anthropic*)*, 20
dragon_judge_anthropic(), *19*
dragon_judge_ellmer *(*dragon_llm_anthropic*)*, 20
dragon_llm_anthropic, 20
dragon_llm_anthropic(), *26*, *45*
dragon_llm_ellmer *(*dragon_llm_anthropic*)*, 20
dragon_logs *(*dragon_status*)*, 41
dragon_lora, 21
dragon_lora(), *7*, *29*, *34*, *48*
dragon_map, 22
dragon_map(), *7*, *12, 13*, *22*, *48*
dragon_map_pairs, 22
dragon_map_pairs(), *7*, *16*, *28*, *47*
dragon_map_prompts, 23
dragon_map_prompts(), *7*, *33*
dragon_merge, 24
dragon_merge(), *15*, *18*, *43*
dragon_metric_regex *(*dragon_metrics*)*, 24
dragon_metrics, 24
dragon_metrics(), *14*
dragon_pipeline, 25
dragon_pipeline(), *42*
dragon_pipeline_cancel, 27
dragon_pipeline_status, 27
dragon_pipeline_status(), *25, 26*
dragon_prefer, 28
dragon_prefer(), *7*, *22*, *28*, *33, 34*, *43*, *47, 48*
dragon_presets, 30
dragon_presets(), *7*, *28*, *33*, *48*
dragon_preview, 30
dragon_progress *(*dragon_status*)*, 41
dragon_progress(), *18*
dragon_prompts, 31

dragon_prompts(), *44–46*
dragon_publish, 31
dragon_publish(), *43*
dragon_python_requirements, 32
dragon_reinforce, 33
dragon_reinforce(), *7*, *23*, *36*, *43*
dragon_remote, 35
dragon_remote(), *7*, *18*, *35, 36*
dragon_remote_providers, 35
dragon_remote_providers(), *35*
dragon_resume, 36
dragon_reward, 36
dragon_reward(), *7*, *33*
dragon_run, 38
dragon_run(), *48*
dragon_runs, 39
dragon_runs(), *4, 5*
dragon_runs_dir, 39
dragon_runs_dir(), *7*, *29*, *34*, *48*
dragon_serve_ollama, 40
dragon_serve_ollama(), *6*
dragon_split, 41
dragon_status, 41
dragon_status(), *18*
dragon_step, *26*, 42
dragon_step_evaluate *(*dragon_step*)*, 42
dragon_step_judge *(*dragon_step*)*, 42
dragon_step_merge *(*dragon_step*)*, 42
dragon_step_prefer *(*dragon_step*)*, 42
dragon_step_publish *(*dragon_step*)*, 42
dragon_step_reinforce *(*dragon_step*)*, 42
dragon_step_synthesize_pairs *(*dragon_step*)*, 42
dragon_step_train *(*dragon_step*)*, 42
dragon_step_train(), *12*
dragon_synthesize, 44
dragon_synthesize(), *21*, *46*
dragon_synthesize_pairs, 46
dragon_synthesize_pairs(), *21*, *31*, *43*
dragon_train, 48
dragon_train(), *6*, *12*, *28–30*, *34*, *41*, *43, 44*
dragon_train_args, 49
dragon_train_args(), *7*, *48*
dragon_unarchive_run *(*dragon_archive_run*)*, 4
dragon_unarchive_run(), *4*
dragon_wait, 50
dragon_worker_stop, 51

glue::glue(), *22*

reticulate::py_require(), *32*

shiny::shinyApp(), *3*