Package 'llamaR'

May 27, 2026

Type Package

Title Interface for Large Language Models via 'llama.cpp'

Version 0.2.4

Description Provides 'R' bindings to 'llama.cpp' for running Large Language Models ('LLMs') locally with optional 'Vulkan' GPU acceleration via 'ggmlR'. Supports model loading, text generation, 'tokenization', token-to-piece conversion, 'embeddings' (single and batch), encoder-decoder inference, low-level batch management, chat templates, 'LoRA' adapters, explicit backend/device selection, multi-GPU split, and 'NUMA' optimization. Includes a high-level 'ragnar'-compatible embedding provider ('embed_llamar'). Built on top of 'ggmlR' for efficient tensor operations.

License MIT + file LICENSE

URL https://github.com/Zabis13/llamaR

BugReports https://github.com/Zabis13/llamaR/issues

Encoding UTF-8

Depends R (>= 4.1.0), ggmlR

LinkingTo ggmlR

SystemRequirements C++17, GNU make

Imports jsonlite, utils

Suggests testthat (>= 3.0.0), withr, drogonR, later, ellmer, callr, knitr, rmarkdown

VignetteBuilder knitr

RoxygenNote 7.3.3

Config/testthat/edition 3

NeedsCompilation yes

Author Yuri Baramykov [aut, cre] (ORCID: <https://orcid.org/0009-0000-7627-4217>), Georgi Gerganov [cph] (Author of the 'llama.cpp' library included in src/)

Maintainer Yuri Baramykov <lbsbmsu@mail.ru>

Repository CRAN

Date/Publication 2026-05-27 17:30:02 UTC

Contents

`chat_llamar` *Chat with a local model through an ellmer::Chat object*

Description

Returns an **ellmer** `Chat` object backed by a local GGUF model, so the whole ellmer / ragnar toolchain (turns, tools, streaming, structured output, `ragnar_register_tool_retrieve()`, `...`) works against local inference. Transport is the OpenAI-compatible HTTP API from `llama_serve_openai`; this function is a thin `chat_vllm` wrapper over it. (We use the vLLM provider because it speaks `/v1/chat/completions` — the de-facto standard our server implements — whereas ellmer's `chat_openai`/`chat_openai_compatible` target OpenAI's newer `/v1/responses`.)

Usage

```
chat_llamar(
  model_path = NULL,
  base_url = NULL,
  port = 11434L,
  n_ctx = 4096L,
  n_gpu_layers = -1L,
  model_id = NULL,
  system_prompt = NULL,
  timeout = 180,
  ...
)
```

Arguments

`model_path`	Path to a GGUF model file. Spawns a server (mode A). Mutually exclusive with `base_url`.
`base_url`	Base URL of a running OpenAI-compatible server, e.g. `"http://127.0.0.1:11434/v1"`. Connects to it (mode B). Mutually exclusive with `model_path`.
`port`	Port for the spawned server (mode A only). Default `11434`.
`n_ctx, n_gpu_layers`	Passed to `llama_serve_openai` when spawning (mode A only).
`model_id`	Model identifier reported to ellmer. Defaults to the model file's base name in mode A; `"llamar"` in mode B.
`system_prompt`	Optional system prompt for the chat.
`timeout`	Seconds to wait for a spawned server to accept connections before erroring (mode A only). Default `180` — large models (e.g. a 14B at Q8) can take a couple of minutes to load from disk.
`...`	Passed on to `chat_vllm`.

Details

Two modes, picked by which argument you pass (DBI-style — like `DBI::dbConnect()` accepting either connection parameters or a ready connection):

`base_url` Connect to a server you already started (e.g. `llama_serve_openai()` in another process, or a worker pool). No process is spawned.

`model_path` Spin up `llama_serve_openai()` in a background R process (via **callr**), wait for it to come up, and return a `Chat` pointed at it. The server process's lifetime is tied to the returned object: when it is garbage-collected (or R exits), the process is killed. Stop it eagerly with `chat_llamar_stop`.

Exactly one of `base_url` or `model_path` must be supplied.

Value

An **ellmer** `Chat` object. In mode A it additionally carries the background process handle (see `chat_llamar_stop`).

Concurrency

The server is single-sequence (one request at a time); see `llama_serve_openai`. For parallel sessions, run a pool of servers on different ports and create one `chat_llamar(base_url=)` per worker.

Tool calls

Tool calling and structured output are mediated by the OpenAI protocol, so they work only as far as the server implements them. The current server does not emit `tool_calls` yet (see TODO), so ellmer tools registered on the returned chat will not be invoked by the model.

See Also

[llama_serve_openai], [chat_llamar_stop]

Examples

```
## Not run:
# Mode A: spawn a server for this model and chat with it.
chat <- chat_llamar(model_path = "model.gguf")
chat$chat("Why is the sky blue?")
chat_llamar_stop(chat)            # or let GC do it

# Mode B: connect to a server you already run.
llama_serve_openai("model.gguf", port = 11434L)   # in another process
chat <- chat_llamar(base_url = "http://127.0.0.1:11434/v1")
chat$chat("Hello!")

## End(Not run)
```

chat_llamar_stop *Stop the server spawned by chat_llamar()*

Description

Kills the background `llama_serve_openai` process that `chat_llamar` started in mode A. A no-op for chats created in mode B (`base_url=`), which own no process. Safe to call more than once.

Usage

```
chat_llamar_stop(chat)
```

Arguments

`chat`	A `Chat` object returned by `chat_llamar`.

Value

Invisibly `TRUE` if a process was killed, `FALSE` otherwise.

See Also

[chat_llamar]

embed_llamar *Embedding provider for ragnar / standalone use*

Description

Computes embeddings using a local GGUF model. When called without `x`, returns a function suitable for passing to `ragnar_store_create(embed = ...)`.

Usage

```
embed_llamar(
  x,
  model,
  n_gpu_layers = 0L,
  n_ctx = 512L,
  n_threads = parallel::detectCores(),
  embedding = FALSE,
  normalize = TRUE
)
```

Arguments

x	Character vector of texts to embed, a data.frame with a `text` column, or missing/`NULL` for partial application.
model	Either a path to a `.gguf` file (character) or a model handle already loaded via `llama_load_model`.
n_gpu_layers	Number of layers to offload to GPU (0 = CPU only, -1 = all). Ignored when `model` is an already-loaded handle.
n_ctx	Context window size for the embedding context. Defaults to 512, typical for embedding models. Ignored when `model` is an already-loaded handle.
n_threads	Number of CPU threads. Ignored when `model` is an already-loaded handle.
embedding	Logical; if `TRUE`, use pooled batch decode (efficient for true embedding models like nomic-embed, bge). If `FALSE` (default), use sequential last-token decode (works with any model).
normalize	Logical; if `TRUE` (default), L2-normalize each embedding vector.

Value

- If `x` is missing or `NULL`: a function `function(x)` that returns a list of numeric vectors (one per input string), suitable for ragnar.
- If `x` is a character vector: a numeric matrix with `nrow = length(x)` and `ncol = n_embd`.
- If `x` is a data.frame: the same data.frame with an added `embedding` column (list of numeric vectors).

Examples

```
## Not run:
# --- Partial application for ragnar ---
store <- ragnar_store_create(
  "my_store",
  embed = embed_llamar(model = "embedding-model.gguf", n_gpu_layers = -1)
)

# --- Direct use with path ---
mat <- embed_llamar(c("hello", "world"), model = "embedding-model.gguf")

# --- Direct use with pre-loaded model ---
mdl <- llama_load_model("embedding-model.gguf", n_gpu_layers = -1)
mat <- embed_llamar(c("hello", "world"), model = mdl)

## End(Not run)
```

llama_backend_devices *List available backend devices*

Description

Returns a data.frame of all compute devices (CPU, GPU, etc.) detected by the ggml backend. Use device names from this list in the `devices` parameter of `llama_load_model`.

Usage

```
llama_backend_devices()
```

Value

A data.frame with columns `name`, `description`, and `type` (one of `"cpu"`, `"gpu"`, `"igpu"`, `"accel"`).

Examples

```
# List available compute devices and pick GPU names for llama_load_model()
devs <- llama_backend_devices()
print(devs)
gpu_names <- devs$name[devs$type == "GPU"]
```

llama_batch_free *Free a llama batch allocated with* `llama_batch_init()`

Description

Free a llama batch allocated with `llama_batch_init()`

Usage

```
llama_batch_free(batch)
```

Arguments

`batch`	An external pointer returned by `llama_batch_init()`.

Value

`NULL` invisibly.

Examples

```
## Not run:
batch <- llama_batch_init(512L)
llama_batch_free(batch)

## End(Not run)
```

llama_batch_init *Initialise a llama batch*

Description

Allocates a `llama_batch` that can hold up to `n_tokens` tokens. Use `llama_batch_free()` to release the memory when done.

Usage

```
llama_batch_init(n_tokens, embd = 0L, n_seq_max = 1L)
```

Arguments

`n_tokens`	Maximum number of tokens in the batch.
`embd`	Embedding size; 0 means token-ID mode (normal inference).
`n_seq_max`	Maximum number of sequences per token.

Value

An external pointer to the allocated batch.

Examples

```
## Not run:
batch <- llama_batch_init(512L)
llama_batch_free(batch)

## End(Not run)
```

llama_chat_apply_template *Apply chat template to messages*

Description

Formats a conversation using the specified chat template. This is essential for instruct/chat models to work correctly.

Usage

```
llama_chat_apply_template(
  messages,
  template = NULL,
  add_generation_prompt = TRUE
)
```

Arguments

messages	List of messages, each with 'role' and 'content' elements. Roles are typically "system", "user", "assistant".
template	Template string (from [llama_chat_template]) or NULL to use default
add_generation_prompt	Whether to add the assistant prompt prefix at the end

Value

A character scalar containing the formatted prompt string, ready to be passed to `llama_generate`.

Examples

```
## Not run:
model <- llama_load_model("llama-3.2-instruct.gguf")
tmpl <- llama_chat_template(model)

messages <- list(
  list(role = "system", content = "You are a helpful assistant."),
  list(role = "user", content = "What is R?")
)

prompt <- llama_chat_apply_template(messages, template = tmpl)
cat(prompt)

ctx <- llama_new_context(model)
response <- llama_generate(ctx, prompt)

## End(Not run)
```

---

llama_chat_builtin_templates

List built-in chat templates

Description

Returns a character vector of all chat template names supported by llama.cpp.

Usage

```
llama_chat_builtin_templates()
```

Value

A character vector of built-in template names.

Examples

```
# See which chat template formats are supported out of the box
templates <- llama_chat_builtin_templates()
head(templates)
```

`llama_chat_template` *Get model's built-in chat template*

Description

Returns the chat template string embedded in the model file, if any. Common templates include ChatML, Llama, Mistral, etc.

Usage

```
llama_chat_template(model, name = NULL)
```

Arguments

`model`	Model handle returned by [llama_load_model]
`name`	Optional template name (NULL for default)

Value

A character scalar with the chat template string, or `NULL` if the model does not contain a built-in template.

Examples

```
## Not run:
model <- llama_load_model("llama-3.2-instruct.gguf")
tmpl <- llama_chat_template(model)
cat(tmpl)

## End(Not run)
```

llama_detokenize *Detokenize token IDs back to text*

Description

Detokenize token IDs back to text

Usage

```
llama_detokenize(ctx, tokens)
```

Arguments

ctx	Context handle returned by [llama_new_context]
tokens	Integer vector of token IDs (as returned by [llama_tokenize])

Value

A character scalar containing the decoded text.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)

# Round-trip: text -> tokens -> text
original <- "Hello, world!"
tokens <- llama_tokenize(ctx, original, add_special = FALSE)
restored <- llama_detokenize(ctx, tokens)
identical(original, restored)  # TRUE

## End(Not run)
```

llama_embeddings *Extract embeddings for a text*

Description

Runs the model in embeddings mode and returns the hidden-state vector of the last token. Note: meaningful only for models that support embeddings.

Usage

```
llama_embeddings(ctx, text)
```

Arguments

ctx	Context handle returned by [llama_new_context]
text	Character string to embed

Value

A numeric vector of length n_embd (the model's embedding dimension) containing the hidden-state representation of the input text.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)

emb1 <- llama_embeddings(ctx, "Hello world")
emb2 <- llama_embeddings(ctx, "Hi there")

# Cosine similarity
similarity <- sum(emb1 * emb2) / (sqrt(sum(emb1^2)) * sqrt(sum(emb2^2)))
cat("Similarity:", similarity, "\n")

## End(Not run)
```

---

llama_embed_batch *Batch embeddings for multiple texts*

Description

Computes embeddings for a character vector of texts in a single decode pass using per-sequence pooling. This is more efficient than calling llama_embeddings in a loop when embedding many texts.

Usage

```
llama_embed_batch(ctx, texts)
```

Arguments

ctx	Context handle returned by [llama_new_context]
texts	Character vector of texts to embed

Details

Requires a model that supports pooled embeddings (e.g. embedding models like nomic-embed, bge, etc.). The context must have enough capacity for the total number of tokens across all texts. Causal attention is automatically disabled during computation.

Value

A numeric matrix with `nrow = length(texts)` and `ncol = n_embd`.

Examples

```
## Not run:
model <- llama_load_model("embedding-model.gguf")
ctx <- llama_new_context(model, n_ctx = 2048L)
llama_set_causal_attn(ctx, FALSE)

mat <- llama_embed_batch(ctx, c("hello world", "foo bar", "test"))
# mat is a 3 x n_embd matrix

## End(Not run)
```

---

llama_encode *Encode tokens using the encoder (encoder-decoder models only)*

Description

Runs the encoder pass for encoder-decoder architectures (e.g. T5, BART). The encoder output is stored internally and used by subsequent decoder calls.

Usage

```
llama_encode(ctx, tokens)
```

Arguments

`ctx`	A context pointer (llama_context).
`tokens`	Integer vector of token IDs to encode.

Value

Integer return code (0 = success, negative = error).

Examples

```
## Not run:
model  <- llama_load_model("t5-model.gguf")
ctx    <- llama_new_context(model)
tokens <- llama_tokenize(ctx, "Hello world")
llama_encode(ctx, tokens)

## End(Not run)
```

llama_free_context *Free an inference context*

Description

Free an inference context

Usage

```
llama_free_context(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects. Releases the memory associated with the inference context.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)
# ... use context ...
llama_free_context(ctx)

## End(Not run)
```

llama_free_model *Free a loaded model*

Description

Free a loaded model

Usage

```
llama_free_model(model)
```

Arguments

model	Model handle returned by [llama_load_model]

Value

No return value, called for side effects. Releases the memory associated with the model.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
# ... use model ...
llama_free_model(model)

## End(Not run)
```

`llama_generate` *Generate text from a prompt*

Description

Tokenizes the prompt, runs the full autoregressive decode loop with sampling, and returns the generated text (excluding the original prompt).

Usage

```
llama_generate(
  ctx,
  prompt,
  max_new_tokens = 256L,
  temp = 0.8,
  top_k = 50L,
  top_p = 0.9,
  seed = 42L,
  min_p = 0,
  typical_p = 1,
  repeat_penalty = 1,
  repeat_last_n = 64L,
  frequency_penalty = 0,
  presence_penalty = 0,
  mirostat = 0L,
  mirostat_tau = 5,
  mirostat_eta = 0.1,
  grammar = NULL,
  with_timings = FALSE
)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`prompt`	Character string prompt
`max_new_tokens`	Maximum number of tokens to generate
`temp`	Sampling temperature. 0 = greedy decoding.
`top_k`	Top-K filtering (0 = disabled)

top_p	Top-P (nucleus) filtering (1.0 = disabled)
seed	Random seed for sampling
min_p	Min-P filtering threshold (0.0 = disabled)
typical_p	Locally typical sampling threshold (1.0 = disabled)
repeat_penalty	Repetition penalty (1.0 = disabled)
repeat_last_n	Number of last tokens to penalize (0 = disabled, -1 = context size)
frequency_penalty	Frequency penalty (0.0 = disabled)
presence_penalty	Presence penalty (0.0 = disabled)
mirostat	Mirostat sampling mode (0 = disabled, 1 = Mirostat, 2 = Mirostat 2.0)
mirostat_tau	Mirostat target entropy (tau parameter)
mirostat_eta	Mirostat learning rate (eta parameter)
grammar	GBNF grammar string for constrained generation (NULL = disabled)
with_timings	If TRUE, attach a named numeric vector of per-stage timings (in ms) as attribute "timings" of the returned text. Stages: tokenize, build_sampler, kv_clear, prefill_dispatch, prefill_sync, gpu_sync (cumulative across decode-loop iterations), sample (cumulative), decode_dispatch (cumulative), detokenize, plus n_iterations and t_total_ms. Adds llama_synchronize calls inside the loop, so it is intended for profiling and may slightly slow generation.

Value

A character scalar containing the generated text (excluding the original prompt).

Examples

```
## Not run:
model <- llama_load_model("model.gguf", n_gpu_layers = -1L)
ctx <- llama_new_context(model, n_ctx = 2048L)

# Basic generation
result <- llama_generate(ctx, "Once upon a time")
cat(result)

# Greedy decoding (deterministic)
result <- llama_generate(ctx, "The answer is", temp = 0)

# More creative output
result <- llama_generate(ctx, "Write a poem about R:",
                         max_new_tokens = 100L,
                         temp = 1.0, top_p = 0.95)

# With repetition penalty
result <- llama_generate(ctx, "List items:",
                         repeat_penalty = 1.1, repeat_last_n = 64L)
```

```
# JSON output with grammar
result <- llama_generate(ctx, "Output JSON:",
                         grammar = 'root ::= "{" "}" ')

## End(Not run)
```

`llama_generate_batch` *Generate completions for multiple prompts in parallel*

Description

Runs continuous batching: all prompts share the same decode loop, so each iteration dispatches one matmul over all still-running sequences. This converts decode from memory-bound vector ops into compute-bound matrix ops on the GPU and lifts throughput compared to calling `llama_generate` in a loop.

Usage

```
llama_generate_batch(
  ctx,
  prompts,
  max_new_tokens = 256L,
  temp = 0.8,
  top_k = 50L,
  top_p = 0.9,
  seed = 42L,
  min_p = 0,
  typical_p = 1,
  repeat_penalty = 1,
  repeat_last_n = 64L,
  frequency_penalty = 0,
  presence_penalty = 0,
  grammar = NULL
)
```

Arguments

`ctx`	Context handle returned by [llama_new_context], created with sufficient `n_seq_max` and `n_ctx` (see Details).
`prompts`	Character vector of prompts, one per parallel sequence.
`max_new_tokens, temp, top_k, top_p, seed, min_p, typical_p, repeat_penalty, repeat_last_n, frequency_penalty, presence_penalty, grammar`	Sampling parameters; see `llama_generate`. Shared across sequences. `seed` is offset per sequence (`seed + s`).

Details

The context must be created with `n_seq_max >= length(prompts)` and `n_ctx` large enough to hold every prompt plus its generated tokens simultaneously. As a rule of thumb: `n_ctx >= sum(prompt_lengths) + length(prompts) * max_new_tokens`.

Each sequence gets its own sampler chain seeded with `seed + seq_index`, so identical prompts still produce diverse outputs at `temp > 0` (useful for self-consistency sampling). Sampler hyperparameters are shared across sequences in this version.

Stop conditions per sequence: end-of-generation token (model-defined) or `max_new_tokens` reached. Mirostat and `with_timings` are not supported here yet — use `llama_generate` for those.

Value

A list of length `length(prompts)`, in the same order as the input. Each element is a list with fields:

- `text`: character scalar with the generated text
- `n_tokens`: integer count of tokens generated
- `finished_reason`: `"eos"` or `"max_tokens"`

Examples

```
## Not run:
model <- llama_load_model("model.gguf", n_gpu_layers = -1L)
# 4 parallel sequences, up to 256 new tokens each
ctx <- llama_new_context(model, n_ctx = 4096L, n_seq_max = 4L,
                         flash_attn = "on")

# Batch classification
prompts <- c("Classify: 'great movie' as positive/negative.",
             "Classify: 'awful service' as positive/negative.",
             "Classify: 'just okay' as positive/negative.",
             "Classify: 'loved every minute' as positive/negative.")
out <- llama_generate_batch(ctx, prompts, max_new_tokens = 16L, temp = 0)
vapply(out, `[[`, character(1), "text")

# Self-consistency sampling: same prompt repeated
samples <- llama_generate_batch(ctx, rep("2 + 2 =", 4L),
                                max_new_tokens = 8L, temp = 0.7)

## End(Not run)
```

---

llama_gen_begin *Begin a streaming (token-by-token) generation*

Description

Sets up sampling and prefills the prompt, returning an opaque state handle that is pulled one chunk at a time with [llama_gen_next]. This is the streaming counterpart to [llama_generate]: same sampler chain and the same output for a given seed, but text arrives incrementally so it can be pushed into an SSE stream as it is produced.

Usage

```
llama_gen_begin(
  ctx,
  prompt,
  max_new_tokens = 256L,
  temp = 0.8,
  top_k = 50L,
  top_p = 0.9,
  seed = 42L,
  min_p = 0,
  typical_p = 1,
  repeat_penalty = 1,
  repeat_last_n = 64L,
  frequency_penalty = 0,
  presence_penalty = 0,
  mirostat = 0L,
  mirostat_tau = 5,
  mirostat_eta = 0.1,
  grammar = NULL
)
```

Arguments

ctx	Context handle returned by [llama_new_context]
prompt	Character string prompt
max_new_tokens	Maximum number of tokens to generate
temp	Sampling temperature. 0 = greedy decoding.
top_k	Top-K filtering (0 = disabled)
top_p	Top-P (nucleus) filtering (1.0 = disabled)
seed	Random seed for sampling
min_p	Min-P filtering threshold (0.0 = disabled)
typical_p	Locally typical sampling threshold (1.0 = disabled)
repeat_penalty	Repetition penalty (1.0 = disabled)
repeat_last_n	Number of last tokens to penalize (0 = disabled, -1 = context size)
frequency_penalty	Frequency penalty (0.0 = disabled)
presence_penalty	Presence penalty (0.0 = disabled)
mirostat	Mirostat sampling mode (0 = disabled, 1 = Mirostat, 2 = Mirostat 2.0)
mirostat_tau	Mirostat target entropy (tau parameter)
mirostat_eta	Mirostat learning rate (eta parameter)
grammar	GBNF grammar string for constrained generation (NULL = disabled)

Details

Typical loop:

```
st <- llama_gen_begin(ctx, prompt)
repeat {
  chunk <- llama_gen_next(st)
  if (is.null(chunk)) break
  cat(chunk)
}
cat(llama_gen_end(st))  # flush any held-back trailing bytes
```

Only one streaming generation may be active per context at a time: each call to `llama_gen_begin` clears the context KV cache.

Value

An external pointer holding the generation state. Pass it to [llama_gen_next] and [llama_gen_end]. The underlying sampler is freed automatically by the garbage collector.

See Also

[llama_gen_next], [llama_gen_end], [llama_generate]

`llama_gen_end` *Finish a streaming generation*

Description

Marks the generation done and returns any bytes still held in the internal UTF-8 carry buffer (the tail of an unfinished character, if generation stopped mid-character). Concatenating every [llama_gen_next] chunk followed by the `llama_gen_end` result reproduces the full [llama_generate] output for the same seed and parameters. Safe to call more than once.

Usage

```
llama_gen_end(state)
```

Arguments

`state`	Generation state handle from [llama_gen_begin].

Value

A length-1 UTF-8 character vector with any remaining buffered text (often `""`).

See Also

[llama_gen_begin], [llama_gen_next]

llama_gen_next *Pull the next chunk of a streaming generation*

Description

Advances a generation started with [llama_gen_begin] by one token and returns the next chunk of decoded text. A possibly-incomplete trailing UTF-8 character is held back until enough bytes arrive, so every returned chunk is valid UTF-8 (the chunk may be `""` when the only new byte is part of an unfinished character).

Usage

```
llama_gen_next(state)
```

Arguments

`state`	Generation state handle from [llama_gen_begin].

Value

A length-1 UTF-8 character vector with the next chunk, or `NULL` when generation has finished (end-of-generation token reached or `max_new_tokens` exhausted). After `NULL`, call [llama_gen_end] to flush any remaining bytes.

See Also

[llama_gen_begin], [llama_gen_end]

llama_get_embeddings *Get all output token embeddings as a matrix*

Description

Returns a matrix of shape `n_outputs` × `n_embd` containing the raw embedding vectors for all tokens whose `logits` flag was set in the batch. Only works when `pooling_type == "none"` (generative models or embedding contexts without pooling). For pooled embeddings use [llama_get_embeddings_seq].

Usage

```
llama_get_embeddings(ctx, n_outputs)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`n_outputs`	Number of outputs requested in the last decode call (i.e. how many tokens had `logits = TRUE` in the batch).

Value

A numeric matrix with `n_outputs` rows and `n_embd` columns.

`llama_get_embeddings_ith` *Get embeddings for the i-th token in the batch*

Description

Returns the embedding vector for a specific token position after a decode call with embeddings enabled. Negative indices count from the end (-1 = last token).

Usage

```
llama_get_embeddings_ith(ctx, i)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`i`	Integer index of the token (0-based, or negative for reverse indexing)

Value

A numeric vector of length `n_embd`.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)
llama_generate(ctx, "Hello world", max_new_tokens = 1L)

# Get the embedding of the last decoded token
emb <- llama_get_embeddings_ith(ctx, -1L)
cat("Embedding dim:", length(emb), "\n")

## End(Not run)
```

llama_get_embeddings_seq *Get pooled embeddings for a sequence*

Description

Returns the pooled embedding vector for a given sequence ID after a batch decode. Only works when the model supports pooling (embedding models).

Usage

```
llama_get_embeddings_seq(ctx, seq_id)
```

Arguments

ctx	Context handle returned by [llama_new_context] with embedding = TRUE
seq_id	Integer sequence ID (0-based)

Value

A numeric vector of length n_embd.

Examples

```
## Not run:
# Get pooled embedding for sequence 0 (requires embedding context)
model <- llama_load_model("nomic-embed.gguf")
ctx <- llama_new_context(model, embedding = TRUE)
mat <- llama_embed_batch(ctx, "Hello world")
emb <- llama_get_embeddings_seq(ctx, 0L)
cat("Pooled embedding dim:", length(emb), "\n")

## End(Not run)
```

llama_get_logits *Get logits from the last decode step*

Description

Returns the raw logit vector (unnormalized log-probabilities) from the last token position after a decode operation.

Usage

```
llama_get_logits(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A numeric vector of length n_vocab containing the logits.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)
result <- llama_generate(ctx, "The capital of France is", max_new_tokens = 1L)
logits <- llama_get_logits(ctx)
# Find top token
top_id <- which.max(logits)

## End(Not run)
```

---

llama_get_logits_ith *Get logits for a specific token position*

Description

Returns the logit vector for token at index i in the last decoded batch. Use i = -1 to get the logits for the last token.

Usage

```
llama_get_logits_ith(ctx, i)
```

Arguments

ctx	Context handle returned by [llama_new_context]
i	Integer index into the last batch (0-based). Use -1 for the last token.

Value

A numeric vector of length n_vocab.

llama_get_model *Get the model associated with a context*

Description

Returns the model handle that was used to create this context. The returned object is the same R external pointer that was passed to [llama_new_context] — no new allocation occurs.

Usage

```
llama_get_model(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A model handle (external pointer), equivalent to the original handle returned by [llama_load_model].

llama_get_verbosity *Get current verbosity level*

Description

Get current verbosity level

Usage

```
llama_get_verbosity()
```

Value

An integer scalar indicating the current verbosity level (0 = silent, 1 = errors only, 2 = normal, 3 = verbose).

Examples

```
# Save current level, suppress output, then restore
old <- llama_get_verbosity()
llama_set_verbosity(0)
# ... noisy operations ...
llama_set_verbosity(old)
```

llama_hf_cache_clear *Clear the model cache*

Description

Removes cached model files. Can clear the entire cache or only files from a specific repository.

Usage

```
llama_hf_cache_clear(repo_id = NULL, confirm = TRUE, cache_dir = NULL)
```

Arguments

repo_id	Character or NULL. If specified, only remove cached files from this repository. If NULL, clear the entire cache.
confirm	Logical. If TRUE (default), ask for confirmation before deleting files in interactive sessions.
cache_dir	Character or NULL. Cache directory to clear. Defaults to llama_hf_cache_dir().

Value

Invisible NULL. Called for its side effect of deleting cached files.

Examples

```
llama_hf_cache_clear(confirm = FALSE)
```

llama_hf_cache_dir *Get the cache directory for downloaded models*

Description

Returns the path to the directory where models downloaded from Hugging Face are cached. The directory is created if it does not exist.

Usage

```
llama_hf_cache_dir()
```

Value

A character string containing the absolute path to the cache directory. The path follows the R user directory convention via R_user_dir.

Examples

```
llama_hf_cache_dir()
```

llama_hf_cache_info *Show information about the model cache*

Description

Lists all cached model files with their sizes and download metadata.

Usage

```
llama_hf_cache_info(cache_dir = NULL)
```

Arguments

cache_dir	Character or NULL. Cache directory to inspect. Defaults to llama_hf_cache_dir().

Value

A data frame with columns:

repo_id Character. The Hugging Face repository identifier.

filename Character. The model file name.

size Numeric. File size in bytes.

size_pretty Character. Human-readable file size.

path Character. Absolute path to the cached file.

downloaded_at Character. Timestamp of when the file was downloaded.

Returns an empty data frame with the same columns if the cache is empty.

Examples

```
llama_hf_cache_info()
```

llama_hf_download *Download a GGUF model from Hugging Face*

Description

Downloads a GGUF model file from a Hugging Face repository. Files are cached locally so subsequent calls return the cached path without re-downloading.

Usage

```
llama_hf_download(
  repo_id,
  filename = NULL,
  pattern = NULL,
  tag = NULL,
  token = NULL,
  cache_dir = NULL,
  revision = "main",
  force = FALSE
)
```

Arguments

repo_id	Character. Hugging Face repository in "org/repo" format.
filename	Character or NULL. Exact filename to download.
pattern	Character or NULL. Glob pattern for filename matching (case-insensitive). If multiple files match, an error is thrown listing the matches.
tag	Character or NULL. Ollama-style tag. First tries the Ollama manifest API; on failure, falls back to pattern matching with *{tag}*.
token	Character or NULL. Hugging Face API token. If NULL, uses the HF_TOKEN environment variable.
cache_dir	Character or NULL. Custom cache directory. Defaults to llama_hf_cache_dir().
revision	Character. Git revision (branch/tag/commit). Defaults to "main".
force	Logical. If TRUE, re-download even if cached. Defaults to FALSE.

Details

Exactly one of filename, pattern, or tag must be specified to identify which file to download.

Value

A character string containing the absolute path to the downloaded (or cached) GGUF model file.

Examples

```
## Not run:
path <- llama_hf_download("TheBloke/Llama-2-7B-GGUF",
                          pattern = "*q2_k*")
print(path)

## End(Not run)
```

llama_hf_list *List GGUF files in a Hugging Face repository*

Description

Queries the Hugging Face API for GGUF model files in the specified repository. Returns a data frame with file names, sizes, and detected quantization levels.

Usage

```
llama_hf_list(repo_id, token = NULL, pattern = NULL)
```

Arguments

repo_id	Character. Hugging Face repository in "org/repo" format, e.g. "TheBloke/Llama-2-7B-GGUF".
token	Character or NULL. Hugging Face API token. If NULL, uses the HF_TOKEN environment variable.
pattern	Character or NULL. Optional glob pattern to filter results (e.g. "*q4_k_m*"). Case-insensitive.

Value

A data frame with columns:

filename Character. The file name within the repository.

size Numeric. File size in bytes.

size_pretty Character. Human-readable file size.

quant Character. Detected quantization level (e.g. "Q4_K_M") or NA if not detected.

Examples

```
files <- llama_hf_list("TheBloke/Llama-2-7B-GGUF")
print(files)
```

llama_load_model *Load a GGUF model file*

Description

Load a GGUF model file

Usage

```
llama_load_model(
  path,
  n_gpu_layers = -1L,
  devices = NULL,
  split_mode = "layer",
  use_mmap = TRUE,
  use_mlock = FALSE
)
```

Arguments

path	Path to the .gguf model file
n_gpu_layers	Number of layers to offload to GPU (-1L = all, 0L = CPU only). Default -1L offloads everything to the GPU when one is detected; if no GPU backend is available, falls back to CPU with a warning.
devices	Character vector of device names or types to use for offloading. NULL (default) uses all available devices. Use "cpu" for CPU-only, "gpu" for first GPU, or specific device names from llama_backend_devices. Multiple devices enable multi-GPU split.
split_mode	Multi-GPU split strategy: "none" (single GPU), "layer" (split layers across GPUs, default), or "row" (tensor-parallel across GPUs).
use_mmap	Logical; map model file into memory (default TRUE).
use_mlock	Logical; force the OS to keep model pages resident (default FALSE).

Value

An external pointer (class externalptr) wrapping the loaded model. This handle is required by llama_new_context, llama_model_info, and other model-level functions. Freed automatically by the garbage collector or manually via llama_free_model.

Examples

```
## Not run:
# Default: full GPU offload (falls back to CPU if no GPU)
model <- llama_load_model("model.gguf")

# Force CPU-only
model <- llama_load_model("model.gguf", n_gpu_layers = 0L)

# Explicit CPU-only backend
model <- llama_load_model("model.gguf", devices = "cpu")

# Specific GPU device (see llama_backend_devices())
model <- llama_load_model("model.gguf", n_gpu_layers = -1L, devices = "Vulkan0")

# Multi-GPU: use two devices
model <- llama_load_model("model.gguf", n_gpu_layers = -1L,
```

```
                          devices = c("Vulkan0", "Vulkan1"))

## End(Not run)
```

`llama_load_model_hf` *Load a model directly from Hugging Face*

Description

Convenience function that downloads a GGUF model from Hugging Face (if not already cached) and loads it via `llama_load_model`.

Usage

```
llama_load_model_hf(repo_id, ..., n_gpu_layers = 0L)
```

Arguments

`repo_id`	Character. Hugging Face repository in `"org/repo"` format.
`...`	Additional arguments passed to `llama_hf_download` (e.g. `pattern`, `cache_dir`, `force`).
`n_gpu_layers`	Integer. Number of layers to offload to GPU. Use `-1L` for all layers. Defaults to `0L` (CPU only).

Value

An external pointer to the loaded model, as returned by `llama_load_model`.

Examples

```
## Not run:
model <- llama_load_model_hf("TheBloke/Llama-2-7B-GGUF",
                             pattern = "*q2_k*")

## End(Not run)
```

llama_lora_apply *Apply a LoRA adapter to context*

Description

Activates a loaded LoRA adapter for the given context. Multiple LoRA adapters can be applied simultaneously.

Usage

```
llama_lora_apply(ctx, lora, scale = 1)
```

Arguments

ctx	Context handle returned by [llama_new_context]
lora	LoRA adapter handle from [llama_lora_load]
scale	Scaling factor for the adapter (1.0 = full effect, 0.5 = half effect)

Value

No return value, called for side effects. Activates the LoRA adapter for the given context.

Examples

```
## Not run:
model <- llama_load_model("base-model.gguf")
lora <- llama_lora_load(model, "adapter.gguf")
ctx <- llama_new_context(model)

# Apply with full strength
llama_lora_apply(ctx, lora, scale = 1.0)

# Or apply with reduced effect
llama_lora_apply(ctx, lora, scale = 0.5)

## End(Not run)
```

llama_lora_clear *Remove all LoRA adapters from context*

Description

Deactivates all LoRA adapters from the context, returning to base model behavior.

Usage

```
llama_lora_clear(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects. Removes all active LoRA adapters from the context.

Examples

```
## Not run:
# Apply multiple LoRAs
llama_lora_apply(ctx, lora1)
llama_lora_apply(ctx, lora2)

# Remove all at once
llama_lora_clear(ctx)

## End(Not run)
```

---

llama_lora_load *Load a LoRA adapter*

Description

Loads a LoRA (Low-Rank Adaptation) adapter file that can be applied to modify the model's behavior without changing the base weights.

Usage

```
llama_lora_load(model, path)
```

Arguments

model	Model handle returned by [llama_load_model]
path	Path to the LoRA adapter file (.gguf or .bin)

Value

An external pointer (class `externalptr`) wrapping the loaded LoRA (Low-Rank Adaptation) adapter. Pass this handle to `llama_lora_apply` to activate the adapter.

Examples

```
## Not run:
model <- llama_load_model("base-model.gguf")
lora <- llama_lora_load(model, "fine-tuned-adapter.gguf")

ctx <- llama_new_context(model)
llama_lora_apply(ctx, lora, scale = 1.0)

# Now generation uses the LoRA-modified model
result <- llama_generate(ctx, "Hello")

## End(Not run)
```

`llama_lora_remove` *Remove a LoRA adapter from context*

Description

Deactivates a specific LoRA adapter from the context.

Usage

```
llama_lora_remove(ctx, lora)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`lora`	LoRA adapter handle to remove

Value

An integer scalar: 0 on success, -1 if the adapter was not applied to this context.

Examples

```
## Not run:
# Remove a specific adapter while keeping others active
llama_lora_remove(ctx, lora)
result <- llama_generate(ctx, "Without adapter: ", max_new_tokens = 20L)

## End(Not run)
```

llama_max_devices *Get maximum number of devices*

Description

Get maximum number of devices

Usage

```
llama_max_devices()
```

Value

An integer scalar: the maximum number of compute devices available.

Examples

```
# Query the maximum number of devices supported by the backend
n <- llama_max_devices()
cat("Max devices:", n, "\n")
```

llama_memory_breakdown_print *Print memory breakdown by device*

Description

Prints a debug summary of how model weights are distributed across compute devices (CPU, GPU layers). Useful for diagnosing memory allocation with partial GPU offload.

Usage

```
llama_memory_breakdown_print(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects.

llama_memory_can_shift *Check if the KV cache supports shifting*

Description

Check if the KV cache supports shifting

Usage

```
llama_memory_can_shift(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A logical scalar: TRUE if the memory supports position shifting.

Examples

```
## Not run:
if (llama_memory_can_shift(ctx)) {
  message("Context shifting is supported")
}

## End(Not run)
```

llama_memory_clear *Clear the KV cache*

Description

Removes all tokens from the KV cache. Call this before starting a new generation from scratch.

Usage

```
llama_memory_clear(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects.

Examples

```
## Not run:
# Clear the KV cache to start a fresh conversation
llama_memory_clear(ctx)
result <- llama_generate(ctx, "New topic: ", max_new_tokens = 50L)

## End(Not run)
```

`llama_memory_seq_add` *Shift token positions in a sequence*

Description

Adds a position delta to all tokens in the given sequence within [p0, p1). This is useful for implementing context shifting (sliding window).

Usage

```
llama_memory_seq_add(ctx, seq_id, p0, p1, delta)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`seq_id`	Sequence ID
`p0`	Start position (inclusive)
`p1`	End position (exclusive)
`delta`	Position shift amount (can be negative)

Value

No return value, called for side effects.

Examples

```
## Not run:
# Shift positions left by 100 for context window management
llama_memory_seq_add(ctx, seq_id = 0L, p0 = 100L, p1 = -1L, delta = -100L)

## End(Not run)
```

llama_memory_seq_cp *Copy a sequence in the KV cache*

Description

Copies cached tokens from one sequence to another in the position range [p0, p1).

Usage

```
llama_memory_seq_cp(ctx, seq_id_src, seq_id_dst, p0 = -1L, p1 = -1L)
```

Arguments

ctx	Context handle returned by [llama_new_context]
seq_id_src	Source sequence ID
seq_id_dst	Destination sequence ID
p0	Start position (inclusive, -1 for beginning)
p1	End position (exclusive, -1 for end)

Value

No return value, called for side effects.

Examples

```
## Not run:
# Copy sequence 0 to sequence 1
llama_memory_seq_cp(ctx, seq_id_src = 0L, seq_id_dst = 1L,
                    p0 = -1L, p1 = -1L)

## End(Not run)
```

llama_memory_seq_div *Integer-divide token positions in a sequence*

Description

Divides all token positions in the range `[p0, p1)` for the given sequence by `d`. Use `p0 = -1` and `p1 = -1` for the full range. Useful for implementing sliding-window context compression.

Usage

```
llama_memory_seq_div(ctx, seq_id, p0, p1, d)
```

Arguments

ctx	Context handle returned by [llama_new_context]
seq_id	Sequence ID
p0	Start position (inclusive). Use -1 for beginning.
p1	End position (exclusive). Use -1 for end.
d	Divisor (positive integer)

Value

No return value, called for side effects.

---

llama_memory_seq_keep *Keep only one sequence in the KV cache*

Description

Removes all sequences except the specified one from the KV cache.

Usage

```
llama_memory_seq_keep(ctx, seq_id)
```

Arguments

ctx	Context handle returned by [llama_new_context]
seq_id	Sequence ID to keep

Value

No return value, called for side effects.

Examples

```
## Not run:
llama_memory_seq_keep(ctx, seq_id = 0L)

## End(Not run)
```

llama_memory_seq_pos_range *Get position range for a sequence*

Description

Returns the minimum and maximum token positions for a given sequence in the KV cache.

Usage

```
llama_memory_seq_pos_range(ctx, seq_id)
```

Arguments

ctx	Context handle returned by [llama_new_context]
seq_id	Sequence ID

Value

A named integer vector with elements min and max.

Examples

```
## Not run:
range <- llama_memory_seq_pos_range(ctx, seq_id = 0L)
cat("Positions:", range["min"], "to", range["max"], "\n")

## End(Not run)
```

llama_memory_seq_rm *Remove tokens from a sequence in the KV cache*

Description

Removes cached tokens for the given sequence in the position range [p0, p1). Use p0 = -1 and p1 = -1 to remove all tokens for the sequence.

Usage

```
llama_memory_seq_rm(ctx, seq_id, p0 = -1L, p1 = -1L)
```

Arguments

ctx	Context handle returned by [llama_new_context]
seq_id	Sequence ID (integer)
p0	Start position (inclusive, -1 for beginning)
p1	End position (exclusive, -1 for end)

Value

A logical scalar: TRUE if tokens were successfully removed.

Examples

```
## Not run:
# Remove all tokens from sequence 0
llama_memory_seq_rm(ctx, seq_id = 0L, p0 = -1L, p1 = -1L)

## End(Not run)
```

---

llama_model_info *Get model metadata*

Description

Get model metadata

Usage

```
llama_model_info(model)
```

Arguments

model	Model handle returned by [llama_load_model]

Value

A named list with fields: - 'n_ctx_train': context size the model was trained with - 'n_embd': embedding dimension - 'n_vocab': vocabulary size - 'n_layer': number of layers - 'n_head': number of attention heads - 'n_head_kv': number of key-value attention heads (GQA) - 'desc': human-readable model description string - 'size': model size in bytes - 'n_params': number of parameters - 'has_encoder': whether the model has an encoder - 'has_decoder': whether the model has a decoder - 'is_recurrent': whether the model is recurrent (e.g. Mamba)

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
info <- llama_model_info(model)
cat("Model:", info$desc, "\n")
cat("Layers:", info$n_layer, "\n")
cat("Context:", info$n_ctx_train, "\n")
cat("Size:", info$size / 1e9, "GB\n")

## End(Not run)
```

llama_model_meta *Get all model metadata as a named character vector*

Description

Returns all key-value metadata pairs stored in the GGUF model file.

Usage

```
llama_model_meta(model)
```

Arguments

model	Model handle returned by [llama_load_model]

Value

A named character vector where names are metadata keys and values are the corresponding metadata values.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
meta <- llama_model_meta(model)
print(meta)

## End(Not run)
```

llama_model_meta_val *Get a single model metadata value by key*

Description

Get a single model metadata value by key

Usage

```
llama_model_meta_val(model, key)
```

Arguments

model	Model handle returned by [llama_load_model]
key	Character string metadata key (e.g. "general.name", "general.architecture")

Value

A character scalar with the metadata value, or NULL if the key does not exist.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
llama_model_meta_val(model, "general.name")
llama_model_meta_val(model, "general.architecture")

## End(Not run)
```

---

llama_new_context *Create an inference context*

Description

Create an inference context

Usage

```
llama_new_context(
  model,
  n_ctx = 2048L,
  n_threads = NULL,
  n_threads_batch = NULL,
  n_batch = 2048L,
  n_ubatch = 512L,
  n_seq_max = 1L,
  flash_attn = "auto",
  embedding = FALSE
)
```

Arguments

model	Model handle returned by [llama_load_model]
n_ctx	Context window size (number of tokens). 0 means use the model's trained value.
n_threads	Number of CPU threads for single-token decode. NULL (default) picks 2L when a GPU backend is available, otherwise 4L.
n_threads_batch	Number of CPU threads for batch (prompt) processing. NULL (default) inherits from n_threads.
n_batch	Logical maximum batch size submitted to a single decode call (tokens). Default 2048L matches llama.cpp.
n_ubatch	Physical micro-batch size used inside decode. Larger values improve prefill throughput on GPU at the cost of memory. Default 512L.

n_seq_max	Maximum number of parallel sequences the context can hold simultaneously (KV cache is partitioned across them). Default 1L for single-prompt use; raise to N when using llama_generate_batch with N prompts. Increasing this does not by itself enlarge the context — also size n_ctx accordingly.
flash_attn	One of "auto" (let llama.cpp decide, default), "on" (force enable Flash Attention), or "off" (disable).
embedding	Logical; if TRUE, create context in embedding mode. This enables embedding output and disables causal attention, suitable for embedding models (e.g. nomic-embed, bge). When TRUE, llama_embed_batch uses efficient pooled batch decode.

Value

An external pointer (class externalptr) wrapping the inference context. This handle is required by generation, tokenization, and embedding functions. Freed automatically by the garbage collector or manually via llama_free_context.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model, n_ctx = 4096L, n_threads = 8L)
# ... use context for generation ...
llama_free_context(ctx)
llama_free_model(model)

# Tune for GPU prefill throughput
ctx <- llama_new_context(model, n_ctx = 4096L,
                         n_ubatch = 2048L, flash_attn = "on")

# Embedding mode
emb_ctx <- llama_new_context(model, n_ctx = 512L, embedding = TRUE)
mat <- llama_embed_batch(emb_ctx, c("hello", "world"))

## End(Not run)
```

---

llama_numa_init *Initialize NUMA optimization*

Description

Call once for better performance on NUMA systems.

Usage

```
llama_numa_init(strategy = "disabled")
```

Arguments

strategy	NUMA strategy: "disabled" (default), "distribute", "isolate", "numactl", or "mirror".

Value

No return value, called for side effects.

Examples

```
## Not run:
# On multi-socket servers, distribute memory across NUMA nodes
# for better memory bandwidth during inference
llama_numa_init("distribute")

# Call before loading any models - affects all subsequent allocations
model <- llama_load_model("model.gguf", n_gpu_layers = 0L)

## End(Not run)
```

---

llama_n_batch *Get logical batch size*

Description

Get logical batch size

Usage

```
llama_n_batch(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: the logical batch size (max tokens per ‘llama_decode‘ call).

llama_n_ctx *Get context window size*

Description

Get context window size

Usage

```
llama_n_ctx(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: the context window size (number of tokens).

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model, n_ctx = 4096L)
llama_n_ctx(ctx)  # 4096

## End(Not run)
```

llama_n_ctx_seq *Get per-sequence context window size*

Description

Get per-sequence context window size

Usage

```
llama_n_ctx_seq(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: maximum context size per sequence.

llama_n_seq_max *Get maximum number of sequences*

Description

Get maximum number of sequences

Usage

```
llama_n_seq_max(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: maximum number of concurrent sequences.

llama_n_threads *Get number of threads for single-token generation*

Description

Get number of threads for single-token generation

Usage

```
llama_n_threads(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: current thread count for generation.

llama_n_threads_batch *Get number of threads for batch processing*

Description

Get number of threads for batch processing

Usage

```
llama_n_threads_batch(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: current thread count for prompt encoding.

llama_n_ubatch *Get physical micro-batch size*

Description

Get physical micro-batch size

Usage

```
llama_n_ubatch(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

An integer scalar: the physical micro-batch size.

llama_perf *Get performance statistics*

Description

Returns timing and count statistics for the current context, including prompt processing time, token generation time, and counts.

Usage

```
llama_perf(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A named list with fields: - ‘t_load_ms‘: model load time in milliseconds - ‘t_p_eval_ms‘: prompt processing time in milliseconds - ‘t_eval_ms‘: token generation time in milliseconds - ‘n_p_eval‘: number of prompt tokens processed - ‘n_eval‘: number of tokens generated - ‘n_reused‘: number of reused compute graphs

Examples

```
## Not run:
result <- llama_generate(ctx, "Hello world")
perf <- llama_perf(ctx)
cat("Prompt speed:", perf$n_p_eval / (perf$t_p_eval_ms / 1000), "tok/s\n")
cat("Generation speed:", perf$n_eval / (perf$t_eval_ms / 1000), "tok/s\n")

## End(Not run)
```

llama_perf_print *Print performance statistics to the console*

Description

Prints a formatted summary of timing and throughput statistics for the context (load time, prompt processing speed, generation speed). Output goes to the R console via the llama.cpp logging callback.

Usage

```
llama_perf_print(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects.

---

llama_perf_reset *Reset performance counters*

Description

Resets the timing and token count statistics for the context.

Usage

```
llama_perf_reset(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects.

Examples

```
## Not run:
# Reset counters before benchmarking a specific generation
llama_perf_reset(ctx)
result <- llama_generate(ctx, "Benchmark prompt", max_new_tokens = 100L)
perf <- llama_perf(ctx)
cat("Generation:", perf$n_eval / (perf$t_eval_ms / 1000), "tok/s\n")

## End(Not run)
```

llama_pooling_type *Get pooling type*

Description

Get pooling type

Usage

```
llama_pooling_type(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A character string: one of ‘"none"‘, ‘"mean"‘, ‘"cls"‘, ‘"last"‘, ‘"rank"‘, ‘"unspecified"‘.

llama_serve_openai *Serve an OpenAI-compatible HTTP API for a local model*

Description

Loads a GGUF model once and exposes it over an OpenAI-compatible HTTP API so any OpenAI client (OpenCode, ellmer, the ‘openai‘ Python SDK, ...) can talk to it. Implements ‘GET /v1/models‘ and ‘POST /v1/chat/completions‘ (both blocking and ‘stream = true‘). The HTTP/SSE layer is provided by **drogonR**; generation runs through llamaR's streaming API (llama_gen_begin / llama_gen_next / llama_gen_end).

Usage

```
llama_serve_openai(
  model_path,
  port = 11434L,
  n_ctx = 4096L,
  n_gpu_layers = -1L,
  model_id = NULL,
  host = "127.0.0.1",
  template = NULL,
  max_tokens = 512L,
  ...
)
```

Arguments

model_path	Path to a GGUF model file.
port	Port to listen on. Default 11434 (the Ollama port, so clients pointed at a local Ollama work unchanged).
n_ctx	Context size for the loaded model.
n_gpu_layers	Layers to offload to GPU (-1 = all).
model_id	Identifier reported in /v1/models and echoed in responses. Defaults to the model file's base name.
host	Address to bind. Default "127.0.0.1" (local only).
template	Chat template string, or NULL to use the model's built-in template.
max_tokens	Default max_new_tokens when a request omits it.
...	Reserved for future options.

Details

The server is single-sequence: requests are handled one at a time on the main R thread (each streamed token is one event-loop pump). This is meant for a single local user/agent, not concurrent load.

drogonR is an optional dependency (Suggests); install it with install.packages("drogonR") (or from its repository) before calling this function.

Value

Invisibly NULL. Blocks serving until drogonR::dr_stop() is called (typically from another process or an interrupt).

See Also

[llama_gen_begin], [llama_generate]

Examples

```
## Not run:
llama_serve_openai("model.gguf", port = 11434L)
# In another shell, point any OpenAI client at
#   http://127.0.0.1:11434/v1
# e.g. GET /v1/models and POST /v1/chat/completions

## End(Not run)
```

`llama_set_abort_callback` *Set or clear the abort callback*

Description

Registers an R function that is called periodically during generation. If the function returns 'TRUE', the current decode operation is aborted. Pass 'NULL' to remove the callback.

Usage

```
llama_set_abort_callback(ctx, fn)
```

Arguments

`ctx`	Context handle returned by [llama_new_context]
`fn`	A zero-argument R function returning a logical scalar, or 'NULL' to clear.

Details

Note: only one callback is active globally — setting a new one replaces the previous one across all contexts.

Value

No return value, called for side effects.

Examples

```
## Not run:
# Abort after 2 seconds
deadline <- Sys.time() + 2
llama_set_abort_callback(ctx, function() Sys.time() > deadline)
result <- llama_generate(ctx, "Tell me a long story", max_new_tokens = 500L)
llama_set_abort_callback(ctx, NULL)

## End(Not run)
```

llama_set_causal_attn *Set causal attention mode*

Description

When disabled, the model uses full (bidirectional) attention. This is useful for embedding models.

Usage

```
llama_set_causal_attn(ctx, causal)
```

Arguments

ctx	Context handle returned by [llama_new_context]
causal	Logical; TRUE for causal (autoregressive) attention, FALSE for full bidirectional attention

Value

No return value, called for side effects.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)
llama_set_causal_attn(ctx, FALSE)  # for embeddings

## End(Not run)
```

llama_set_threads *Set the number of threads for a context*

Description

Set the number of threads for a context

Usage

```
llama_set_threads(ctx, n_threads, n_threads_batch = n_threads)
```

Arguments

ctx	Context handle returned by [llama_new_context]
n_threads	Number of threads for single-token generation
n_threads_batch	Number of threads for batch processing (prompt encoding). Defaults to the same value as n_threads.

Value

No return value, called for side effects.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)
llama_set_threads(ctx, n_threads = 8L)

## End(Not run)
```

`llama_set_verbosity` *Set logging verbosity level*

Description

Controls how much diagnostic output is printed during model loading and inference.

Usage

```
llama_set_verbosity(level)
```

Arguments

`level`	Integer verbosity level: - 0: Silent (no output) - 1: Errors only (default) - 2: Normal (warnings and info) - 3: Verbose (all debug messages)

Value

No return value, called for side effects. Sets the global verbosity level used by the underlying 'llama.cpp' library.

Examples

```
# Suppress all output
llama_set_verbosity(0)

# Show only errors
llama_set_verbosity(1)

# Verbose output for debugging
llama_set_verbosity(3)
```

llama_set_warmup *Set warmup mode*

Description

When ‘warmup = TRUE‘, the context runs in warmup mode which pre-caches model weights in GPU memory without producing meaningful outputs. Call with ‘warmup = FALSE‘ to return to normal inference mode.

Usage

```
llama_set_warmup(ctx, warmup)
```

Arguments

ctx	Context handle returned by [llama_new_context]
warmup	Logical; ‘TRUE‘ to enable warmup mode, ‘FALSE‘ to disable.

Value

No return value, called for side effects.

llama_state_get_size *Get the size of the serialized context state in bytes*

Description

Returns the number of bytes required to serialize the current context state (KV cache + sampling state). Use before allocating a buffer for raw state I/O.

Usage

```
llama_state_get_size(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

A numeric scalar (size in bytes).

llama_state_load *Load context state from file*

Description

Restores a previously saved context state (including KV cache).

Usage

```
llama_state_load(ctx, path)
```

Arguments

ctx	Context handle returned by [llama_new_context]
path	File path to load state from

Value

A logical scalar: TRUE on success (errors on failure).

Examples

```
## Not run:
llama_state_load(ctx, "state.bin")
# Continue generation from saved state
result <- llama_generate(ctx, "")

## End(Not run)
```

llama_state_save *Save context state to file*

Description

Saves the full context state (including KV cache) to a binary file. This allows resuming generation later from the exact same state.

Usage

```
llama_state_save(ctx, path)
```

Arguments

ctx	Context handle returned by [llama_new_context]
path	File path to save state to

Value

A logical scalar: TRUE on success (errors on failure).

Examples

```
## Not run:
llama_state_save(ctx, "state.bin")

## End(Not run)
```

---

llama_supports_gpu *Check whether GPU offloading is available*

Description

Returns ‘TRUE‘ if at least one GPU backend (e.g. Vulkan) was detected at runtime. Use the result to decide whether to pass ‘n_gpu_layers != 0‘ to [llama_load_model].

Usage

```
llama_supports_gpu()
```

Value

A logical scalar: TRUE if at least one GPU backend (e.g. Vulkan) is available, FALSE otherwise.

Examples

```
if (llama_supports_gpu()) {
  message("GPU available, will use Vulkan backend")
} else {
  message("GPU not available, using CPU only")
}
```

---

llama_supports_mlock *Check whether memory locking is supported*

Description

Check whether memory locking is supported

Usage

```
llama_supports_mlock()
```

Value

A logical scalar: TRUE if mlock is supported.

Examples

```
# Check if memory locking is available (prevents swapping model to disk)
if (llama_supports_mlock()) {
  message("mlock available — model weights can be pinned in RAM")
}
```

---

llama_supports_mmap *Check whether memory-mapped file I/O is supported*

Description

Check whether memory-mapped file I/O is supported

Usage

```
llama_supports_mmap()
```

Value

A logical scalar: TRUE if mmap is supported.

Examples

```
# Check memory-mapping support before loading large models
if (llama_supports_mmap()) {
  message("mmap available — large models will load faster")
}
```

---

llama_supports_rpc *Check whether RPC backend is available*

Description

Check whether RPC backend is available

Usage

```
llama_supports_rpc()
```

Value

A logical scalar: ‘TRUE‘ if the RPC backend is compiled in.

llama_synchronize *Synchronize asynchronous computation*

Description

Blocks until all pending GPU/async operations for this context are complete. Normally not needed — ‘llama_decode‘ and ‘llama_generate‘ are synchronous — but useful when using low-level batch APIs in async mode.

Usage

```
llama_synchronize(ctx)
```

Arguments

ctx	Context handle returned by [llama_new_context]

Value

No return value, called for side effects.

llama_system_info *Get system information string*

Description

Returns a string with information about the system capabilities detected by llama.cpp (SIMD support, etc.).

Usage

```
llama_system_info()
```

Value

A character scalar with system capability information.

Examples

```
cat(llama_system_info(), "\n")
```

llama_time_us *Get current time in microseconds*

Description

Get current time in microseconds

Usage

```
llama_time_us()
```

Value

A numeric scalar with the current time in microseconds.

Examples

```
# Measure elapsed time for an operation
t0 <- llama_time_us()
Sys.sleep(0.01)
elapsed_ms <- (llama_time_us() - t0) / 1000
cat("Elapsed:", round(elapsed_ms, 1), "ms\n")
```

llama_tokenize *Tokenize text into token IDs*

Description

Tokenize text into token IDs

Usage

```
llama_tokenize(ctx, text, add_special = TRUE, parse_special = FALSE)
```

Arguments

ctx	Context handle returned by [llama_new_context]
text	Character string to tokenize
add_special	Whether to add special tokens (BOS/EOS) as configured by the model
parse_special	Whether to parse control/special tokens (e.g. Mistral's [INST], ChatML's <|im_start|>) as single tokens rather than as their literal characters. Use TRUE for a prompt produced by [llama_chat_apply_template]; the default FALSE treats such markup as plain text.

Value

An integer vector of token IDs as used by the model's vocabulary.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx <- llama_new_context(model)

tokens <- llama_tokenize(ctx, "Hello, world!")
print(tokens)
# [1] 1 15043 29892 3186 29991

# Without special tokens
tokens <- llama_tokenize(ctx, "Hello", add_special = FALSE)

# Parse a templated prompt's role markers as control tokens
prompt <- llama_chat_apply_template(list(list(role = "user", content = "hi")))
tokens <- llama_tokenize(ctx, prompt, parse_special = TRUE)

## End(Not run)
```

---

llama_token_to_piece *Convert a single token ID to its text piece*

Description

Convert a single token ID to its text piece

Usage

```
llama_token_to_piece(ctx, token, special = FALSE)
```

Arguments

ctx	A context pointer (llama_context).
token	Integer token ID.
special	Logical. If TRUE, render special tokens (e.g. <bos>).

Value

A character string — the text piece for the token.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
ctx   <- llama_new_context(model)

# Inspect individual tokens from tokenizer output
tokens <- llama_tokenize(ctx, "Hello world")
pieces <- vapply(tokens, function(t) llama_token_to_piece(ctx, t), "")
cat(paste(pieces, collapse = "|"), "\n")

## End(Not run)
```

---

llama_vocab_get_score *Get the score of a token*

Description

Returns the log-probability score stored in the vocabulary (used by SPM/UGM tokenizers).

Usage

```
llama_vocab_get_score(model, token)
```

Arguments

model	Model handle returned by [llama_load_model]
token	Integer token ID (0-based)

Value

A numeric scalar.

---

llama_vocab_get_text *Get the text representation of a token*

Description

Returns the raw text string stored in the vocabulary for a given token ID. Unlike [llama_token_to_piece], this does not apply any special rendering — it returns exactly what is stored in the GGUF vocabulary table.

Usage

```
llama_vocab_get_text(model, token)
```

Arguments

model	Model handle returned by [llama_load_model]
token	Integer token ID (0-based)

Value

A character string, or 'NULL' if the token has no text entry.

---

llama_vocab_info *Get vocabulary special token IDs*

Description

Returns the token IDs for special tokens (BOS, EOS, etc.) and fill-in-middle (FIM) tokens used by the model's vocabulary. A value of -1 indicates the token is not defined.

Usage

```
llama_vocab_info(model)
```

Arguments

model	Model handle returned by [llama_load_model]

Value

A named integer vector with token IDs for: `bos`, `eos`, `eot`, `sep`, `nl`, `pad`, `fim_pre`, `fim_suf`, `fim_mid`, `fim_rep`, `fim_sep`. A value of -1 means the token is not defined by the model.

Examples

```
## Not run:
model <- llama_load_model("model.gguf")
vocab <- llama_vocab_info(model)
cat("BOS token:", vocab["bos"], "\n")
cat("EOS token:", vocab["eos"], "\n")

## End(Not run)
```

llama_vocab_is_control *Check if a token is a control token*

Description

Check if a token is a control token

Usage

```
llama_vocab_is_control(model, token)
```

Arguments

model	Model handle returned by [llama_load_model]
token	Integer token ID (0-based)

Value

A logical scalar.

llama_vocab_is_eog *Check if a token is an end-of-generation token*

Description

Returns ‘TRUE‘ for EOS, EOT, and other tokens that signal end of output. Useful for implementing custom generation loops.

Usage

```
llama_vocab_is_eog(model, token)
```

Arguments

model	Model handle returned by [llama_load_model]
token	Integer token ID (0-based)

Value

A logical scalar.

`llama_vocab_type` *Get vocabulary type*

Description

Get vocabulary type

Usage

```
llama_vocab_type(model)
```

Arguments

`model`	Model handle returned by [llama_load_model]

Value

A character string: one of ‘"spm"‘ (LLaMA/SentencePiece BPE), ‘"bpe"‘ (GPT-2 BPE), ‘"wpm"‘ (BERT WordPiece), ‘"ugm"‘ (T5 Unigram), ‘"rwkv"‘, ‘"plamo2"‘, or ‘"none"‘.

Index