```
        self.gguf_writer.add_rope_dimension_count(rope_dim)
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.NONE)
        self.gguf_writer.add_leading_dense_block_count(hparams["first_k_dense_replace"])
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])
        self.gguf_writer.add_expert_feed_forward_length(hparams["moe_intermediate_size"])
        self.gguf_writer.add_expert_weights_scale(1.0)
        self.gguf_writer.add_expert_count(hparams["num_experts"])
        self.gguf_writer.add_expert_shared_count(hparams["num_shared_experts"])
        self.gguf_writer.add_expert_weights_norm(hparams["norm_topk_prob"])

    _experts: list[dict[str, Tensor]] | None = None

    @staticmethod
    def permute(weights: Tensor, n_head: int, n_head_kv: int | None):
        if n_head_kv is not None and n_head != n_head_kv:
            n_head = n_head_kv
        return (weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
                .swapaxes(1, 2)
                .reshape(weights.shape))

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        n_head = self.hparams["num_attention_heads"]
        n_kv_head = self.hparams.get("num_key_value_heads")
        n_embd = self.hparams["hidden_size"]
        head_dim = self.hparams.get("head_dim") or n_embd // n_head

        output_name = self.format_tensor_name(gguf.MODEL_TENSOR.OUTPUT)

        if name.endswith("attention.dense.weight"):
            return [(self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_OUT, bid), data_torch)]
        elif name.endswith("query_key_value.weight"):
            q, k, v = data_torch.split([n_head * head_dim, n_kv_head * head_dim, n_kv_head * head_dim], dim=-2)

            return [
                    (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_Q, bid), BailingMoeModel.permute(q, n_head,
n_head)),
                    (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_K, bid), BailingMoeModel.permute(k, n_head,
n_kv_head)),
                (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_V, bid), v)
            ]
        elif name.find("mlp.experts") != -1:
            n_experts = self.hparams["num_experts"]
            assert bid is not None

            tensors: list[tuple[str, Tensor]] = []

            if self._experts is None:
                self._experts = [{} for _ in range(self.block_count)]

            self._experts[bid][name] = data_torch

            if len(self._experts[bid]) >= n_experts * 3:
                # merge the experts into a single 3d tensor
                for w_name in ["down_proj", "gate_proj", "up_proj"]:
                    datas: list[Tensor] = []

                    for xid in range(n_experts):
                        ename = f"model.layers.{bid}.mlp.experts.{xid}.{w_name}.weight"
                        datas.append(self._experts[bid][ename])
                        del self._experts[bid][ename]

                    data_torch = torch.stack(datas, dim=0)

                    merged_name = f"model.layers.{bid}.mlp.experts.{w_name}.weight"

                    new_name = self.map_tensor_name(merged_name)

                    tensors.append((new_name, data_torch))

            return tensors

        new_name = self.map_tensor_name(name)

        if new_name == output_name and self.hparams.get("norm_head"):
            data_torch = data_torch.float()
            data_torch /= torch.norm(data_torch, p=2, dim=0, keepdim=True) + 1e-7

        return [(new_name, data_torch)]

    def prepare_tensors(self):
        super().prepare_tensors()

        if self._experts is not None:
            # flatten `list[dict[str, Tensor]]` into `list[str]`
            experts = [k for d in self._experts for k in d.keys()]
            if len(experts) > 0:
                raise ValueError(f"Unprocessed experts: {experts}")


@Model.register("ChameleonForConditionalGeneration")
@Model.register("ChameleonForCausalLM")  # obsolete
class ChameleonModel(Model):
    model_arch = gguf.MODEL_ARCH.CHAMELEON

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_swin_norm(self.hparams.get("swin_norm", False))

    def set_vocab(self):
        self._set_vocab_gpt2()

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # ignore image tokenizer for now
        # TODO: remove this once image support is implemented for Chameleon
        if name.startswith("model.vqmodel"):
            return []
        n_head = self.hparams["num_attention_heads"]
        n_kv_head = self.hparams.get("num_key_value_heads")
        hidden_dim = self.hparams.get("hidden_size")

        if name.endswith(("q_proj.weight", "q_proj.bias")):
            data_torch = LlamaModel.permute(data_torch, n_head, n_head)
        if name.endswith(("k_proj.weight", "k_proj.bias")):
            data_torch = LlamaModel.permute(data_torch, n_head, n_kv_head)
        if name.endswith(("q_norm.weight", "q_norm.bias")):
            data_torch = ChameleonModel._reverse_hf_permute(data_torch, n_head, hidden_dim)
        if name.endswith(("k_norm.weight", "k_norm.bias")):
            data_torch = ChameleonModel._reverse_hf_permute(data_torch, n_kv_head, hidden_dim)

        return [(self.map_tensor_name(name), data_torch)]

                                                                                     #                      see:
https://github.com/huggingface/transformers/blob/72fb02c47dbbe1999ae105319f24631cad6e2e00/src/transformers/mode
ls/chameleon/convert_chameleon_weights_to_hf.py#L176-L203
    @staticmethod
    def _reverse_hf_permute(data_torch, n_heads, hidden_dim):
        head_dim = hidden_dim // n_heads
        data_torch = data_torch[0].view(2, head_dim // 2).t().reshape(1, -1)
        data_torch = data_torch.repeat_interleave(n_heads, 0)
        return data_torch


###### CONVERSION LOGIC ######


# tree of lazy tensors
class LazyTorchTensor(gguf.LazyBase):
    _tensor_type = torch.Tensor
    # to keep the type-checker happy
    dtype: torch.dtype
    shape: torch.Size

    # only used when converting a torch.Tensor to a np.ndarray
    _dtype_map: dict[torch.dtype, type] = {
        torch.float16: np.float16,
        torch.float32: np.float32,
    }

    # used for safetensors slices
                                                                                     #                      ref:
https://github.com/huggingface/safetensors/blob/079781fd0dc455ba0fe851e2b4507c33d0c0d407/bindings/python/src/li
b.rs#L1046
    # TODO: uncomment U64, U32, and U16, ref: https://github.com/pytorch/pytorch/issues/58734
    _dtype_str_map: dict[str, torch.dtype] = {
        "F64": torch.float64,
        "F32": torch.float32,
        "BF16": torch.bfloat16,
        "F16": torch.float16,
        # "U64": torch.uint64,
        "I64": torch.int64,
        # "U32": torch.uint32,
        "I32": torch.int32,
        # "U16": torch.uint16,
        "I16": torch.int16,
        "U8": torch.uint8,
        "I8": torch.int8,
        "BOOL": torch.bool,
        "F8_E4M3": torch.float8_e4m3fn,
        "F8_E5M2": torch.float8_e5m2,
    }

    def numpy(self) -> gguf.LazyNumpyTensor:
        dtype = self._dtype_map[self.dtype]
        return gguf.LazyNumpyTensor(
            meta=gguf.LazyNumpyTensor.meta_with_dtype_and_shape(dtype, self.shape),
            args=(self,),
            func=(lambda s: s.numpy())
        )

    @classmethod
    def meta_with_dtype_and_shape(cls, dtype: torch.dtype, shape: tuple[int, ...]) -> Tensor:
        return torch.empty(size=shape, dtype=dtype, device="meta")

    @classmethod
    def from_safetensors_slice(cls, st_slice: Any) -> Tensor:
        dtype = cls._dtype_str_map[st_slice.get_dtype()]
        shape: tuple[int, ...] = tuple(st_slice.get_shape())
        lazy = cls(meta=cls.meta_with_dtype_and_shape(dtype, shape), args=(st_slice,), func=lambda s: s[:])
        return cast(torch.Tensor, lazy)

    @classmethod
    def from_remote_tensor(cls, remote_tensor: gguf.utility.RemoteTensor):
        dtype = cls._dtype_str_map[remote_tensor.dtype]
        shape = remote_tensor.shape
        meta = cls.meta_with_dtype_and_shape(dtype, shape)
        lazy = cls(meta=meta, args=(remote_tensor,), func=lambda r: torch.frombuffer(r.data(), dtype=dtype).reshape(shape))
        return cast(torch.Tensor, lazy)

    @classmethod
    def __torch_function__(cls, func, types, args=(), kwargs=None):
        del types  # unused

        if kwargs is None:
            kwargs = {}

        if func is torch.Tensor.numpy:
            return args[0].numpy()

        return cls._wrap_fn(func)(*args, **kwargs)


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Convert a huggingface model to a GGML compatible file")
    parser.add_argument(
        "--vocab-only", action="store_true",
        help="extract only the vocab",
    )
    parser.add_argument(
        "--outfile", type=Path,
        help="path to write to; default: based on input. {ftype} will be replaced by the outtype.",
    )
    parser.add_argument(
        "--outtype", type=str, choices=["f32", "f16", "bf16", "q8_0", "tq1_0", "tq2_0", "auto"], default="f16",
        help="output format - use f32 for float32, f16 for float16, bf16 for bfloat16, q8_0 for Q8_0, tq1_0 or tq2_0 for ternary, and auto for the highest-fidelity 16-bit float type depending on the first loaded tensor type",
    )
    parser.add_argument(
        "--bigendian", action="store_true",
        help="model is executed on big endian machine",
    )
    parser.add_argument(
        "model", type=Path,
        help="directory containing model file",
        nargs="?",
    )
    parser.add_argument(
        "--use-temp-file", action="store_true",
        help="use the tempfile library while processing (helpful when running out of memory, process killed)",
    )
    parser.add_argument(
        "--no-lazy", action="store_true",
        help="use more RAM by computing all outputs before writing (use in case lazy evaluation is broken)",
    )
    parser.add_argument(
        "--model-name", type=str, default=None,
        help="name of the model",
    )
    parser.add_argument(
        "--verbose", action="store_true",
        help="increase output verbosity",
    )
    parser.add_argument(
        "--split-max-tensors", type=int, default=0,
        help="max tensors in each split",
    )
    parser.add_argument(
        "--split-max-size", type=str, default="0",
        help="max size per split N(M|G)",
    )
    parser.add_argument(
        "--dry-run", action="store_true",
        help="only print out a split plan and exit, without writing any new files",
    )
    parser.add_argument(
        "--no-tensor-first-split", action="store_true",
        help="do not add tensors to the first split (disabled by default)"
    )
    parser.add_argument(
        "--metadata", type=Path,
        help="Specify the path for an authorship metadata override file"
    )
    parser.add_argument(
        "--print-supported-models", action="store_true",
        help="Print the supported models"
    )
    parser.add_argument(
        "--remote", action="store_true",
        help="(Experimental) Read safetensors file remotely without downloading to disk. Config and tokenizer files will still be downloaded. To use this feature, you need to specify Hugging Face model repo name instead of a local directory. For example: 'HuggingFaceTB/SmolLM2-1.7B-Instruct'. Note: To access gated repo, set HF_TOKEN environment variable to your Hugging Face token.",
    )

    args = parser.parse_args()
    if not args.print_supported_models and args.model is None:
        parser.error("the following arguments are required: model")
    return args


def split_str_to_n_bytes(split_str: str) -> int:
    if split_str.endswith("K"):
        n = int(split_str[:-1]) * 1000
    elif split_str.endswith("M"):
        n = int(split_str[:-1]) * 1000 * 1000
    elif split_str.endswith("G"):
        n = int(split_str[:-1]) * 1000 * 1000 * 1000
    elif split_str.isnumeric():
        n = int(split_str)
    else:
        raise ValueError(f"Invalid split size: {split_str}, must be a number, optionally followed by K, M, or G")

    if n < 0:
        raise ValueError(f"Invalid split size: {split_str}, must be positive")

    return n


def main() -> None:
    args = parse_args()

    if args.print_supported_models:
        logger.error("Supported models:")
        Model.print_registered_models()
        sys.exit(0)

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG)
    else:
```

```
    logging.basicConfig(level=logging.INFO)

dir_model = args.model

if args.remote:
    from huggingface_hub import snapshot_download
    local_dir = snapshot_download(
        repo_id=str(dir_model),
        allow_patterns=["LICENSE", "*.json", "*.md", "*.txt", "tokenizer.model"])
    dir_model = Path(local_dir)
    logger.info(f"Downloaded config and tokenizer to {local_dir}")

if not dir_model.is_dir():
    logger.error(f'Error: {args.model} is not a directory')
    sys.exit(1)

ftype_map: dict[str, gguf.LlamaFileType] = {
    "f32": gguf.LlamaFileType.ALL_F32,
    "f16": gguf.LlamaFileType.MOSTLY_F16,
    "bf16": gguf.LlamaFileType.MOSTLY_BF16,
    "q8_0": gguf.LlamaFileType.MOSTLY_Q8_0,
    "tq1_0": gguf.LlamaFileType.MOSTLY_TQ1_0,
    "tq2_0": gguf.LlamaFileType.MOSTLY_TQ2_0,
    "auto": gguf.LlamaFileType.GUESSED,
}

is_split = args.split_max_tensors > 0 or args.split_max_size != "0"
if args.use_temp_file and is_split:
    logger.error("Error: Cannot use temp file when splitting")
    sys.exit(1)

if args.outfile is not None:
    fname_out = args.outfile
elif args.remote:
    # if remote, use the model ID as the output file name
    fname_out = Path("./" + str(args.model).replace("/", "-") + "-{ftype}.gguf")
else:
    fname_out = dir_model

logger.info(f"Loading model: {dir_model.name}")

hparams = Model.load_hparams(dir_model)

with torch.inference_mode():
    output_type = ftype_map[args.outtype]
    model_architecture = hparams["architectures"][0]
    try:
        model_class = Model.from_model_architecture(model_architecture)
    except NotImplementedError:
        logger.error(f"Model {model_architecture} is not supported")
        sys.exit(1)

    model_instance = model_class(dir_model, output_type, fname_out,
                                 is_big_endian=args.bigendian, use_temp_file=args.use_temp_file,
                                     eager=args.no_lazy,
                                     metadata_override=args.metadata, model_name=args.model_name,
                                     split_max_tensors=args.split_max_tensors,
                                                      split_max_size=split_str_to_n_bytes(args.split_max_size),
dry_run=args.dry_run,
                                     small_first_shard=args.no_tensor_first_split,
                                     remote_hf_model_id=str(args.model) if args.remote else None)

        if args.vocab_only:
            logger.info("Exporting model vocab...")
            model_instance.write_vocab()
            logger.info(f"Model vocab successfully exported to {model_instance.fname_out}")
        else:
            logger.info("Exporting model...")
            model_instance.write()
            out_path = f"{model_instance.fname_out.parent}{os.sep}" if is_split else model_instance.fname_out
            logger.info(f"Model successfully exported to {out_path}")


if __name__ == '__main__':
    main()

==== convert_hf_to_gguf_update.py ====
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# This script downloads the tokenizer models of the specified models from Huggingface and
# generates the get_vocab_base_pre() function for convert_hf_to_gguf.py
#
# This is necessary in order to analyze the type of pre-tokenizer used by the model and
# provide the necessary information to llama.cpp via the GGUF header in order to implement
# the same pre-tokenizer.
#
# ref: https://github.com/ggml-org/llama.cpp/pull/6920
#
# Instructions:
#
# - Add a new model to the "models" list
# - Run the script with your huggingface token:
#
#   python3 convert_hf_to_gguf_update.py <huggingface_token>
#
# - The convert_hf_to_gguf.py script will have had its get_vocab_base_pre() function updated
# - Update llama.cpp with the new pre-tokenizer if necessary
#
# TODO: generate tokenizer tests for llama.cpp
#

import logging
import os
import pathlib
import re

import requests
```

```python
import sys
import json
import shutil

from hashlib import sha256
from enum import IntEnum, auto
from transformers import AutoTokenizer

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger("convert_hf_to_gguf_update")
sess = requests.Session()


class TOKENIZER_TYPE(IntEnum):
    SPM = auto()
    BPE = auto()
    WPM = auto()
    UGM = auto()


# TODO: this string has to exercise as much pre-tokenizer functionality as possible
#       will be updated with time - contributions welcome
CHK_TXT = '\n \n\n \n\n\n \t \t\t \t\n  \n   \n    \n     \nLAUNCH (normal) ???? (multiple emojis concatenated)
[OK] ?? 3 33 333 3333 33333 333333 3333333 33333333 3.3 3..3 3...3 ??????????????? ????apple??1314151??
------======= ???? ?? ????????? \'\'\'\'\'\'\'``````````\"\"\"\"......!!!!!!?????? I\'ve been \'told he\'s there,
\'RE you sure? \'M not sure I\'ll make it, \'D you like some tea? We\'Ve a\'lL'

if len(sys.argv) == 2:
    token = sys.argv[1]
    if not token.startswith("hf_"):
        logger.info("Huggingface token seems invalid")
        logger.info("Usage: python convert_hf_to_gguf_update.py <huggingface_token>")
        sys.exit(1)
else:
    logger.info("Usage: python convert_hf_to_gguf_update.py <huggingface_token>")
    sys.exit(1)

# TODO: add models here, base models preferred
models = [
    {"name": "llama-spm", "tokt": TOKENIZER_TYPE.SPM, "repo":
"https://huggingface.co/meta-llama/Llama-2-7b-hf", },
    {"name": "llama-bpe", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/meta-llama/Meta-Llama-3-8B", },
    {"name": "phi-3", "tokt": TOKENIZER_TYPE.SPM, "repo":
"https://huggingface.co/microsoft/Phi-3-mini-4k-instruct", },
    {"name": "deepseek-llm", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/deepseek-ai/deepseek-llm-7b-base", },
    {"name": "deepseek-coder", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base", },
    {"name": "falcon",         "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/tiiuae/falcon-7b",
},
    {"name": "bert-bge", "tokt": TOKENIZER_TYPE.WPM, "repo":
"https://huggingface.co/BAAI/bge-small-en-v1.5", },
    {"name": "falcon3", "tokt": TOKENIZER_TYPE.BPE, "repo":
```

```
"https://huggingface.co/tiiuae/Falcon3-7B-Base", },
    {"name": "bert-bge-large", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/BAAI/bge-large-zh-v1.5", },
    {"name": "mpt", "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/mosaicml/mpt-7b",
},
    {"name": "starcoder", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/bigcode/starcoder2-3b", },
    {"name": "gpt-2", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/openai-community/gpt2", },
    {"name": "stablelm2", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/stabilityai/stablelm-2-zephyr-1_6b", },
    {"name": "refact", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/smallcloudai/Refact-1_6-base", },
    {"name": "command-r", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/CohereForAI/c4ai-command-r-v01", },
    {"name": "qwen2", "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/Qwen/Qwen1.5-7B",
},
    {"name": "olmo", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/allenai/OLMo-1.7-7B-hf", },
    {"name": "dbrx", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/databricks/dbrx-base", },
    {"name": "jina-v1-en", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/jinaai/jina-reranker-v1-tiny-en", },
    {"name": "jina-v2-en", "tokt": TOKENIZER_TYPE.WPM, "repo":
"https://huggingface.co/jinaai/jina-embeddings-v2-base-en", }, # WPM!
    {"name": "jina-v2-es", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/jinaai/jina-embeddings-v2-base-es", },
    {"name": "jina-v2-de", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/jinaai/jina-embeddings-v2-base-de", },
    {"name": "smaug-bpe", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/abacusai/Smaug-Llama-3-70B-Instruct", },
    {"name": "poro-chat", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/LumiOpen/Poro-34B-chat", },
    {"name": "jina-v2-code", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/jinaai/jina-embeddings-v2-base-code", },
    {"name": "viking", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/LumiOpen/Viking-7B", }, # Also used for Viking 13B and 33B
    {"name": "gemma", "tokt": TOKENIZER_TYPE.SPM, "repo": "https://huggingface.co/google/gemma-2b",
},
    {"name": "gemma-2", "tokt": TOKENIZER_TYPE.SPM, "repo":
"https://huggingface.co/google/gemma-2-9b", },
    {"name": "jais", "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/core42/jais-13b",
},
    {"name": "t5", "tokt": TOKENIZER_TYPE.UGM, "repo":
"https://huggingface.co/google-t5/t5-small", },
    {"name": "codeshell", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/WisdomShell/CodeShell-7B", },
    {"name": "tekken", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/mistralai/Mistral-Nemo-Base-2407", },
    {"name": "smollm", "tokt": TOKENIZER_TYPE.BPE, "repo":
"https://huggingface.co/HuggingFaceTB/SmolLM-135M", },
    {'name': "bloom", "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/bigscience/bloom",
},
    {'name': "gpt3-finnish", "tokt": TOKENIZER_TYPE.BPE, "repo":
```

```
"https://huggingface.co/TurkuNLP/gpt3-finnish-small", },
              {"name":   "exaone",                                     "tokt":   TOKENIZER_TYPE.BPE,   "repo":
"https://huggingface.co/LGAI-EXAONE/EXAONE-3.0-7.8B-Instruct", },
    {"name": "phi-2",            "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/microsoft/phi-2",
},
                {"name":    "chameleon",                              "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/facebook/chameleon-7b", },
                 {"name":    "minerva-7b",                            "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/sapienzanlp/Minerva-7B-base-v1.0", },
                  {"name":    "roberta-bpe",                          "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/sentence-transformers/stsb-roberta-base"},
               {"name":    "gigachat",                                "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/ai-sage/GigaChat-20B-A3B-instruct"},
              {"name":   "megrez",                                     "tokt":   TOKENIZER_TYPE.BPE,   "repo":
"https://huggingface.co/Infinigence/Megrez-3B-Instruct"},
                  {"name":    "deepseek-v3",                          "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/deepseek-ai/DeepSeek-V3"},
                          {"name":      "deepseek-r1-qwen",      "tokt":      TOKENIZER_TYPE.BPE,      "repo":
"https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B"},
    {"name": "gpt-4o",           "tokt": TOKENIZER_TYPE.BPE, "repo": "https://huggingface.co/Xenova/gpt-4o", },
               {"name":    "superbpe",                                "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/UW/OLMo2-8B-SuperBPE-t180k", },
               {"name":    "trillion",                                "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/trillionlabs/Trillion-7B-preview", },
                 {"name":    "bailingmoe",                            "tokt":    TOKENIZER_TYPE.BPE,    "repo":
"https://huggingface.co/inclusionAI/Ling-lite", },
              {"name":   "llama4",                                     "tokt":   TOKENIZER_TYPE.BPE,   "repo":
"https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct", },
             {"name":   "glm4",                                         "tokt":   TOKENIZER_TYPE.BPE,   "repo":
"https://huggingface.co/THUDM/glm-4-9b-hf", },
]


def download_file_with_auth(url, token, save_path):
    headers = {"Authorization": f"Bearer {token}"}
    response = sess.get(url, headers=headers)
    response.raise_for_status()
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    with open(save_path, 'wb') as downloaded_file:
        downloaded_file.write(response.content)
    logger.info(f"File {save_path} downloaded successfully")


def download_model(model):
    name = model["name"]
    repo = model["repo"]
    tokt = model["tokt"]

    os.makedirs(f"models/tokenizers/{name}", exist_ok=True)

    files = ["config.json", "tokenizer.json", "tokenizer_config.json"]

    if name == "gpt-4o":
        # Xenova/gpt-4o is tokenizer-only, it does not contain config.json
        files = ["tokenizer.json", "tokenizer_config.json"]

    if tokt == TOKENIZER_TYPE.SPM:
        files.append("tokenizer.model")

    if tokt == TOKENIZER_TYPE.UGM:
        files.append("spiece.model")

    if os.path.isdir(repo):
        # If repo is a path on the file system, copy the directory
        for file in files:
            src_path = os.path.join(repo, file)
            dst_path = f"models/tokenizers/{name}/{file}"
            if os.path.isfile(dst_path):
                logger.info(f"{name}: File {dst_path} already exists - skipping")
                continue
            if os.path.isfile(src_path):
                shutil.copy2(src_path, dst_path)
                logger.info(f"{name}: Copied {src_path} to {dst_path}")
            else:
                logger.warning(f"{name}: Source file {src_path} does not exist")
    else:
        # If repo is a URL, download the files
        for file in files:
            save_path = f"models/tokenizers/{name}/{file}"
            if os.path.isfile(save_path):
                logger.info(f"{name}: File {save_path} already exists - skipping")
                continue
            download_file_with_auth(f"{repo}/resolve/main/{file}", token, save_path)


for model in models:
    try:
        download_model(model)
    except Exception as e:
        logger.error(f"Failed to download model {model['name']}. Error: {e}")


# generate the source code for the convert_hf_to_gguf.py:get_vocab_base_pre() function:

src_ifs = ""
for model in models:
    name = model["name"]
    tokt = model["tokt"]

    if tokt == TOKENIZER_TYPE.SPM or tokt == TOKENIZER_TYPE.UGM:
        continue

    # Skip if the tokenizer folder does not exist or there are other download issues previously
    if not os.path.exists(f"models/tokenizers/{name}"):
        logger.warning(f"Directory for tokenizer {name} not found. Skipping...")
        continue

    # create the tokenizer
    try:
        if name == "t5":
            tokenizer = AutoTokenizer.from_pretrained(f"models/tokenizers/{name}", use_fast=False)
        else:
            tokenizer = AutoTokenizer.from_pretrained(f"models/tokenizers/{name}")
    except OSError as e:
        logger.error(f"Error loading tokenizer for model {name}. The model may not exist or is not accessible with the provided token. Error: {e}")
        continue  # Skip to the next model if the tokenizer can't be loaded

    chktok = tokenizer.encode(CHK_TXT)
    chkhsh = sha256(str(chktok).encode()).hexdigest()

    logger.info(f"model: {name}")
    logger.info(f"tokt: {tokt}")
    logger.info(f"repo: {model['repo']}")
    logger.info(f"chktok: {chktok}")
    logger.info(f"chkhsh: {chkhsh}")

    # print the "pre_tokenizer" content from the tokenizer.json
    with open(f"models/tokenizers/{name}/tokenizer.json", "r", encoding="utf-8") as f:
        cfg = json.load(f)
        normalizer = cfg["normalizer"]
        logger.info("normalizer: " + json.dumps(normalizer, indent=4))
        pre_tokenizer = cfg["pre_tokenizer"]
        logger.info("pre_tokenizer: " + json.dumps(pre_tokenizer, indent=4))
        if "ignore_merges" in cfg["model"]:
            logger.info("ignore_merges: " + json.dumps(cfg["model"]["ignore_merges"], indent=4))

    logger.info("")

    src_ifs += f"        if chkhsh == \"{chkhsh}\":\n"
    src_ifs += f"            # ref: {model['repo']}\n"
    src_ifs += f"            res = \"{name}\"\n"

src_func = f"""
    def get_vocab_base_pre(self, tokenizer) -> str:
        # encoding this string and hashing the resulting tokens would (hopefully) give us a unique identifier that
        # is specific for the BPE pre-tokenizer used by the model
        # we will use this unique identifier to write a "tokenizer.ggml.pre" entry in the GGUF file which we can
        # use in llama.cpp to implement the same pre-tokenizer

        chktxt = {repr(CHK_TXT)}

        chktok = tokenizer.encode(chktxt)
        chkhsh = sha256(str(chktok).encode()).hexdigest()

        logger.debug(f"chktok: {{chktok}}")
        logger.debug(f"chkhsh: {{chkhsh}}")

        res = None
        # NOTE: if you get an error here, you need to update the convert_hf_to_gguf_update.py script
        #       or pull the latest version of the model from Huggingface
        #       don't edit the hashes manually!
{src_ifs}
        if res is None:
            logger.warning("\\n")

logger.warning("**************************************************************************************")
            logger.warning("** WARNING: The BPE pre-tokenizer was not recognized!")
            logger.warning("**          There are 2 possible reasons for this:")
            logger.warning("**          - the model has not been added to convert_hf_to_gguf_update.py yet")
            logger.warning("**          - the pre-tokenization config has changed upstream")
            logger.warning("**          Check your model files and convert_hf_to_gguf_update.py and update them
accordingly.")
            logger.warning("** ref:     https://github.com/ggml-org/llama.cpp/pull/6920")
            logger.warning("**")
            logger.warning(f"** chkhsh:  {{chkhsh}}")

logger.warning("**************************************************************************************")
            logger.warning("\\n")
            raise NotImplementedError("BPE pre-tokenizer was not recognized - update get_vocab_base_pre()")

        logger.debug(f"tokenizer.ggml.pre: {{repr(res)}}")
        logger.debug(f"chkhsh: {{chkhsh}}")

        return res
"""

convert_py_pth = pathlib.Path("convert_hf_to_gguf.py")
convert_py = convert_py_pth.read_text(encoding="utf-8")
convert_py = re.sub(
    r"(# Marker: Start get_vocab_base_pre)(.+?)( +# Marker: End get_vocab_base_pre)",
    lambda m: m.group(1) + src_func + m.group(3),
    convert_py,
    flags=re.DOTALL | re.MULTILINE,
)

convert_py_pth.write_text(convert_py, encoding="utf-8")

logger.info("+++ convert_hf_to_gguf.py was updated")

# generate tests for each tokenizer model

tests = [
    "ied 4 ? months",
    "F?hrer",
    "",
    " ",
    "  ",
    "   ",
    "\t",
    "\n",
    "\n\n",
    "\n\n\n",
    "\t\n",
    "Hello world",
    " Hello world",
    "Hello World",
    " Hello World",
    " Hello World!",
    "Hello, world!",
    " Hello, world!",
    " this is ?.cpp",
    "w048 7tuijk dsdfhu",
    "???? ?? ?????????",
    "???????????????????",
    "LAUNCH (normal) ???? (multiple emojis concatenated) [OK] (only emoji that has its own token)",
    "Hello",
    " Hello",
    "  Hello",
    "   Hello",
    "    Hello",
    "    Hello\n    Hello",
    " (",
    "\n =",
    "' era",
    "Hello, y'all! How are you ? ????apple??1314151??",
    "!!!!!!",
    "3",
    "33",
    "333",
    "3333",
    "33333",
    "333333",
    "3333333",
    "33333333",
    "333333333",
    "C?a Vi?t", # llama-bpe fails on this
    " discards",
    CHK_TXT,
]

# write the tests to ./models/ggml-vocab-{name}.gguf.inp
# the format is:
#
# test0
# __ggml_vocab_test__
# test1
# __ggml_vocab_test__
# ...
#

# with each model, encode all tests and write the results in ./models/ggml-vocab-{name}.gguf.out
# for each test, write the resulting tokens on a separate line

for model in models:
    name = model["name"]
    tokt = model["tokt"]
    # Skip if the tokenizer folder does not exist or there are other download issues previously
    if not os.path.exists(f"models/tokenizers/{name}"):
        logger.warning(f"Directory for tokenizer {name} not found. Skipping...")
        continue

    # create the tokenizer
    try:
        if name == "t5":
            tokenizer = AutoTokenizer.from_pretrained(f"models/tokenizers/{name}", use_fast=False)
        else:
            tokenizer = AutoTokenizer.from_pretrained(f"models/tokenizers/{name}")
    except OSError as e:
        logger.error(f"Failed to load tokenizer for model {name}. Error: {e}")
        continue  # Skip this model and continue with the next one in the loop

    with open(f"models/ggml-vocab-{name}.gguf.inp", "w", encoding="utf-8") as f:
        for text in tests:
            f.write(f"{text}")
            f.write("\n__ggml_vocab_test__\n")

    with open(f"models/ggml-vocab-{name}.gguf.out", "w") as f:
        for text in tests:
            res = tokenizer.encode(text, add_special_tokens=False)
            for r in res:
                f.write(f" {r}")
            f.write("\n")

    logger.info(f"Tests for {name} written in ./models/ggml-vocab-{name}.gguf.*")

# generate commands for creating vocab files

logger.info("\nRun the following commands to generate the vocab files for testing:\n")

for model in models:
    name = model["name"]

    print(f"python3 convert_hf_to_gguf.py models/tokenizers/{name}/ --outfile models/ggml-vocab-{name}.gguf --vocab-only") # noqa: NP100

logger.info("\n")

==== convert_image_encoder_to_gguf.py ====
import argparse
import os
import json
import re

import torch
import numpy as np
from gguf import *
from transformers import CLIPModel, CLIPProcessor, CLIPVisionModel, SiglipVisionModel

TEXT = "clip.text"
```

```
VISION = "clip.vision"


def k(raw_key: str, arch: str) -> str:
    return raw_key.format(arch=arch)


def should_skip_tensor(name: str, has_text: bool, has_vision: bool, has_llava: bool) -> bool:
    if name in (
        "logit_scale",
        "text_model.embeddings.position_ids",
        "vision_model.embeddings.position_ids",
    ):
        return True

    if has_llava and name in ["visual_projection.weight", "vision_model.post_layernorm.weight",
"vision_model.post_layernorm.bias"]:
        return True

    if name.startswith("v") and not has_vision:
        return True

    if name.startswith("t") and not has_text:
        return True

    return False


def get_tensor_name(name: str) -> str:
    # Standardize the transformers llava next keys for
    # image newline / mm projector with the classes in haotian-liu LLaVA
    if name == "image_newline":
        return "model.image_newline"
    if name.startswith("multi_modal_projector"):
        name = name.replace("multi_modal_projector", "mm")
        if "linear_1" in name:
            name = name.replace("linear_1", "0")
        if "linear_2" in name:
            name = name.replace("linear_2", "2")
        return name

    if "projection" in name:
        return name
    if "mm_projector" in name:
        name = name.replace("model.mm_projector", "mm")
        name = re.sub(r'mm\.mlp\.mlp', 'mm.model.mlp', name, count=1)
        name = re.sub(r'mm\.peg\.peg', 'mm.model.peg', name, count=1)
        return name

    return name.replace("text_model", "t").replace("vision_model", "v").replace("encoder.layers",
"blk").replace("embeddings.", "").replace("_proj", "").replace("self_attn.", "attn_").replace("layer_norm",
"ln").replace("layernorm", "ln").replace("mlp.fc1", "ffn_down").replace("mlp.fc2",
"ffn_up").replace("embedding", "embd").replace("final", "post").replace("layrnorm", "ln")
```

```
def bytes_to_unicode():
    """
    Returns list of utf-8 byte and a corresponding list of unicode strings.
    The reversible bpe codes work on unicode strings.
    This means you need a large # of unicode characters in your vocab if you want to avoid UNKs.
    When you're at something like a 10B token dataset you end up needing around 5K for decent coverage.
    This is a significant percentage of your normal, say, 32K bpe vocab.
    To avoid that, we want lookup tables between utf-8 bytes and unicode strings.
    And avoids mapping to whitespace/control characters the bpe code barfs on.
    """
    bs = (
        list(range(ord("!"), ord("~") + 1))
        + list(range(ord("?"), ord("?") + 1))
        + list(range(ord("?"), ord("?") + 1))
    )
    cs = bs[:]
    n = 0
    for b in range(2**8):
        if b not in bs:
            bs.append(b)
            cs.append(2**8 + n)
            n += 1
    cs = [chr(n) for n in cs]
    return dict(zip(bs, cs))


ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model-dir", help="Path to model directory cloned from HF Hub", required=True)
ap.add_argument("--use-f32", action="store_true", default=False, help="Use f32 instead of f16")
ap.add_argument('--bigendian', action="store_true", default=False, help="Model is executed on big-endian machine")
ap.add_argument("--text-only", action="store_true", required=False,
                help="Save a text-only model. It can't be used to encode images")
ap.add_argument("--vision-only", action="store_true", required=False,
                help="Save a vision-only model. It can't be used to encode texts")
ap.add_argument("--clip-model-is-vision", action="store_true", required=False,
                help="The clip model is a pure vision model (ShareGPT4V vision extract for example)")

# Selectable visual encoders that are compatible with this script
encoder_group = ap.add_mutually_exclusive_group()
encoder_group.add_argument("--clip-model-is-openclip", action="store_true", required=False,
                help="The clip model is from openclip (for ViT-SO400M type))")
encoder_group.add_argument("--clip-model-is-siglip", action="store_true", required=False,
                help="the visual encoder is Siglip.")

ap.add_argument("--llava-projector", help="Path to llava.projector file. If specified, save an image encoder for LLaVA models.")
ap.add_argument("--projector-type", help="Type of projector. Possible values: mlp, ldp, ldpv2", choices=["mlp", "ldp", "ldpv2"], default="mlp")
ap.add_argument("-o", "--output-dir", help="Directory to save GGUF files. Default is the original model directory", default=None)
# Example --image_mean 0.48145466 0.4578275 0.40821073 --image_std 0.26862954 0.26130258 0.27577711
# Example --image_mean 0.5 0.5 0.5 --image_std 0.5 0.5 0.5
```

```
default_image_mean = [0.48145466, 0.4578275, 0.40821073]
default_image_std = [0.26862954, 0.26130258, 0.27577711]
ap.add_argument('--image-mean', type=float, nargs='+', help='Mean of the images for normalization (overrides processor) ', default=None)
ap.add_argument('--image-std', type=float, nargs='+', help='Standard deviation of the images for normalization (overrides processor)', default=None)

# with proper
args = ap.parse_args()


if args.text_only and args.vision_only:
    print("--text-only and --image-only arguments cannot be specified at the same time.")
    exit(1)

if args.use_f32:
    print("WARNING: Weights for the convolution op is always saved in f16, as the convolution op in GGML does not support 32-bit kernel weights yet.")

# output in the same directory as the model if output_dir is None
dir_model = args.model_dir

if (
    args.clip_model_is_vision or
    not os.path.exists(dir_model + "/vocab.json") or
    args.clip_model_is_openclip or
    args.clip_model_is_siglip
):
    vocab = None
    tokens = None
else:
    with open(dir_model + "/vocab.json", "r", encoding="utf-8") as f:
        vocab = json.load(f)
        tokens = [key for key in vocab]

with open(dir_model + "/config.json", "r", encoding="utf-8") as f:
    config = json.load(f)
    if args.clip_model_is_vision:
        v_hparams = config
        t_hparams = None
    else:
        v_hparams = config["vision_config"]
        t_hparams = config["text_config"]

# possible data types
#   ftype == 0 -> float32
#   ftype == 1 -> float16
#
# map from ftype to string
ftype_str = ["f32", "f16"]

ftype = 1
if args.use_f32:
    ftype = 0
```

```
if args.clip_model_is_siglip:
    model = SiglipVisionModel.from_pretrained(dir_model)
    processor = None
elif args.clip_model_is_vision or args.clip_model_is_openclip:
    model = CLIPVisionModel.from_pretrained(dir_model)
    processor = None
else:
    model = CLIPModel.from_pretrained(dir_model)
    processor = CLIPProcessor.from_pretrained(dir_model)

fname_middle = None
has_text_encoder = True
has_vision_encoder = True
has_llava_projector = False
if args.text_only:
    fname_middle = "text-"
    has_vision_encoder = False
elif args.llava_projector is not None:
    fname_middle = "mmproj-"
    has_text_encoder = False
    has_llava_projector = True
elif args.vision_only:
    fname_middle = "vision-"
    has_text_encoder = False
else:
    fname_middle = ""

output_dir = args.output_dir if args.output_dir is not None else dir_model
os.makedirs(output_dir, exist_ok=True)
output_prefix = os.path.basename(output_dir).replace("ggml_", "")
fname_out = os.path.join(output_dir, f"{fname_middle}model-{ftype_str[ftype]}.gguf")
fout = GGUFWriter(path=fname_out, arch="clip", endianess=GGUFEndian.LITTLE if not args.bigendian else GGUFEndian.BIG)

fout.add_bool("clip.has_text_encoder", has_text_encoder)
fout.add_bool("clip.has_vision_encoder", has_vision_encoder)
fout.add_bool("clip.has_llava_projector", has_llava_projector)
fout.add_file_type(ftype)
model_name = config["_name_or_path"] if "_name_or_path" in config else os.path.basename(dir_model)
fout.add_name(model_name)
if args.text_only:
    fout.add_description("text-only CLIP model")
elif args.vision_only and not has_llava_projector:
    fout.add_description("vision-only CLIP model")
elif has_llava_projector:
    fout.add_description("image encoder for LLaVA")
    # add projector type
    fout.add_string("clip.projector_type", args.projector_type)
else:
    fout.add_description("two-tower CLIP model")

if has_text_encoder:
    assert t_hparams is not None
    assert tokens is not None
    if args.clip_model_is_siglip:
        text_projection_dim = 0
    else:
        text_projection_dim = t_hparams.get("projection_dim", config["projection_dim"])
    # text_model hparams
    fout.add_uint32(k(KEY_CONTEXT_LENGTH, TEXT), t_hparams["max_position_embeddings"])
    fout.add_uint32(k(KEY_EMBEDDING_LENGTH, TEXT), t_hparams["hidden_size"])
    fout.add_uint32(k(KEY_FEED_FORWARD_LENGTH, TEXT), t_hparams["intermediate_size"])
    fout.add_uint32("clip.text.projection_dim", text_projection_dim)
    fout.add_uint32(k(KEY_ATTENTION_HEAD_COUNT, TEXT), t_hparams["num_attention_heads"])
    fout.add_float32(k(KEY_ATTENTION_LAYERNORM_EPS, TEXT), t_hparams["layer_norm_eps"])
    fout.add_uint32(k(KEY_BLOCK_COUNT, TEXT), t_hparams["num_hidden_layers"])
    fout.add_token_list(tokens)



def get_non_negative_vision_feature_layers(v_hparams):
    """
    Determine the vision feature layer(s) for the llava model, which are indices into the
    hidden states of the visual encoder. Note that the hidden states array generally takes the
    form:

        [<emb input>, <output of enc block 0>, ... <output of enc block num_hidden_layers>]

    so feature indices should be offset as n+1 to get the output of encoder block n.
    We convert all vision feature layers to non-negative so that -1 can be used in
    the model as an unset value. If no vision feature layer is found, we leave it unset.
    """
    num_hidden_layers = v_hparams["num_hidden_layers"]
    to_non_negative = lambda layer_idx: layer_idx  if layer_idx >= 0 else num_hidden_layers + layer_idx + 1
    feature_layers_key = None
    # Key used for llava models in transformers
    if "vision_feature_layer" in config:
        feature_layers_key = "vision_feature_layer"
    # Key used for llava models in the original format
    elif "mm_vision_select_layer" in config:
        feature_layers_key = "mm_vision_select_layer"
    if feature_layers_key is not None:
        feature_layers = config[feature_layers_key]
        if isinstance(feature_layers, int):
            feature_layers = [feature_layers]
        return [to_non_negative(feature_layer) for feature_layer in feature_layers]

# Determine if we have explicitly specified vision feature layers in our config
feature_layers = get_non_negative_vision_feature_layers(v_hparams)

if has_vision_encoder:
    # Siglip does not have a visual projector; set projection dim to 0
    if args.clip_model_is_siglip:
        visual_projection_dim = 0
    else:
        visual_projection_dim = v_hparams.get("projection_dim", config["projection_dim"])
    # set vision_model hparams
    fout.add_uint32("clip.vision.image_size", v_hparams["image_size"])
    fout.add_uint32("clip.vision.patch_size", v_hparams["patch_size"])
    fout.add_uint32(k(KEY_EMBEDDING_LENGTH, VISION), v_hparams["hidden_size"])
    fout.add_uint32(k(KEY_FEED_FORWARD_LENGTH, VISION), v_hparams["intermediate_size"])
    fout.add_uint32("clip.vision.projection_dim", visual_projection_dim)
    fout.add_uint32(k(KEY_ATTENTION_HEAD_COUNT, VISION), v_hparams["num_attention_heads"])
    fout.add_float32(k(KEY_ATTENTION_LAYERNORM_EPS, VISION), v_hparams["layer_norm_eps"])
    if feature_layers:
        block_count = max(feature_layers)
    else:
                      block_count = v_hparams["num_hidden_layers"] - 1 if has_llava_projector else
v_hparams["num_hidden_layers"]
    fout.add_uint32(k(KEY_BLOCK_COUNT, VISION), block_count)
                            #     /**
                            #      "image_grid_pinpoints": [
                            #         [
                            #         336,
                            #         672
                            #         ],
                            #         [
                            #         672,
                            #         336
                            #         ],
                            #         [
                            #         672,
                            #         672
                            #         ],
                            #         [
                            #         1008,
                            #         336
                            #         ],
                            #         [
                            #         336,
                            #         1008
                            #         ]
                            #     ],
                            #     Flattened:
                            #     [
                            #         336, 672,
                            #         672, 336,
                            #         672, 672,
                            #         1008, 336,
                            #         336, 1008
                            #     ]
                            #  *
                            #  */
    if "image_grid_pinpoints" in v_hparams:
        # flatten it
        image_grid_pinpoints = []
        for pinpoint in v_hparams["image_grid_pinpoints"]:
            for p in pinpoint:
                image_grid_pinpoints.append(p)
        fout.add_array("clip.vision.image_grid_pinpoints", image_grid_pinpoints)
    if "image_crop_resolution" in v_hparams:
        fout.add_uint32("clip.vision.image_crop_resolution", v_hparams["image_crop_resolution"])
    if "image_aspect_ratio" in v_hparams:
        fout.add_string("clip.vision.image_aspect_ratio", v_hparams["image_aspect_ratio"])
    if "image_split_resolution" in v_hparams:
        fout.add_uint32("clip.vision.image_split_resolution", v_hparams["image_split_resolution"])
    if "mm_patch_merge_type" in v_hparams:
        fout.add_string("clip.vision.mm_patch_merge_type", v_hparams["mm_patch_merge_type"])
    if "mm_projector_type" in v_hparams:
        fout.add_string("clip.vision.mm_projector_type", v_hparams["mm_projector_type"])
    if feature_layers:
        fout.add_array("clip.vision.feature_layer", feature_layers)

    if processor is not None:
        image_mean = processor.image_processor.image_mean if args.image_mean is None or args.image_mean == default_image_mean else args.image_mean  # pyright: ignore[reportAttributeAccessIssue]
        image_std = processor.image_processor.image_std if args.image_std is None or args.image_std == default_image_std else args.image_std  # pyright: ignore[reportAttributeAccessIssue]
    else:
        image_mean = args.image_mean if args.image_mean is not None else default_image_mean
        image_std = args.image_std if args.image_std is not None else default_image_std
    fout.add_array("clip.vision.image_mean", image_mean)
    fout.add_array("clip.vision.image_std", image_std)

use_gelu = v_hparams["hidden_act"] == "gelu"
fout.add_bool("clip.use_gelu", use_gelu)


if has_llava_projector:
    # By default, we drop the last layer for llava projector
    # models unless we have explicitly set vision feature layers
    if feature_layers is None:
        model.vision_model.encoder.layers.pop(-1)
    else:
        model.vision_model.encoder.layers = model.vision_model.encoder.layers[:max(feature_layers)]

    projector = torch.load(args.llava_projector)
    for name, data in projector.items():
        name = get_tensor_name(name)
        # pw and dw conv ndim==4
        if data.ndim == 2 or data.ndim == 4:
            data = data.squeeze().numpy().astype(np.float16)
        else:
            data = data.squeeze().numpy().astype(np.float32)

        fout.add_tensor(name, data)

    print("Projector tensors added\n")

state_dict = model.state_dict()
for name, data in state_dict.items():
    if should_skip_tensor(name, has_text_encoder, has_vision_encoder, has_llava_projector):
        # we don't need this
        print(f"skipping parameter: {name}")
        continue

    name = get_tensor_name(name)
    data = data.squeeze().numpy()

    n_dims = len(data.shape)

    # ftype == 0 -> float32, ftype == 1 -> float16
    ftype_cur = 0
    if n_dims == 4:
        print(f"tensor {name} is always saved in f16")
        data = data.astype(np.float16)
        ftype_cur = 1
    elif ftype == 1:
        if name[-7:] == ".weight" and n_dims == 2:
            print("  Converting to float16")
            data = data.astype(np.float16)
            ftype_cur = 1
        else:
            print("  Converting to float32")
            data = data.astype(np.float32)
            ftype_cur = 0
    else:
        if data.dtype != np.float32:
            print("  Converting to float32")
            data = data.astype(np.float32)
            ftype_cur = 0

    print(f"{name} - {ftype_str[ftype_cur]} - shape = {data.shape}")
    fout.add_tensor(name, data)


fout.write_header_to_file()
fout.write_kv_data_to_file()
fout.write_tensors_to_file()
fout.close()

print("Done. Output file: " + fname_out)

==== convert_legacy_llama.py ====
#!/usr/bin/env python3
from __future__ import annotations

import logging
import argparse
import concurrent.futures
import enum
import faulthandler
import functools
import itertools
import json
import math
import mmap
import os
import pickle
import re
import signal
import struct
import sys
import textwrap
import time
import zipfile
from abc import ABC, abstractmethod
from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor
from dataclasses import dataclass
from pathlib import Path
from typing import TYPE_CHECKING, Any, Callable, IO, Iterable, Literal, TypeVar

import numpy as np

if 'NO_LOCAL_GGUF' not in os.environ:
    # use .parent.parent since we are in "examples" directory
    sys.path.insert(1, str(Path(__file__).parent.parent / 'gguf-py'))

import gguf
from gguf import BaseVocab, Vocab, NoVocab, BpeVocab, SentencePieceVocab, LlamaHfVocab

if TYPE_CHECKING:
    from typing_extensions import Self, TypeAlias

logger = logging.getLogger("convert")

if hasattr(faulthandler, 'register') and hasattr(signal, 'SIGUSR1'):
    faulthandler.register(signal.SIGUSR1)

NDArray: TypeAlias = 'np.ndarray[Any, Any]'

ARCH = gguf.MODEL_ARCH.LLAMA

DEFAULT_CONCURRENCY = 8

ADDED_TOKENS_FILE = 'added_tokens.json'
FAST_TOKENIZER_FILE = 'tokenizer.json'

#
# data types
#


@dataclass(frozen=True)
class DataType:
    name: str
    dtype: np.dtype[Any]
    valid_conversions: list[str]

    def elements_to_bytes(self, n_elements: int) -> int:
        return n_elements * self.dtype.itemsize
```

```
@dataclass(frozen=True)
class UnquantizedDataType(DataType):
    pass


DT_F16  = UnquantizedDataType('F16',  dtype = np.dtype(np.float16), valid_conversions = ['F32', 'Q8_0'])
DT_F32  = UnquantizedDataType('F32',  dtype = np.dtype(np.float32), valid_conversions = ['F16', 'Q8_0'])
DT_I32  = UnquantizedDataType('I32',  dtype = np.dtype(np.int16),   valid_conversions = [])
DT_BF16 = UnquantizedDataType('BF16', dtype = np.dtype(np.uint16),  valid_conversions = ['F32', 'F16', 'Q8_0'])


@dataclass(frozen=True)
class QuantizedDataType(DataType):
    block_size: int
    quantized_dtype: np.dtype[Any]
    ggml_type: gguf.GGMLQuantizationType

    def quantize(self, arr: NDArray) -> NDArray:
        raise NotImplementedError(f'Quantization for {self.name} not implemented')

    def elements_to_bytes(self, n_elements: int) -> int:
        assert n_elements % self.block_size == 0, f'Invalid number of elements {n_elements} for {self.name} with block size {self.block_size}'
        return self.quantized_dtype.itemsize * (n_elements // self.block_size)


@dataclass(frozen=True)
class Q8_0QuantizedDataType(QuantizedDataType):
    # Mini Q8_0 quantization in Python!
    def quantize(self, arr: NDArray) -> NDArray:
        assert arr.size % self.block_size == 0 and arr.size != 0, f'Bad array size {arr.size}'
        assert arr.dtype == np.float32, f'Bad array type {arr.dtype}'
        n_blocks = arr.size // self.block_size
        blocks = arr.reshape((n_blocks, self.block_size))
        # Much faster implementation of block quantization contributed by @Cebtenzzre

        def quantize_blocks_q8_0(blocks: NDArray) -> Iterable[tuple[Any, Any]]:
            d = abs(blocks).max(axis = 1) / np.float32(127)
            with np.errstate(divide = 'ignore'):
                qs = (blocks / d[:, None]).round()
            qs[d == 0] = 0
            yield from zip(d, qs)
        return np.fromiter(quantize_blocks_q8_0(blocks), count = n_blocks, dtype = self.quantized_dtype)


DT_Q8_0 = Q8_0QuantizedDataType('Q8_0',
                                dtype = np.dtype(np.float32), valid_conversions = [],
                                ggml_type = gguf.GGMLQuantizationType.Q8_0, block_size = 32,
                                quantized_dtype = np.dtype([('d', '<f2'), ('qs', 'i1', (32,))]))

# Quantized types skipped here because they may also map to np.float32
NUMPY_TYPE_TO_DATA_TYPE: dict[np.dtype[Any], DataType] = {}
for dt in (DT_BF16, DT_F16, DT_F32, DT_I32):
    if dt.dtype in NUMPY_TYPE_TO_DATA_TYPE:
        raise ValueError(f'Invalid duplicate data type {dt}')
    NUMPY_TYPE_TO_DATA_TYPE[dt.dtype] = dt

SAFETENSORS_DATA_TYPES: dict[str, DataType] = {
    'BF16': DT_BF16,
    'F16': DT_F16,
    'F32': DT_F32,
    'I32': DT_I32,
}

# TODO: match this with `llama_ftype`
# TODO: rename to LLAMAFileType
# TODO: move to `gguf.py`


class GGMLFileType(enum.IntEnum):
    AllF32     = 0
    MostlyF16  = 1  # except 1d tensors
    MostlyQ8_0 = 7  # except 1d tensors

    def type_for_tensor(self, name: str, tensor: LazyTensor) -> DataType:
        dt = GGML_FILE_TYPE_TO_DATA_TYPE.get(self)
        if dt is None:
            raise ValueError(self)
        # Convert all 1D tensors to F32.  Most of the codebase that takes in 1D tensors only handles F32 tensors, and most of the outputs tensors are F32.
        #  Also The 1d tensors aren't much of a performance/size issue.  So instead of having to have separate F32 and F16 implementations of both, just convert everything to F32 for now.
        return dt if len(tensor.shape) > 1 else DT_F32


GGML_FILE_TYPE_TO_DATA_TYPE: dict[GGMLFileType, DataType] = {
    GGMLFileType.AllF32    : DT_F32,
    GGMLFileType.MostlyF16 : DT_F16,
    GGMLFileType.MostlyQ8_0: DT_Q8_0,
}

#
# hparams loading
#


@dataclass
class Params:
    n_vocab:        int
    n_embd:         int
    n_layer:        int
    n_ctx:          int
    n_ff:           int
    n_head:         int
    n_head_kv:      int
    n_experts:      int | None = None
    n_experts_used: int | None = None
    f_norm_eps:     float | None = None

    rope_scaling_type: gguf.RopeScalingType | None = None
    f_rope_freq_base: float | None = None
    f_rope_scale: float | None = None
    n_ctx_orig: int | None = None
    rope_finetuned: bool | None = None

    ftype: GGMLFileType | None = None

    # path to the directory containing the model files
    path_model: Path | None = None

    @staticmethod
    def guessed(model: LazyModel) -> Params:
        # try transformer naming first
        n_vocab, n_embd = model["model.embed_tokens.weight"].shape if "model.embed_tokens.weight" in model else model["tok_embeddings.weight"].shape

        # try transformer naming first
        if "model.layers.0.self_attn.q_proj.weight" in model:
            n_layer = next(i for i in itertools.count() if f"model.layers.{i}.self_attn.q_proj.weight" not in model)
        elif "model.layers.0.self_attn.W_pack.weight" in model:   # next: try baichuan naming
            n_layer = next(i for i in itertools.count() if f"model.layers.{i}.self_attn.W_pack.weight" not in model)
        else:
            n_layer = next(i for i in itertools.count() if f"layers.{i}.attention.wq.weight" not in model)

        if n_layer < 1:
            msg = """\
                failed to guess 'n_layer'. This model is unknown or unsupported.
                Suggestion: provide 'config.json' of the model in the same directory containing model files."""
            raise KeyError(textwrap.dedent(msg))

        n_head = n_embd // 128 # guessed
        n_mult = 256           # guessed

        # TODO: verify this
        n_ff = int(2 * (4 * n_embd) / 3)
        n_ff = n_mult * ((n_ff + n_mult - 1) // n_mult)

        return Params(
            n_vocab    = n_vocab,
            n_embd     = n_embd,
            n_layer    = n_layer,
            n_ctx      = -1,
            n_ff       = n_ff,
            n_head     = n_head,
            n_head_kv  = n_head,
            f_norm_eps = 1e-5,
        )

    @staticmethod
```

```
def loadHFTransformerJson(model: LazyModel, config_path: Path) -> Params:
    with open(config_path) as f:
        config = json.load(f)

    rope_scaling_type = f_rope_scale = n_ctx_orig = rope_finetuned = None
    rope_scaling = config.get("rope_scaling")

    if rope_scaling is not None and (typ := rope_scaling.get("type")):
        rope_factor = rope_scaling.get("factor")
        f_rope_scale = rope_factor
        if typ == "linear":
            rope_scaling_type = gguf.RopeScalingType.LINEAR
        elif typ == "yarn":
            rope_scaling_type = gguf.RopeScalingType.YARN
            n_ctx_orig = rope_scaling['original_max_position_embeddings']
            rope_finetuned = rope_scaling['finetuned']
        else:
            raise NotImplementedError(f'Unknown rope scaling type: {typ}')

    if "max_sequence_length" in config:
        n_ctx = config["max_sequence_length"]
    elif "max_position_embeddings" in config:
        n_ctx = config["max_position_embeddings"]
    else:
        msg = """\
            failed to guess 'n_ctx'. This model is unknown or unsupported.
            Suggestion: provide 'config.json' of the model in the same directory containing model files."""
        raise KeyError(textwrap.dedent(msg))

    n_experts      = None
    n_experts_used = None

    if "num_local_experts" in config:
        n_experts = config["num_local_experts"]
        n_experts_used = config["num_experts_per_tok"]

    return Params(
        n_vocab           = config["vocab_size"],
        n_embd            = config["hidden_size"],
        n_layer           = config["num_hidden_layers"],
        n_ctx             = n_ctx,
        n_ff              = config["intermediate_size"],
        n_head            = (n_head := config["num_attention_heads"]),
        n_head_kv         = config.get("num_key_value_heads", n_head),
        n_experts         = n_experts,
        n_experts_used    = n_experts_used,
        f_norm_eps        = config["rms_norm_eps"],
        f_rope_freq_base  = config.get("rope_theta"),
        rope_scaling_type = rope_scaling_type,
        f_rope_scale      = f_rope_scale,
        n_ctx_orig        = n_ctx_orig,
        rope_finetuned    = rope_finetuned,
    )
# LLaMA v2 70B params.json
# {"dim": 8192, "multiple_of": 4096, "ffn_dim_multiplier": 1.3, "n_heads": 64, "n_kv_heads": 8, "n_layers": 80, "norm_eps": 1e-05, "vocab_size": -1}
@staticmethod
def loadOriginalParamsJson(model: LazyModel, config_path: Path) -> Params:
    with open(config_path) as f:
        config = json.load(f)

    n_experts      = None
    n_experts_used = None
    f_rope_freq_base = None
    n_ff = None

    # hack to determine LLaMA v1 vs v2 vs CodeLlama
    if config.get("moe"):
        # Mixtral
        n_ctx = 32768
    elif config.get("rope_theta") == 1000000:
        # CodeLlama
        n_ctx = 16384
    elif config["norm_eps"] == 1e-05:
        # LLaMA v2
        n_ctx = 4096
    else:
        # LLaMA v1
        n_ctx = 2048

    if "layers.0.feed_forward.w1.weight" in model:
        n_ff = model["layers.0.feed_forward.w1.weight"].shape[0]

    if config.get("moe"):
        n_ff = model["layers.0.feed_forward.experts.0.w1.weight"].shape[0]
        n_experts      = config["moe"]["num_experts"]
        n_experts_used = config["moe"]["num_experts_per_tok"]
        f_rope_freq_base = 1e6

    assert n_ff is not None

    return Params(
        n_vocab          = model["tok_embeddings.weight"].shape[0],
        n_embd           = config["dim"],
        n_layer          = config["n_layers"],
        n_ctx            = n_ctx,
        n_ff             = n_ff,
        n_head           = (n_head := config["n_heads"]),
        n_head_kv        = config.get("n_kv_heads", n_head),
        n_experts        = n_experts,
        n_experts_used   = n_experts_used,
        f_norm_eps       = config["norm_eps"],
        f_rope_freq_base = config.get("rope_theta", f_rope_freq_base),
    )

@staticmethod
def load(model_plus: ModelPlus) -> Params:
    hf_config_path   = model_plus.paths[0].parent / "config.json"
    orig_config_path = model_plus.paths[0].parent / "params.json"

    if hf_config_path.exists():
        params = Params.loadHFTransformerJson(model_plus.model, hf_config_path)
    elif orig_config_path.exists():
        params = Params.loadOriginalParamsJson(model_plus.model, orig_config_path)
    elif model_plus.format != 'none':
        params = Params.guessed(model_plus.model)
    else:
        raise ValueError('Cannot guess params when model format is none')

    params.path_model = model_plus.paths[0].parent

    return params


#
# data loading
# TODO: reuse (probably move to gguf.py?)
#


def permute(weights: NDArray, n_head: int, n_head_kv: int) -> NDArray:
if n_head_kv is not None and n_head != n_head_kv:
    n_head = n_head_kv
return (weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
        .swapaxes(1, 2)
        .reshape(weights.shape))


class Tensor(ABC):
ndarray: NDArray
data_type: DataType

@abstractmethod
def astype(self, data_type: DataType) -> Self: ...
@abstractmethod
def permute(self, n_head: int, n_head_kv: int) -> Self: ...
@abstractmethod
def permute_part(self, n_part: int, n_head: int, n_head_kv: int) -> Self: ...
@abstractmethod
def part(self, n_part: int) -> Self: ...
@abstractmethod
def to_ggml(self) -> GGMLCompatibleTensor: ...


def bf16_to_fp32(bf16_arr: np.ndarray[Any, np.dtype[np.uint16]]) -> NDArray:
assert bf16_arr.dtype == np.uint16, f"Input array should be of dtype uint16, but got {bf16_arr.dtype}"
fp32_arr = bf16_arr.astype(np.uint32) << 16
return fp32_arr.view(np.float32)


class UnquantizedTensor(Tensor):
def __init__(self, ndarray: NDArray):
    assert isinstance(ndarray, np.ndarray)
    self.ndarray = ndarray
    self.data_type = NUMPY_TYPE_TO_DATA_TYPE[ndarray.dtype]

def astype(self, data_type: DataType) -> UnquantizedTensor:
    dtype = data_type.dtype
    if self.data_type == DT_BF16:
        self.ndarray = bf16_to_fp32(self.ndarray)
    return UnquantizedTensor(self.ndarray.astype(dtype))

def to_ggml(self) -> Self:
    return self

def permute_part(self, n_part: int, n_head: int, n_head_kv: int) -> UnquantizedTensor:
    r = self.ndarray.shape[0] // 3
    return UnquantizedTensor(permute(self.ndarray[r * n_part : r * n_part + r, ...], n_head, n_head_kv))

def part(self, n_part: int) -> UnquantizedTensor:
    r = self.ndarray.shape[0] // 3
    return UnquantizedTensor(self.ndarray[r * n_part : r * n_part + r, ...])

def permute(self, n_head: int, n_head_kv: int) -> UnquantizedTensor:
    return UnquantizedTensor(permute(self.ndarray, n_head, n_head_kv))


def load_unquantized(lazy_tensor: LazyTensor, expected_dtype: Any = None, convert: bool = False) -> NDArray:
tensor = lazy_tensor.load()
assert isinstance(tensor, UnquantizedTensor)

# double-check:
actual_shape = list(tensor.ndarray.shape)
assert actual_shape == lazy_tensor.shape, (actual_shape, lazy_tensor.shape)
if expected_dtype is not None and expected_dtype != tensor.ndarray.dtype:
    if convert:
        tensor.ndarray = tensor.ndarray.astype(expected_dtype)
    else:
        raise ValueError(f'expected this tensor to have dtype {expected_dtype}, got
{tensor.ndarray.dtype}')

return tensor.ndarray


GGMLCompatibleTensor = UnquantizedTensor


@dataclass
class LazyTensor:
_load: Callable[[], Tensor]
shape: list[int]
data_type: DataType
description: str

def load(self) -> Tensor:
    ret = self._load()
    # Should be okay if it maps to the same numpy type?
    assert ret.data_type == self.data_type or (self.data_type.dtype == ret.data_type.dtype), \
        (self.data_type, ret.data_type, self.description)
    return ret

def astype(self, data_type: DataType) -> LazyTensor:
    self.validate_conversion_to(data_type)

    def load() -> Tensor:
        return self.load().astype(data_type)
    return LazyTensor(load, self.shape, data_type, f'convert({data_type}) {self.description}')

def validate_conversion_to(self, data_type: DataType) -> None:
    if data_type != self.data_type and data_type.name not in self.data_type.valid_conversions:
        raise ValueError(f'Cannot validate conversion from {self.data_type} to {data_type}.')


LazyModel: TypeAlias = 'dict[str, LazyTensor]'

ModelFormat: TypeAlias = Literal['ggml', 'torch', 'safetensors', 'none']

@dataclass
class ModelPlus:
model: LazyModel
paths: list[Path]  # Where this was read from.
format: ModelFormat
vocab: BaseVocab | None  # For GGML models (which have vocab built in), the vocab.


def merge_sharded(models: list[LazyModel]) -> LazyModel:
# Original LLaMA models have each file contain one part of each tensor.
# Use a dict instead of a set to preserve order.
names = {name: None for model in models for name in model}

def convert(name: str) -> LazyTensor:
    lazy_tensors = [model[name] for model in models]
    if len(lazy_tensors) == 1:
        # only one file; don't go through this procedure since there might
        # be quantized tensors
        return lazy_tensors[0]
    if len(lazy_tensors[0].shape) == 1:
        # the tensor is just duplicated in every file
        return lazy_tensors[0]
    if name.startswith('tok_embeddings.') or \
       name.endswith('.attention.wo.weight') or \
       name.endswith('.feed_forward.w2.weight'):
        # split by columns
        axis = 1
    else:
        # split by rows
        axis = 0
    concatenated_shape = list(lazy_tensors[0].shape)
    concatenated_shape[axis] = sum(tensor.shape[axis] for tensor in lazy_tensors)
    def load() -> UnquantizedTensor:
        ndarrays = [load_unquantized(tensor) for tensor in lazy_tensors]
        concatenated = np.concatenate(ndarrays, axis=axis)
        return UnquantizedTensor(concatenated)
    description = 'concatenated[[' + '] | ['.join(lt.description for lt in lazy_tensors) + ']]'
    return LazyTensor(load, concatenated_shape, lazy_tensors[0].data_type, description)
return {name: convert(name) for name in names}


def merge_multifile_models(models_plus: list[ModelPlus]) -> ModelPlus:
formats: set[ModelFormat] = set(mp.format for mp in models_plus)
assert len(formats) == 1, "different formats?"
format = formats.pop()
paths = [path for mp in models_plus for path in mp.paths]
# Use the first non-None vocab, if any.
try:
    vocab = next(mp.vocab for mp in models_plus if mp.vocab is not None)
except StopIteration:
    vocab = None

if any("model.embed_tokens.weight" in mp.model for mp in models_plus):
    # Transformers models put different tensors in different files, but
    # don't split individual tensors between files.
    model: LazyModel = {}
    for mp in models_plus:
        model.update(mp.model)
else:
    model = merge_sharded([mp.model for mp in models_plus])

return ModelPlus(model, paths, format, vocab)


def permute_lazy(lazy_tensor: LazyTensor, n_head: int, n_head_kv: int) -> LazyTensor:
def load() -> Tensor:
    return lazy_tensor.load().permute(n_head, n_head_kv)
return LazyTensor(load, lazy_tensor.shape, lazy_tensor.data_type, f'permute({n_head}, {n_head_kv}) ' + lazy_tensor.description)


def permute_part_lazy(lazy_tensor: LazyTensor, n_part: int, n_head: int, n_head_kv: int) -> LazyTensor:
def load() -> Tensor:
    return lazy_tensor.load().permute_part(n_part, n_head, n_head_kv)
s = lazy_tensor.shape.copy()
s[0] = s[0] // 3
return LazyTensor(load, s, lazy_tensor.data_type, f'permute({n_head}, {n_head_kv}) ' + lazy_tensor.description)


def part_lazy(lazy_tensor: LazyTensor, n_part: int) -> LazyTensor:
def load() -> Tensor:
    return lazy_tensor.load().part(n_part)
s = lazy_tensor.shape.copy()
s[0] = s[0] // 3
return LazyTensor(load, s, lazy_tensor.data_type, 'part ' + lazy_tensor.description)


def pack_experts_lazy(lazy_tensors: list[LazyTensor]) -> LazyTensor:
def load() -> Tensor:
    tensors = [lazy_tensor.load() for lazy_tensor in lazy_tensors]
    return UnquantizedTensor(np.array([tensor.ndarray for tensor in tensors]))
s = lazy_tensors[0].shape.copy()
s.insert(0, len(lazy_tensors))
return LazyTensor(load, s, lazy_tensors[0].data_type, 'pack_experts ' + ' | '.join(lt.description for lt in lazy_tensors))


# Functionality that simulates `torch.load` but where individual tensors are
# only loaded into memory on demand, not all at once.
# PyTorch can't do this natively as of time of writing:
# - https://github.com/pytorch/pytorch/issues/64327
# This allows us to de-shard without multiplying RAM usage, and also
# conveniently drops the PyTorch dependency (though we still need numpy).


@dataclass
class LazyStorageKind:
data_type: DataType


@dataclass
class LazyStorage:
load: Callable[[int, int], NDArray]
kind: LazyStorageKind
description: str


class LazyUnpickler(pickle.Unpickler):
def __init__(self, fp: IO[bytes], data_base_path: str, zip_file: zipfile.ZipFile):
    super().__init__(fp)
    self.data_base_path = data_base_path
    self.zip_file = zip_file

def persistent_load(self, pid: Any) -> Any:
    assert pid[0] == 'storage'
    assert isinstance(pid[1], LazyStorageKind)
    data_type = pid[1].data_type
    filename_stem = pid[2]
    filename = f'{self.data_base_path}/{filename_stem}'
    info = self.zip_file.getinfo(filename)

    def load(offset: int, elm_count: int) -> NDArray:
        dtype = data_type.dtype
        with self.zip_file.open(info) as fp:
            fp.seek(offset * dtype.itemsize)
            size = elm_count * dtype.itemsize
            data = fp.read(size)
        assert len(data) == size
        return np.frombuffer(data, dtype)
    description = f'storage data_type={data_type} path-in-zip={filename} path={self.zip_file.filename}'
    return LazyStorage(load=load, kind=pid[1], description=description)

@staticmethod
def lazy_rebuild_tensor_v2(storage: Any, storage_offset: Any, size: Any, stride: Any,
                           requires_grad: Any, backward_hooks: Any, metadata: Any = None) -> LazyTensor:
    assert isinstance(storage, LazyStorage)

    def load() -> UnquantizedTensor:
        elm_count = stride[0] * size[0]
        return UnquantizedTensor(storage.load(storage_offset, elm_count).reshape(size))
    description = f'pickled storage_offset={storage_offset} in {storage.description}'
    return LazyTensor(load, list(size), storage.kind.data_type, description)

@staticmethod
def rebuild_from_type_v2(func, new_type, args, state):
    return func(*args)

CLASSES: dict[tuple[str, str], type[LazyTensor] | LazyStorageKind] = {
    # getattr used here as a workaround for mypy not being smart enough to determine
    # the staticmethods have a __func__ attribute.
    ('torch._tensor', '_rebuild_from_type_v2'): getattr(rebuild_from_type_v2, '__func__'),
    ('torch._utils', '_rebuild_tensor_v2'): getattr(lazy_rebuild_tensor_v2, '__func__'),
    ('torch', 'BFloat16Storage'): LazyStorageKind(DT_BF16),
    ('torch', 'HalfStorage'): LazyStorageKind(DT_F16),
    ('torch', 'FloatStorage'): LazyStorageKind(DT_F32),
    ('torch', 'IntStorage'): LazyStorageKind(DT_I32),
    ('torch', 'Tensor'): LazyTensor,
}

def find_class(self, module: str, name: str) -> Any:
    if not module.startswith('torch'):
        return super().find_class(module, name)
    return self.CLASSES[(module, name)]


def lazy_load_torch_file(outer_fp: IO[bytes], path: Path) -> ModelPlus:
zf = zipfile.ZipFile(outer_fp)
pickle_paths = [name for name in zf.namelist() if name.endswith('.pkl')]
assert len(pickle_paths) == 1, pickle_paths
pickle_fp = zf.open(pickle_paths[0], 'r')
unpickler = LazyUnpickler(pickle_fp,
                          data_base_path=pickle_paths[0][:-4],
                          zip_file=zf)
model = unpickler.load()
if 'model' in model: model = model['model']
as_dict = dict(model.items())
return ModelPlus(model=as_dict, paths=[path], format='torch', vocab=None)


def lazy_load_safetensors_file(fp: IO[bytes], path: Path) -> ModelPlus:
header_size, = struct.unpack('<Q', fp.read(8))
header: dict[str, dict[str, Any]] = json.loads(fp.read(header_size))
# Use mmap for the actual data to avoid race conditions with the file offset.
mapped = memoryview(mmap.mmap(fp.fileno(), 0, access=mmap.ACCESS_READ))
byte_buf = mapped[8 + header_size:]

def convert(info: dict[str, Any]) -> LazyTensor:
    data_type = SAFETENSORS_DATA_TYPES[info['dtype']]
    numpy_dtype = data_type.dtype
    shape: list[int] = info['shape']
    begin, end = info['data_offsets']
    assert 0 <= begin <= end <= len(byte_buf)
    assert end - begin == math.prod(shape) * numpy_dtype.itemsize
    buf = byte_buf[begin:end]

    def load() -> UnquantizedTensor:
        return UnquantizedTensor(np.frombuffer(buf, dtype=numpy_dtype).reshape(shape))
    description = f'safetensors begin={begin} end={end} type={data_type} path={path}'
    return LazyTensor(load, shape, data_type, description)
model = {name: convert(info) for (name, info) in header.items() if name != '__metadata__'}
return ModelPlus(model=model, paths=[path], format='safetensors', vocab=None)


def must_read(fp: IO[bytes], length: int) -> bytes:
ret = fp.read(length)
if len(ret) < length:
    raise EOFError("unexpectedly reached end of file")
return ret


@functools.lru_cache(maxsize=None)
def lazy_load_file(path: Path) -> ModelPlus:
fp = open(path, 'rb')
first8 = fp.read(8)
fp.seek(0)
if first8[:2] == b'PK':
    # A zip file, i.e. PyTorch format
    return lazy_load_torch_file(fp, path)
elif struct.unpack('<Q', first8)[0] < 16 * 1024 * 1024:
    # Probably safetensors
    return lazy_load_safetensors_file(fp, path)
else:
    raise ValueError(f"unknown format: {path}")


In = TypeVar('In')
Out = TypeVar('Out')


def bounded_parallel_map(func: Callable[[In], Out], iterable: Iterable[In], concurrency: int, max_workers: int
| None = None, use_processpool_executor: bool = False) -> Iterable[Out]:
'''Parallel map, but with backpressure.  If the caller doesn't call `next`
fast enough, this will stop calling `func` at some point rather than
letting results pile up in memory.  Specifically, there is a max of one
output value buffered per thread.'''
if concurrency < 2:
    yield from map(func, iterable)
    # Not reached.
iterable = iter(iterable)
executor_class: type[ThreadPoolExecutor] | type[ProcessPoolExecutor]
if use_processpool_executor:
    executor_class = ProcessPoolExecutor
else:
    executor_class = ThreadPoolExecutor
with executor_class(max_workers=max_workers) as executor:
    futures: list[concurrent.futures.Future[Out]] = []
    done = False
    for _ in range(concurrency):
        try:
            futures.append(executor.submit(func, next(iterable)))
        except StopIteration:
            done = True
            break

    while futures:
        result = futures.pop(0).result()
        while not done and len(futures) < concurrency:
            try:
                futures.append(executor.submit(func, next(iterable)))
            except StopIteration:
                done = True
                break
        yield result


def check_vocab_size(params: Params, vocab: BaseVocab, pad_vocab: bool = False) -> None:
# Handle special case where the model's vocab size is not set
if params.n_vocab == -1:
    raise ValueError(
        "The model's vocab size is set to -1 in params.json. Please update it manually."
        + (f" Maybe {vocab.vocab_size}?" if isinstance(vocab, Vocab) else ""),
    )
if not isinstance(vocab, Vocab):
    return  # model has no vocab

# Check for a vocab size mismatch
if params.n_vocab == vocab.vocab_size:
    logger.warning("Ignoring added_tokens.json since model matches vocab size without it.")
    return

if pad_vocab and params.n_vocab > vocab.vocab_size:
    pad_count = params.n_vocab - vocab.vocab_size
    logger.debug(
        f"Padding vocab with {pad_count} token(s) - <dummy00001> through <dummy{pad_count:05}>"
    )
    for i in range(1, pad_count + 1):
        vocab.added_tokens_dict[f"<dummy{i:05}>"] = -1
        vocab.added_tokens_list.append(f"<dummy{i:05}>")
    vocab.vocab_size = params.n_vocab
    return
msg = f"Vocab size mismatch (model has {params.n_vocab}, but {vocab.fname_tokenizer} has {vocab.vocab_size})."
if vocab.vocab_size < params.n_vocab < vocab.vocab_size + 20:
    msg += f"  Most likely you are missing added_tokens.json (should be in {vocab.fname_tokenizer.parent})."
if vocab.vocab_size < params.n_vocab:
    msg += " Add the --pad-vocab option and try again."

raise ValueError(msg)


class OutputFile:
def __init__(self, fname_out: Path, endianess:gguf.GGUFEndian = gguf.GGUFEndian.LITTLE):
    self.gguf = gguf.GGUFWriter(fname_out, gguf.MODEL_ARCH_NAMES[ARCH], endianess=endianess)

def add_meta_model(self, params: Params, metadata: gguf.Metadata | None) -> None:
    # Metadata About The Model And Its Provenence
    name = "LLaMA"
    if metadata is not None and metadata.name is not None:
        name = metadata.name
    elif params.path_model is not None:
        name = params.path_model.name
    elif params.n_ctx == 4096:
        # Heuristic detection of LLaMA v2 model
        name = "LLaMA v2"

    self.gguf.add_name(name)

    if metadata is not None:
        if metadata.author is not None:
            self.gguf.add_author(metadata.author)
        if metadata.version is not None:
            self.gguf.add_version(metadata.version)
        if metadata.organization is not None:
            self.gguf.add_organization(metadata.organization)

        if metadata.finetune is not None:
            self.gguf.add_finetune(metadata.finetune)
        if metadata.basename is not None:
            self.gguf.add_basename(metadata.basename)

        if metadata.description is not None:
            self.gguf.add_description(metadata.description)
        if metadata.quantized_by is not None:
            self.gguf.add_quantized_by(metadata.quantized_by)

        if metadata.size_label is not None:
            self.gguf.add_size_label(metadata.size_label)

        if metadata.license is not None:
            self.gguf.add_license(metadata.license)
        if metadata.license_name is not None:
            self.gguf.add_license_name(metadata.license_name)
        if metadata.license_link is not None:
            self.gguf.add_license_link(metadata.license_link)

        if metadata.url is not None:
            self.gguf.add_url(metadata.url)
        if metadata.doi is not None:
            self.gguf.add_doi(metadata.doi)
        if metadata.uuid is not None:
            self.gguf.add_uuid(metadata.uuid)
        if metadata.repo_url is not None:
            self.gguf.add_repo_url(metadata.repo_url)

        if metadata.source_url is not None:
            self.gguf.add_source_url(metadata.source_url)
        if metadata.source_doi is not None:
            self.gguf.add_source_doi(metadata.source_doi)
        if metadata.source_uuid is not None:
            self.gguf.add_source_uuid(metadata.source_uuid)
        if metadata.source_repo_url is not None:
            self.gguf.add_source_repo_url(metadata.source_repo_url)

        if metadata.base_models is not None:
            self.gguf.add_base_model_count(len(metadata.base_models))
            for key, base_model_entry in enumerate(metadata.base_models):
                if "name" in base_model_entry:
                    self.gguf.add_base_model_name(key, base_model_entry["name"])
                if "author" in base_model_entry:
                    self.gguf.add_base_model_author(key, base_model_entry["author"])
                if "version" in base_model_entry:
                    self.gguf.add_base_model_version(key, base_model_entry["version"])
                if "organization" in base_model_entry:
                    self.gguf.add_base_model_organization(key, base_model_entry["organization"])
                if "description" in base_model_entry:
                    self.gguf.add_base_model_description(key, base_model_entry["description"])
                if "url" in base_model_entry:
                    self.gguf.add_base_model_url(key, base_model_entry["url"])
                if "doi" in base_model_entry:
                    self.gguf.add_base_model_doi(key, base_model_entry["doi"])
                if "uuid" in base_model_entry:
                    self.gguf.add_base_model_uuid(key, base_model_entry["uuid"])
                if "repo_url" in base_model_entry:
                    self.gguf.add_base_model_repo_url(key, base_model_entry["repo_url"])

        if metadata.datasets is not None:
            self.gguf.add_dataset_count(len(metadata.datasets))
            for key, dataset_entry in enumerate(metadata.datasets):
                if "name" in dataset_entry:
                    self.gguf.add_dataset_name(key, dataset_entry["name"])
                if "author" in dataset_entry:
                    self.gguf.add_dataset_author(key, dataset_entry["author"])
                if "version" in dataset_entry:
                    self.gguf.add_dataset_version(key, dataset_entry["version"])
                if "organization" in dataset_entry:
                    self.gguf.add_dataset_organization(key, dataset_entry["organization"])
                if "description" in dataset_entry:
                    self.gguf.add_dataset_description(key, dataset_entry["description"])
                if "url" in dataset_entry:
                    self.gguf.add_dataset_url(key, dataset_entry["url"])
                if "doi" in dataset_entry:
                    self.gguf.add_dataset_doi(key, dataset_entry["doi"])
                if "uuid" in dataset_entry:
                    self.gguf.add_dataset_uuid(key, dataset_entry["uuid"])
                if "repo_url" in dataset_entry:
                    self.gguf.add_dataset_repo_url(key, dataset_entry["repo_url"])

        if metadata.tags is not None:
            self.gguf.add_tags(metadata.tags)
        if metadata.languages is not None:
            self.gguf.add_languages(metadata.languages)

def add_meta_arch(self, params: Params) -> None:
    # Metadata About The Neural Architecture Itself
    self.gguf.add_vocab_size(params.n_vocab)
    self.gguf.add_context_length(params.n_ctx)
    self.gguf.add_embedding_length(params.n_embd)
    self.gguf.add_block_count(params.n_layer)
    self.gguf.add_feed_forward_length(params.n_ff)
    self.gguf.add_rope_dimension_count(params.n_embd // params.n_head)
    self.gguf.add_head_count          (params.n_head)
    self.gguf.add_head_count_kv       (params.n_head_kv)

    if params.n_experts:
        self.gguf.add_expert_count(params.n_experts)

    if params.n_experts_used:
        self.gguf.add_expert_used_count(params.n_experts_used)

    if params.f_norm_eps:
        self.gguf.add_layer_norm_rms_eps(params.f_norm_eps)
    else:
        raise ValueError('f_norm_eps is None')

    if params.f_rope_freq_base is not None:
        self.gguf.add_rope_freq_base(params.f_rope_freq_base)

    if params.rope_scaling_type:
        assert params.f_rope_scale is not None
        self.gguf.add_rope_scaling_type(params.rope_scaling_type)
        self.gguf.add_rope_scaling_factor(params.f_rope_scale)

    if params.n_ctx_orig is not None:
        self.gguf.add_rope_scaling_orig_ctx_len(params.n_ctx_orig)

    if params.rope_finetuned is not None:
        self.gguf.add_rope_scaling_finetuned(params.rope_finetuned)

    if params.ftype is not None:
        self.gguf.add_file_type(params.ftype)
def extract_vocabulary_from_model(self, vocab: Vocab) -> tuple[list[bytes], list[float], list[gguf.TokenType]]:
    tokens = []
    scores = []
    toktypes = []

    # NOTE: `all_tokens` returns the base vocabulary and added tokens
    for text, score, toktype in vocab.all_tokens():
        tokens.append(text)
        scores.append(score)
        toktypes.append(toktype)

    assert len(tokens) == vocab.vocab_size

    return tokens, scores, toktypes

def add_meta_vocab(self, vocab: Vocab) -> None:
    # Ensure that tokenizer_model is added to the GGUF model
    self.gguf.add_tokenizer_model(vocab.tokenizer_model)

    # Extract model vocabulary for model conversion
    tokens, scores, toktypes = self.extract_vocabulary_from_model(vocab)

    # Add extracted token information for model conversion
    self.gguf.add_token_list(tokens)
    self.gguf.add_token_scores(scores)
    self.gguf.add_token_types(toktypes)

def add_meta_special_vocab(self, svocab: gguf.SpecialVocab) -> None:
    svocab.add_to_gguf(self.gguf)

def add_tensor_info(self, name: str, tensor: LazyTensor) -> None:
    n_elements = int(np.prod(tensor.shape))
    raw_dtype = getattr(tensor.data_type, 'ggml_type', None)
    data_type = getattr(tensor.data_type, 'quantized_type', None) or tensor.data_type.dtype
    data_nbytes = tensor.data_type.elements_to_bytes(n_elements)
    self.gguf.add_tensor_info(name, tensor.shape, data_type, data_nbytes, raw_dtype=raw_dtype)

def write_meta(self) -> None:
    self.gguf.write_header_to_file()
    self.gguf.write_kv_data_to_file()

def write_tensor_info(self) -> None:
    self.gguf.write_ti_data_to_file()

def write_tensor_data(self, ftype: GGMLFileType, model: LazyModel, concurrency: int) -> None:
    ndarrays_inner = bounded_parallel_map(OutputFile.do_item, model.items(), concurrency=concurrency)
    if ftype == GGMLFileType.MostlyQ8_0:
        ndarrays = bounded_parallel_map(
            OutputFile.maybe_do_quantize, ndarrays_inner, concurrency=concurrency, max_workers=concurrency,
            use_processpool_executor=True,
        )
    else:
        ndarrays = map(OutputFile.maybe_do_quantize, ndarrays_inner)

    start = time.time()
    for i, ((name, lazy_tensor), ndarray) in enumerate(zip(model.items(), ndarrays)):
        elapsed = time.time() - start
        size = ' x '.join(f"{dim:6d}" for dim in lazy_tensor.shape)
        padi = len(str(len(model)))
        logger.info(
                    f"[{i + 1:{padi}d}/{len(model)}] Writing tensor {name:38s} | size {size:16} | type {lazy_tensor.data_type.name:4} | T+{int(elapsed):4}"
        )
        self.gguf.write_tensor_data(ndarray)

def close(self) -> None:
    self.gguf.close()

@staticmethod
def write_vocab_only(
    fname_out: Path, params: Params, vocab: Vocab, svocab: gguf.SpecialVocab,
    endianess: gguf.GGUFEndian = gguf.GGUFEndian.LITTLE, pad_vocab: bool = False, metadata: gguf.Metadata | None = None,
) -> None:
    check_vocab_size(params, vocab, pad_vocab=pad_vocab)

    of = OutputFile(fname_out, endianess=endianess)

    # meta data
    of.add_meta_model(params, metadata)
    of.add_meta_arch(params)
    of.add_meta_vocab(vocab)
    of.add_meta_special_vocab(svocab)

    of.write_meta()

    of.close()

@staticmethod
def do_item(item: tuple[str, LazyTensor]) -> tuple[DataType, NDArray]:
    name, lazy_tensor = item
    tensor = lazy_tensor.load().to_ggml()
    return (lazy_tensor.data_type, tensor.ndarray)

@staticmethod
def maybe_do_quantize(item: tuple[DataType, NDArray]) -> NDArray:
    dt, arr = item
    if not isinstance(dt, QuantizedDataType):
        return arr
    return dt.quantize(arr)

@staticmethod
def write_all(
    fname_out: Path, ftype: GGMLFileType, params: Params, model: LazyModel, vocab: BaseVocab, svocab: gguf.SpecialVocab,
    concurrency: int = DEFAULT_CONCURRENCY, endianess: gguf.GGUFEndian = gguf.GGUFEndian.LITTLE,
    pad_vocab: bool = False,
    metadata: gguf.Metadata | None = None,
) -> None:
    check_vocab_size(params, vocab, pad_vocab=pad_vocab)

    of = OutputFile(fname_out, endianess=endianess)

    # meta data
    of.add_meta_model(params, metadata)
    of.add_meta_arch(params)
    if isinstance(vocab, Vocab):
        of.add_meta_vocab(vocab)
        of.add_meta_special_vocab(svocab)
    else:  # NoVocab
        of.gguf.add_tokenizer_model(vocab.tokenizer_model)

    # tensor info
    for name, lazy_tensor in model.items():
        of.add_tensor_info(name, lazy_tensor)

    of.write_meta()
    of.write_tensor_info()

    # tensor data
    of.write_tensor_data(ftype, model, concurrency)

    of.close()


def pick_output_type(model: LazyModel, output_type_str: str | None) -> GGMLFileType:
wq_type = model[gguf.TENSOR_NAMES[gguf.MODEL_TENSOR.ATTN_Q].format(bid=0) + ".weight"].data_type

if output_type_str == "f32" or (output_type_str is None and wq_type in (DT_F32, DT_BF16)):
    return GGMLFileType.AllF32
if output_type_str == "f16" or (output_type_str is None and wq_type == DT_F16):
    return GGMLFileType.MostlyF16
if output_type_str == "q8_0":
    return GGMLFileType.MostlyQ8_0

name_to_type = {name: lazy_tensor.data_type for (name, lazy_tensor) in model.items()}

raise ValueError(f"Unexpected combination of types: {name_to_type}")


def per_model_weight_count_estimation(tensors: Iterable[tuple[str, LazyTensor]]) -> tuple[int, int, int]:
total_params = 0
shared_params = 0
expert_params = 0

for name, lazy_tensor in tensors:
    # We don't need these
    if name.endswith((".attention.masked_bias", ".attention.bias", ".rotary_emb.inv_freq")):
        continue
    # Got A Tensor
    sum_weights_in_tensor: int = 1

    # Tensor Volume
    for dim in lazy_tensor.shape:
        sum_weights_in_tensor *= dim

    if ".experts." in name:
        if ".experts.0." in name:
                    expert_params += sum_weights_in_tensor
    else:
        shared_params += sum_weights_in_tensor

    total_params += sum_weights_in_tensor

return total_params, shared_params, expert_params


def convert_to_output_type(model: LazyModel, output_type: GGMLFileType) -> LazyModel:
return {name: tensor.astype(output_type.type_for_tensor(name, tensor))
        for (name, tensor) in model.items()}


def convert_model_names(model: LazyModel, params: Params, skip_unknown: bool) -> LazyModel:
tmap = gguf.TensorNameMap(ARCH, params.n_layer)
should_skip = set(gguf.MODEL_TENSOR_SKIP.get(ARCH, []))

tmp = model

# merge experts into one tensor
if params.n_experts and params.n_experts > 0:
    for i_l in range(params.n_layer):
        for w in range(1, 4):
                    experts = []
                    for e in range(params.n_experts):
                        if f"layers.{i_l}.feed_forward.experts.{e}.w{w}.weight" in model:
                            experts.append(model[f"layers.{i_l}.feed_forward.experts.{e}.w{w}.weight"])
                            del tmp[f"layers.{i_l}.feed_forward.experts.{e}.w{w}.weight"]
                        elif f"model.layers.{i_l}.block_sparse_moe.experts.{e}.w{w}.weight" in model:
                            experts.append(model[f"model.layers.{i_l}.block_sparse_moe.experts.{e}.w{w}.weight"])
                            del tmp[f"model.layers.{i_l}.block_sparse_moe.experts.{e}.w{w}.weight"]
                        else:
                            raise ValueError(f"Expert tensor not found: layers.{i_l}.feed_forward.experts.{e}.w{w}.weight")
                    tmp[f"layers.{i_l}.feed_forward.experts.w{w}.weight"] = pack_experts_lazy(experts)

# HF models permut or pack some of the tensors, so we need to undo that
for i in itertools.count():
    if f"model.layers.{i}.self_attn.q_proj.weight" in model:
        logger.debug(f"Permuting layer {i}")
        tmp[f"model.layers.{i}.self_attn.q_proj.weight"] = permute_lazy(model[f"model.layers.{i}.self_attn.q_proj.weight"], params.n_head, params.n_head)
        tmp[f"model.layers.{i}.self_attn.k_proj.weight"] = permute_lazy(model[f"model.layers.{i}.self_attn.k_proj.weight"], params.n_head, params.n_head_kv)
```

```
                           #  tmp[f"model.layers.{i}.self_attn.v_proj.weight"]  =
model[f"model.layers.{i}.self_attn.v_proj.weight"]
        elif f"model.layers.{i}.self_attn.W_pack.weight" in model:
            logger.debug(f"Unpacking and permuting layer {i}")
                                                          tmp[f"model.layers.{i}.self_attn.q_proj.weight"]    =
permute_part_lazy(model[f"model.layers.{i}.self_attn.W_pack.weight"], 0, params.n_head, params.n_head)
                                                          tmp[f"model.layers.{i}.self_attn.k_proj.weight"]    =
permute_part_lazy(model[f"model.layers.{i}.self_attn.W_pack.weight"], 1, params.n_head, params.n_head_kv)
                              tmp[f"model.layers.{i}.self_attn.v_proj.weight"]  =  part_lazy
(model[f"model.layers.{i}.self_attn.W_pack.weight"], 2)
            del tmp[f"model.layers.{i}.self_attn.W_pack.weight"]
        else:
            break

    out: LazyModel = {}
    for name, lazy_tensor in model.items():
          tensor_type, name_new = tmap.get_type_and_name(name, try_suffixes = (".weight", ".bias")) or (None,
None)
        if name_new is None:
            if skip_unknown:
                logger.warning(f"Unexpected tensor name: {name} - skipping")
                continue
            raise ValueError(f"Unexpected tensor name: {name}. Use --skip-unknown to ignore it (e.g. LLaVA)")

        if tensor_type in should_skip:
            logger.debug(f"skipping tensor {name_new}")
            continue

        logger.debug(f"{name:48s} -> {name_new:40s} | {lazy_tensor.data_type.name:6s} | {lazy_tensor.shape}")
        out[name_new] = lazy_tensor

    return out


def nth_multifile_path(path: Path, n: int) -> Path | None:
    '''Given any path belonging to a multi-file model (e.g. foo.bin.1), return
    the nth path in the model.
    '''
    # Support the following patterns:
    patterns = [
        # - x.00.pth, x.01.pth, etc.
        (r'\.[0-9]{2}\.pth$', f'.{n:02}.pth'),
        # - x-00001-of-00002.bin, x-00002-of-00002.bin, etc.
        (r'-[0-9]{5}-of-(.*)$', fr'-{n:05}-of-\1'),
        # x.bin, x.bin.1, etc.
        (r'(\.[0-9]+)?$', r'\1' if n == 0 else fr'\1.{n}')
    ]
    for regex, replacement in patterns:
        if re.search(regex, path.name):
            new_path = path.with_name(re.sub(regex, replacement, path.name))
            if new_path.exists():
                return new_path
    return None
```

```
def find_multifile_paths(path: Path) -> list[Path]:
    '''Given any path belonging to a multi-file model (e.g. foo.bin.1), return
    the whole list of paths in the model.
    '''
    ret: list[Path] = []
    for i in itertools.count():
        nth_path = nth_multifile_path(path, i)
        if nth_path is None:
            break
        ret.append(nth_path)
    if not ret:
        # No matches.  This should only happen if the file was named, e.g.,
        # foo.0, and there was no file named foo.  Oh well, try to process it
        # as a single file.
        return [path]
    return ret


def load_some_model(path: Path) -> ModelPlus:
    '''Load a model of any supported format.'''
    # Be extra-friendly and accept either a file or a directory:
    if path.is_dir():
        # Check if it's a set of safetensors files first
        globs = ["model-00001-of-*.safetensors", "model.safetensors", "consolidated.safetensors"]
        files = [file for glob in globs for file in path.glob(glob)]
        if not files:
            # Try the PyTorch patterns too, with lower priority
            globs = ["consolidated.00.pth", "pytorch_model-00001-of-*.bin", "*.pt", "pytorch_model.bin"]
            files = [file for glob in globs for file in path.glob(glob)]
        if not files:
            raise FileNotFoundError(f"Can't find model in directory {path}")
        if len(files) > 1:
            raise ValueError(f"Found multiple models in {path}, not sure which to pick: {files}")
        path = files[0]

    paths = find_multifile_paths(path)
    models_plus: list[ModelPlus] = []
    for path in paths:
        logger.info(f"Loading model file {path}")
        models_plus.append(lazy_load_file(path))

    model_plus = merge_multifile_models(models_plus)
    return model_plus


class VocabFactory:
    _VOCAB_CLASSES: list[type[Vocab]] = [SentencePieceVocab, BpeVocab, LlamaHfVocab]

    def __init__(self, path: Path):
        self.path = path

    def _create_special_vocab(self, vocab: BaseVocab, model_parent_path: Path) -> gguf.SpecialVocab:
        load_merges = vocab.name == "bpe"
        n_vocab = vocab.vocab_size if isinstance(vocab, Vocab) else None
        return gguf.SpecialVocab(
            model_parent_path,
            load_merges=load_merges,
            special_token_types=None,  # Predetermined or passed as a parameter
            n_vocab=n_vocab,
        )

    def _create_vocab_by_path(self, vocab_types: list[str]) -> Vocab:
        vocab_classes: dict[str, type[Vocab]] = {cls.name: cls for cls in self._VOCAB_CLASSES}
        selected_vocabs: dict[str, type[Vocab]] = {}
        for vtype in vocab_types:
            try:
                selected_vocabs[vtype] = vocab_classes[vtype]
            except KeyError:
                raise ValueError(f"Unsupported vocabulary type {vtype}") from None

        for vtype, cls in selected_vocabs.items():
            try:
                vocab = cls(self.path)
                break
            except FileNotFoundError:
                pass  # ignore unavailable tokenizers
        else:
            raise FileNotFoundError(f"Could not find a tokenizer matching any of {vocab_types}")

        logger.info(f"Loaded vocab file {vocab.fname_tokenizer!r}, type {vocab.name!r}")
        return vocab

    def load_vocab(self, vocab_types: list[str] | None, model_parent_path: Path) -> tuple[BaseVocab,
gguf.SpecialVocab]:
        vocab: BaseVocab
        if vocab_types is None:
            vocab = NoVocab()
        else:
            vocab = self._create_vocab_by_path(vocab_types)
        # FIXME: Respect --vocab-dir?
        special_vocab = self._create_special_vocab(
            vocab,
            model_parent_path,
        )
        return vocab, special_vocab


def default_convention_outfile(file_type: GGMLFileType, expert_count: int | None, model_params_count:
tuple[int, int, int], metadata: gguf.Metadata) -> str:
    name = metadata.name if metadata.name is not None else None
    basename = metadata.basename if metadata.basename is not None else None
    finetune = metadata.finetune if metadata.finetune is not None else None
    version = metadata.version if metadata.version is not None else None
    size_label = metadata.size_label if metadata.size_label is not None else
gguf.size_label(*model_params_count, expert_count=expert_count or 0)

    output_type = {
        GGMLFileType.AllF32:    "F32",
        GGMLFileType.MostlyF16: "F16",
        GGMLFileType.MostlyQ8_0: "Q8_0",
    }[file_type]

    return gguf.naming_convention(name, basename, finetune, version, size_label, output_type)


def default_outfile(model_paths: list[Path], file_type: GGMLFileType, expert_count: int | None, model_params_count: tuple[int, int, int], metadata: gguf.Metadata) -> Path:
    default_filename = default_convention_outfile(file_type, expert_count, model_params_count, metadata)
    ret = model_paths[0].parent / f"{default_filename}.gguf"
    if ret in model_paths:
        logger.error(
            f"Error: Default output path ({ret}) would overwrite the input. "
            "Please explicitly specify a path using --outfile.")
        sys.exit(1)
    return ret


def do_dump_model(model_plus: ModelPlus) -> None:
    print(f"model_plus.paths = {model_plus.paths!r}") # noqa: NP100
    print(f"model_plus.format = {model_plus.format!r}") # noqa: NP100
    print(f"model_plus.vocab = {model_plus.vocab!r}") # noqa: NP100
    for name, lazy_tensor in model_plus.model.items():
        print(f"{name}: shape={lazy_tensor.shape} type={lazy_tensor.data_type}; {lazy_tensor.description}") # noqa: NP100


def main(args_in: list[str] | None = None) -> None:
    output_choices = ["f32", "f16"]
    if np.uint32(1) == np.uint32(1).newbyteorder("<"):
        # We currently only support Q8_0 output on little endian systems.
        output_choices.append("q8_0")
    parser = argparse.ArgumentParser(description="Convert a LLaMA model to a GGML compatible file")
    parser.add_argument("--dump",         action="store_true",    help="don't convert, just show what's in the model")
    parser.add_argument("--dump-single",  action="store_true",    help="don't convert, just show what's in a single model file")
    parser.add_argument("--vocab-only",   action="store_true",    help="extract only the vocab")
    parser.add_argument("--no-vocab",     action="store_true",    help="store model without the vocab")
    parser.add_argument("--outtype",      choices=output_choices, help="output format - note: q8_0 may be very slow (default: f16 or f32 based on input)")
    parser.add_argument("--vocab-dir",    type=Path,              help="directory containing tokenizer.model, if separate from model file")
    parser.add_argument("--vocab-type",                           help="vocab types to try in order, choose from 'spm', 'bpe', 'hfft' (default: spm,hfft)", default="spm,hfft")
    parser.add_argument("--outfile",      type=Path,              help="path to write to; default: based on input")
    parser.add_argument("model",          type=Path,              help="directory containing model file, or model file itself (*.pth, *.pt, *.bin)")
    parser.add_argument("--ctx",          type=int,               help="model training context (default: based on input)")
    parser.add_argument("--concurrency",  type=int,               help=f"concurrency used for conversion
(default: {DEFAULT_CONCURRENCY})", default=DEFAULT_CONCURRENCY)
    parser.add_argument("--big-endian",   action="store_true",    help="model is executed on big endian
machine")
    parser.add_argument("--pad-vocab",    action="store_true",    help="add pad tokens when model vocab expects
more than tokenizer metadata provides")
    parser.add_argument("--skip-unknown", action="store_true",    help="skip unknown tensor names instead of
failing")
    parser.add_argument("--verbose",      action="store_true",    help="increase output verbosity")
    parser.add_argument("--metadata",     type=Path,              help="Specify the path for an authorship
metadata override file")
    parser.add_argument("--get-outfile",  action="store_true",    help="get calculated default outfile name")
    parser.add_argument("--model-name",   type=str, default=None, help="name of the model")

    args = parser.parse_args(args_in)

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG)
    elif args.dump_single or args.dump or args.get_outfile:
        # Avoid printing anything besides the dump output
        logging.basicConfig(level=logging.WARNING)
    else:
        logging.basicConfig(level=logging.INFO)

    model_name = args.model_name
    dir_model = args.model

    metadata = gguf.Metadata.load(args.metadata, dir_model, model_name)

    if args.get_outfile:
        model_plus = load_some_model(dir_model)
        params = Params.load(model_plus)
        model = convert_model_names(model_plus.model, params, args.skip_unknown)
        model_params_count = per_model_weight_count_estimation(model_plus.model.items())
        ftype = pick_output_type(model, args.outtype)

        if (metadata is None or metadata.name is None) and params.path_model is not None:
            metadata.name = params.path_model.name

        print(f"{default_convention_outfile(ftype, params.n_experts, model_params_count, metadata)}") # noqa:
NP100
        return

    if args.no_vocab and args.vocab_only:
        raise ValueError("--vocab-only does not make sense with --no-vocab")

    if args.dump_single:
        model_plus = lazy_load_file(dir_model)
        do_dump_model(model_plus)
        return

    if not args.vocab_only:
        model_plus = load_some_model(dir_model)
    else:
        model_plus = ModelPlus(model = {}, paths = [dir_model / 'dummy'], format = 'none', vocab = None)
```