```
            "type": field.types[0].name if field.types else 'UNKNOWN',
            "offset": field.offset,
        }
        metadata[field.name] = curr
        if field.types[:1] == [GGUFValueType.ARRAY]:
            curr["array_types"] = [t.name for t in field.types][1:]
            if not args.json_array:
                continue
            curr["value"] = field.contents()
        else:
            curr["value"] = field.contents()
    if not args.no_tensors:
        for idx, tensor in enumerate(reader.tensors):
            tensors[tensor.name] = {
                "index": idx,
                "shape": tensor.shape.tolist(),
                "type": tensor.tensor_type.name,
                "offset": tensor.field.offset,
            }
    json.dump(result, sys.stdout)


def markdown_table_with_alignment_support(header_map: list[dict[str, str]], data: list[dict[str, Any]]):
    # JSON to Markdown table formatting: https://stackoverflow.com/a/72983854/2850957

    # Alignment Utility Function
    def strAlign(padding: int, alignMode: str | None, strVal: str):
        if alignMode == 'center':
            return strVal.center(padding)
        elif alignMode == 'right':
            return strVal.rjust(padding - 1) + ' '
        elif alignMode == 'left':
            return ' ' + strVal.ljust(padding - 1)
        else: # default left
            return ' ' + strVal.ljust(padding - 1)

    def dashAlign(padding: int, alignMode: str | None):
        if alignMode == 'center':
            return ':' + '-' * (padding - 2) + ':'
        elif alignMode == 'right':
            return '-' * (padding - 1) + ':'
        elif alignMode == 'left':
            return ':' + '-' * (padding - 1)
        else: # default left
            return '-' * (padding)

    # Calculate Padding For Each Column Based On Header and Data Length
    rowsPadding = {}
    for index, columnEntry in enumerate(header_map):
          padCount = max([len(str(v)) for d in data for k, v in d.items() if k == columnEntry['key_name']],
default=0) + 2
        headerPadCount = len(columnEntry['header_name']) + 2
        rowsPadding[index] = headerPadCount if padCount <= headerPadCount else padCount
```

```
    # Render Markdown Header
    rows = []
                               rows.append('|'.join(strAlign(rowsPadding[index],       columnEntry.get('align'),
str(columnEntry['header_name'])) for index, columnEntry in enumerate(header_map)))
       rows.append('|'.join(dashAlign(rowsPadding[index], columnEntry.get('align')) for index, columnEntry in
enumerate(header_map)))

    # Render Tabular Data
    for item in data:
                                    rows.append('|'.join(strAlign(rowsPadding[index],    columnEntry.get('align'),
str(item[columnEntry['key_name']])) for index, columnEntry in enumerate(header_map)))

    # Convert Tabular String Rows Into String
    tableString = ""
    for row in rows:
        tableString += f'|{row}|\n'

    return tableString


def element_count_rounded_notation(count: int) -> str:
    if count > 1e15 :
        # Quadrillion
        scaled_amount = count * 1e-15
        scale_suffix = "Q"
    elif count > 1e12 :
        # Trillions
        scaled_amount = count * 1e-12
        scale_suffix = "T"
    elif count > 1e9 :
        # Billions
        scaled_amount = count * 1e-9
        scale_suffix = "B"
    elif count > 1e6 :
        # Millions
        scaled_amount = count * 1e-6
        scale_suffix = "M"
    elif count > 1e3 :
        # Thousands
        scaled_amount = count * 1e-3
        scale_suffix = "K"
    else:
        # Under Thousands
        scaled_amount = count
        scale_suffix = ""
    return f"{'~' if count > 1e3 else ''}{round(scaled_amount)}{scale_suffix}"


def translate_tensor_name(name):
    words = name.split(".")

    # Source: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md#standardized-tensor-names
    abbreviation_dictionary = {
        'token_embd': 'Token embedding',
```

```
        'pos_embd': 'Position embedding',
        'output_norm': 'Output normalization',
        'output': 'Output',
        'attn_norm': 'Attention normalization',
        'attn_norm_2': 'Attention normalization',
        'attn_qkv': 'Attention query-key-value',
        'attn_q': 'Attention query',
        'attn_k': 'Attention key',
        'attn_v': 'Attention value',
        'attn_output': 'Attention output',
        'ffn_norm': 'Feed-forward network normalization',
        'ffn_up': 'Feed-forward network "up"',
        'ffn_gate': 'Feed-forward network "gate"',
        'ffn_down': 'Feed-forward network "down"',
        'ffn_gate_inp': 'Expert-routing layer for the Feed-forward network in Mixture of Expert models',
        'ffn_gate_exp': 'Feed-forward network "gate" layer per expert in Mixture of Expert models',
        'ffn_down_exp': 'Feed-forward network "down" layer per expert in Mixture of Expert models',
        'ffn_up_exp': 'Feed-forward network "up" layer per expert in Mixture of Expert models',
        'ssm_in': 'State space model input projections',
        'ssm_conv1d': 'State space model rolling/shift',
        'ssm_x': 'State space model selective parametrization',
        'ssm_a': 'State space model state compression',
        'ssm_d': 'State space model skip connection',
        'ssm_dt': 'State space model time step',
        'ssm_out': 'State space model output projection',
        'blk': 'Block',
        'enc': 'Encoder',
        'dec': 'Decoder',
    }

    expanded_words = []
    for word in words:
        word_norm = word.strip().lower()
        if word_norm in abbreviation_dictionary:
            expanded_words.append(abbreviation_dictionary[word_norm].title())
        else:
            expanded_words.append(word.title())

    return ' '.join(expanded_words)


def dump_markdown_metadata(reader: GGUFReader, args: argparse.Namespace) -> None:
    host_endian, file_endian = get_file_host_endian(reader)
    markdown_content = ""
    markdown_content += f'# {args.model} - GGUF Internal File Dump\n\n'
    markdown_content += f'- Endian: {file_endian} endian\n'
    markdown_content += '\n'
    markdown_content += '## Key Value Metadata Store\n\n'
    markdown_content += f'There are {len(reader.fields)} key-value pairs in this file\n'
    markdown_content += '\n'

    kv_dump_table: list[dict[str, str | int]] = []
    for n, field in enumerate(reader.fields.values(), 1):
        if not field.types:
```

```
            pretty_type = 'N/A'
        elif field.types[0] == GGUFValueType.ARRAY:
            nest_count = len(field.types) - 1
            pretty_type = '[' * nest_count + str(field.types[-1].name) + ']' * nest_count
        else:
            pretty_type = str(field.types[-1].name)

        def escape_markdown_inline_code(value_string):
            # Find the longest contiguous sequence of backticks in the string then
            # wrap string with appropriate number of backticks required to escape it
            max_backticks = max((len(match.group(0)) for match in re.finditer(r'`+', value_string)), default=0)
            inline_code_marker = '`' * (max_backticks + 1)

            # If the string starts or ends with a backtick, add a space at the beginning and end
            if value_string.startswith('`') or value_string.endswith('`'):
                value_string = f" {value_string} "

            return f"{inline_code_marker}{value_string}{inline_code_marker}"

        total_elements = len(field.data)
        value = ""
        if len(field.types) == 1:
            curr_type = field.types[0]
            if curr_type == GGUFValueType.STRING:
                truncate_length = 60
                value_string = str(bytes(field.parts[-1]), encoding='utf-8')
                if len(value_string) > truncate_length:
                    head = escape_markdown_inline_code(value_string[:truncate_length // 2])
                    tail = escape_markdown_inline_code(value_string[-truncate_length // 2:])
                    value = "{head}...{tail}".format(head=head, tail=tail)
                else:
                    value = escape_markdown_inline_code(value_string)
            elif curr_type in reader.gguf_scalar_to_np:
                value = str(field.parts[-1][0])
        else:
            if field.types[0] == GGUFValueType.ARRAY:
                curr_type = field.types[1]
                array_elements = []

                if curr_type == GGUFValueType.STRING:
                    render_element = min(5, total_elements)
                    for element_pos in range(render_element):
                        truncate_length = 30
                           value_string = str(bytes(field.parts[-1 - (total_elements - element_pos - 1) * 2]),
encoding='utf-8')
                        if len(value_string) > truncate_length:
                            head = escape_markdown_inline_code(value_string[:truncate_length // 2])
                            tail = escape_markdown_inline_code(value_string[-truncate_length // 2:])
                            value = "{head}...{tail}".format(head=head, tail=tail)
                        else:
                            value = escape_markdown_inline_code(value_string)
                        array_elements.append(value)

                elif curr_type in reader.gguf_scalar_to_np:
```

```
                    render_element = min(7, total_elements)
                    for element_pos in range(render_element):
                        array_elements.append(str(field.parts[-1 - (total_elements - element_pos - 1)][0]))

                value = f'[ {", ".join(array_elements).strip()}{", ..." if total_elements > len(array_elements) else ""} ]'

                    kv_dump_table.append({"n":n,   "pretty_type":pretty_type,   "total_elements":total_elements, "field_name":field.name, "value":value})

    kv_dump_table_header_map = [
        {'key_name':'n',                'header_name':'POS',      'align':'right'},
        {'key_name':'pretty_type',      'header_name':'TYPE',     'align':'left'},
        {'key_name':'total_elements',   'header_name':'Count',    'align':'right'},
        {'key_name':'field_name',       'header_name':'Key',      'align':'left'},
        {'key_name':'value',            'header_name':'Value',    'align':'left'},
    ]

    markdown_content += markdown_table_with_alignment_support(kv_dump_table_header_map, kv_dump_table)

    markdown_content += "\n"

    if not args.no_tensors:
        # Group tensors by their prefix and maintain order
        tensor_prefix_order: list[str] = []
        tensor_name_to_key: dict[str, int] = {}
        tensor_groups: dict[str, list[ReaderTensor]] = {}
        total_elements = sum(tensor.n_elements for tensor in reader.tensors)

        # Parsing Tensors Record
        for key, tensor in enumerate(reader.tensors):
            tensor_components = tensor.name.split('.')

            # Classify Tensor Group
            tensor_group_name = "base"
            if tensor_components[0] == 'blk':
                tensor_group_name = f"{tensor_components[0]}.{tensor_components[1]}"
            elif tensor_components[0] in ['enc', 'dec'] and tensor_components[1] == 'blk':
                tensor_group_name = f"{tensor_components[0]}.{tensor_components[1]}.{tensor_components[2]}"
            elif tensor_components[0] in ['enc', 'dec']:
                tensor_group_name = f"{tensor_components[0]}"

            # Check if new Tensor Group
            if tensor_group_name not in tensor_groups:
                tensor_groups[tensor_group_name] = []
                tensor_prefix_order.append(tensor_group_name)

            # Record Tensor and Tensor Position
            tensor_groups[tensor_group_name].append(tensor)
            tensor_name_to_key[tensor.name] = key

        # Tensors Mapping Dump
                markdown_content  +=  f'##  Tensors  Overview  {element_count_rounded_notation(total_elements)} Elements\n\n'
```

```
        markdown_content += f'Total number of elements in all tensors: {total_elements} Elements\n'
        markdown_content += '\n'

        for group in tensor_prefix_order:
            tensors = tensor_groups[group]
            group_elements = sum(tensor.n_elements for tensor in tensors)
                                markdown_content  +=  f"-  [{translate_tensor_name(group)}  Tensor  Group  -
{element_count_rounded_notation(group_elements)} Elements](#{group.replace('.', '_')})\n"

        markdown_content += "\n"

        markdown_content += "### Tensor Data Offset\n"
        markdown_content += '\n'
        markdown_content += 'This table contains the offset and data segment relative to start of file\n'
        markdown_content += '\n'

        tensor_mapping_table: list[dict[str, str | int]] = []
        for key, tensor in enumerate(reader.tensors):
            data_offset_pretty = '{0:#16x}'.format(tensor.data_offset)
            data_size_pretty = '{0:#16x}'.format(tensor.n_bytes)
                                        tensor_mapping_table.append({"t_id":key,   "layer_name":tensor.name,
"data_offset":data_offset_pretty, "data_size":data_size_pretty})

        tensors_mapping_table_header_map = [
            {'key_name':'t_id',         'header_name':'T_ID',               'align':'right'},
            {'key_name':'layer_name',   'header_name':'Tensor Layer Name',  'align':'left'},
            {'key_name':'data_offset',  'header_name':'Data Offset (B)',    'align':'right'},
            {'key_name':'data_size',    'header_name':'Data Size (B)',      'align':'right'},
        ]

                 markdown_content  +=  markdown_table_with_alignment_support(tensors_mapping_table_header_map,
tensor_mapping_table)
        markdown_content += "\n"

        for group in tensor_prefix_order:
            tensors = tensor_groups[group]
            group_elements = sum(tensor.n_elements for tensor in tensors)
            group_percentage = group_elements / total_elements * 100
                markdown_content += f"### <a name=\"{group.replace('.', '_')}\">{translate_tensor_name(group)}
Tensor Group : {element_count_rounded_notation(group_elements)} Elements</a>\n\n"

            # Precalculate column sizing for visual consistency
            prettify_element_est_count_size: int = 1
            prettify_element_count_size: int = 1
            prettify_dimension_max_widths: dict[int, int] = {}
            for tensor in tensors:
                                       prettify_element_est_count_size  =  max(prettify_element_est_count_size,
len(str(element_count_rounded_notation(tensor.n_elements))))
                prettify_element_count_size = max(prettify_element_count_size, len(str(tensor.n_elements)))
                for i, dimension_size in enumerate(list(tensor.shape) + [1] * (4 - len(tensor.shape))):
                                prettify_dimension_max_widths[i]  =  max(prettify_dimension_max_widths.get(i,1),
len(str(dimension_size)))

            # Generate Tensor Layer Table Content
```

```
            tensor_dump_table: list[dict[str, str | int]] = []
            for tensor in tensors:
                                human_friendly_name  =  translate_tensor_name(tensor.name.replace(".weight",
".(W)").replace(".bias", ".(B)"))
                    pretty_dimension = ' x '.join(f'{str(d):>{prettify_dimension_max_widths[i]}}' for i, d in
enumerate(list(tensor.shape) + [1] * (4 - len(tensor.shape))))
                                                                                                element_count_est     =
f"({element_count_rounded_notation(tensor.n_elements):>{prettify_element_est_count_size}})"
                                                                element_count_string    =    f"{element_count_est}
{tensor.n_elements:>{prettify_element_count_size}}"
                type_name_string = f"{tensor.tensor_type.name}"
                    tensor_dump_table.append({"t_id":tensor_name_to_key[tensor.name], "layer_name":tensor.name,
"human_layer_name":human_friendly_name,                                        "element_count":element_count_string,
"pretty_dimension":pretty_dimension, "tensor_type":type_name_string})

            tensor_dump_table_header_map = [
                    {'key_name':'t_id',                'header_name':'T_ID',
'align':'right'},
                     {'key_name':'layer_name',          'header_name':'Tensor  Layer  Name',
'align':'left'},
                          {'key_name':'human_layer_name',  'header_name':'Human  Friendly  Tensor  Layer  Name',
'align':'left'},
                     {'key_name':'element_count',     'header_name':'Elements',
'align':'left'},
                     {'key_name':'pretty_dimension',  'header_name':'Shape',
'align':'left'},
                    {'key_name':'tensor_type',        'header_name':'Type',
'align':'left'},
            ]

                      markdown_content  +=  markdown_table_with_alignment_support(tensor_dump_table_header_map,
tensor_dump_table)

            markdown_content += "\n"
                                        markdown_content    +=    f"-    Total    elements    in    {group}:
({element_count_rounded_notation(group_elements):>4}) {group_elements}\n"
            markdown_content += f"- Percentage of total elements: {group_percentage:.2f}%\n"
            markdown_content += "\n\n"

    print(markdown_content)  # noqa: NP100


def main() -> None:
    parser = argparse.ArgumentParser(description="Dump GGUF file metadata")
    parser.add_argument("model",           type=str,            help="GGUF format model filename")
    parser.add_argument("--no-tensors", action="store_true", help="Don't dump tensor metadata")
    parser.add_argument("--json",       action="store_true", help="Produce JSON output")
     parser.add_argument("--json-array", action="store_true", help="Include full array values in JSON output
(long)")
    parser.add_argument("--data-offset",    action="store_true", help="Start of data offset")
    parser.add_argument("--data-alignment", action="store_true", help="Data alignment applied globally to data
field")
    parser.add_argument("--markdown",   action="store_true", help="Produce markdown output")
    parser.add_argument("--verbose",    action="store_true", help="increase output verbosity")
```

```
    args = parser.parse_args(None if len(sys.argv) > 1 else ["--help"])

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

    if not args.json and not args.markdown and not args.data_offset and not args.data_alignment:
        logger.info(f'* Loading: {args.model}')

    reader = GGUFReader(args.model, 'r')

    if args.json:
        dump_metadata_json(reader, args)
    elif args.markdown:
        dump_markdown_metadata(reader, args)
    elif args.data_offset:
        print(reader.data_offset)  # noqa: NP100
    elif args.data_alignment:
        print(reader.alignment)  # noqa: NP100
    else:
        dump_metadata(reader, args)


if __name__ == '__main__':
    main()

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

import uuid
import hashlib

import logging
import argparse
import os
import sys
from pathlib import Path

from tqdm import tqdm

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

from gguf import GGUFReader  # noqa: E402


logger = logging.getLogger("gguf-hash")

# UUID_NAMESPACE_LLAMA_CPP = uuid.uuid5(uuid.NAMESPACE_URL, 'en.wikipedia.org/wiki/Llama.cpp')
UUID_NAMESPACE_LLAMA_CPP = uuid.UUID('ef001206-dadc-5f6d-a15f-3359e577d4e5')


# For more information about what field.parts and field.data represent,
```

```
# please see the comments in the modify_gguf.py example.
def gguf_hash(reader: GGUFReader, filename: str, disable_progress_bar: bool, no_layer: bool) -> None:
    sha1 = hashlib.sha1()
    sha256 = hashlib.sha256()
    uuidv5_sha1 = hashlib.sha1()
    uuidv5_sha1.update(UUID_NAMESPACE_LLAMA_CPP.bytes)

    # Total Weight Calculation For Progress Bar
    total_weights = 0
    for n, tensor in enumerate(reader.tensors, 1):

        # We don't need these
        if tensor.name.endswith((".attention.masked_bias", ".attention.bias", ".rotary_emb.inv_freq")):
            continue

        # Calculate Tensor Volume
        sum_weights_in_tensor = 1
        for dim in tensor.shape:
            sum_weights_in_tensor *= dim
        total_weights += sum_weights_in_tensor

    # Hash Progress Bar
                bar    =    tqdm(desc="Hashing",    total=total_weights,    unit="weights",    unit_scale=True,
disable=disable_progress_bar)

    # Hashing Process
    for tensor in reader.tensors:

        # We don't need these
        if tensor.name.endswith((".attention.masked_bias", ".attention.bias", ".rotary_emb.inv_freq")):
            continue

        # Progressbar
        sum_weights_in_tensor = 1
        for dim in tensor.shape:
            sum_weights_in_tensor *= dim
        bar.update(sum_weights_in_tensor)

        if not no_layer:

            sha1_layer = hashlib.sha1()
            sha1_layer.update(tensor.data.data)
            print("sha1      {0}  {1}:{2}".format(sha1_layer.hexdigest(), filename, tensor.name)) # noqa: NP100

            sha256_layer = hashlib.sha256()
            sha256_layer.update(tensor.data.data)
              print("sha256     {0}   {1}:{2}".format(sha256_layer.hexdigest(), filename, tensor.name)) # noqa:
NP100

        sha1.update(tensor.data.data)
        sha256.update(tensor.data.data)
        uuidv5_sha1.update(tensor.data.data)

    # Flush Hash Progress Bar
```

```
    bar.close()

    # Display Hash Output
    print("sha1      {0}  {1}".format(sha1.hexdigest(), filename)) # noqa: NP100
    print("sha256    {0}  {1}".format(sha256.hexdigest(), filename)) # noqa: NP100
    print("uuid      {0}  {1}".format(uuid.UUID(bytes=uuidv5_sha1.digest()[:16], version=5), filename)) # noqa:
NP100


def main() -> None:
    parser = argparse.ArgumentParser(description="Dump GGUF file metadata")
    parser.add_argument("model",         type=str,            help="GGUF format model filename")
    parser.add_argument("--no-layer",    action="store_true", help="exclude per layer hash")
    parser.add_argument("--verbose",     action="store_true", help="increase output verbosity")
    parser.add_argument("--progressbar", action="store_true", help="enable progressbar")
    args = parser.parse_args(None if len(sys.argv) > 1 else ["--help"])
    logging.basicConfig(level=logging.DEBUG if args.verbose else logging.INFO)
    reader = GGUFReader(args.model, 'r')
    gguf_hash(reader, args.model, not args.progressbar, args.no_layer)


if __name__ == '__main__':
    main()

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

import logging
import argparse
import os
import sys
import json
from pathlib import Path

from tqdm import tqdm
from typing import Any, Sequence, NamedTuple

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

import gguf

logger = logging.getLogger("gguf-new-metadata")


class MetadataDetails(NamedTuple):
    type: gguf.GGUFValueType
    value: Any
    description: str = ''


def get_field_data(reader: gguf.GGUFReader, key: str) -> Any:
```

```
    field = reader.get_field(key)

    return field.contents() if field else None


def find_token(token_list: Sequence[int], token: str) -> Sequence[int]:
    token_ids = [index for index, value in enumerate(token_list) if value == token]

    if len(token_ids) == 0:
        raise LookupError(f'Unable to find "{token}" in token list!')

    return token_ids


def copy_with_new_metadata(reader: gguf.GGUFReader, writer: gguf.GGUFWriter, new_metadata: dict[str, MetadataDetails], remove_metadata: Sequence[str]) -> None:
    for field in reader.fields.values():
        # Suppress virtual fields and fields written by GGUFWriter
        if field.name == gguf.Keys.General.ARCHITECTURE or field.name.startswith('GGUF.'):
            logger.debug(f'Suppressing {field.name}')
            continue

        # Skip old chat templates if we have new ones
        if field.name.startswith(gguf.Keys.Tokenizer.CHAT_TEMPLATE) and gguf.Keys.Tokenizer.CHAT_TEMPLATE in new_metadata:
            logger.debug(f'Skipping {field.name}')
            continue

        if field.name in remove_metadata:
            logger.debug(f'Removing {field.name}')
            continue

        old_val = MetadataDetails(field.types[0], field.contents())
        val = new_metadata.get(field.name, old_val)

        if field.name in new_metadata:
            logger.debug(f'Modifying {field.name}: "{old_val.value}" -> "{val.value}" {val.description}')
            del new_metadata[field.name]
        elif val.value is not None:
            logger.debug(f'Copying {field.name}')

        if val.value is not None:
            writer.add_key_value(field.name, val.value, val.type)

    if gguf.Keys.Tokenizer.CHAT_TEMPLATE in new_metadata:
        logger.debug('Adding chat template(s)')
        writer.add_chat_template(new_metadata[gguf.Keys.Tokenizer.CHAT_TEMPLATE].value)
        del new_metadata[gguf.Keys.Tokenizer.CHAT_TEMPLATE]

    for key, val in new_metadata.items():
        logger.debug(f'Adding {key}: "{val.value}" {val.description}')
        writer.add_key_value(key, val.value, val.type)

    total_bytes = 0
```

```
    for tensor in reader.tensors:
        total_bytes += tensor.n_bytes
               writer.add_tensor_info(tensor.name,  tensor.data.shape,  tensor.data.dtype,  tensor.data.nbytes,
tensor.tensor_type)

    bar = tqdm(desc="Writing", total=total_bytes, unit="byte", unit_scale=True)

    writer.write_header_to_file()
    writer.write_kv_data_to_file()
    writer.write_ti_data_to_file()

    for tensor in reader.tensors:
        writer.write_tensor_data(tensor.data)
        bar.update(tensor.n_bytes)

    writer.close()


def main() -> None:
     tokenizer_metadata = (getattr(gguf.Keys.Tokenizer, n) for n in gguf.Keys.Tokenizer.__dict__.keys() if not
n.startswith('_'))
          token_names  =  dict((n.split('.')[-1][:-len('_token_id')],  n)  for  n  in  tokenizer_metadata  if
n.endswith('_token_id'))

    parser = argparse.ArgumentParser(description="Make a copy of a GGUF file with new metadata")
    parser.add_argument("input",                                          type=Path, help="GGUF format model input
filename")
     parser.add_argument("output",                                             type=Path, help="GGUF format model
output filename")
      parser.add_argument("--general-name",                                          type=str,   help="The models
general.name", metavar='"name"')
      parser.add_argument("--general-description",                                  type=str,   help="The models
general.description", metavar='"Description ..."')
     parser.add_argument("--chat-template",                                  type=str,  help="Chat template string
(or JSON string containing templates)", metavar='"{% ... %} ..."')
    parser.add_argument("--chat-template-config",                         type=Path, help="Config file containing
chat template(s)", metavar='tokenizer_config.json')
      parser.add_argument("--pre-tokenizer",                                         type=str,   help="The models
tokenizer.ggml.pre", metavar='"pre tokenizer"')
    parser.add_argument("--remove-metadata",      action="append",     type=str,  help="Remove metadata (by key
name) from output model", metavar='general.url')
      parser.add_argument("--special-token",           action="append",       type=str,   help="Special token by
value", nargs=2, metavar=(' | '.join(token_names.keys()), '"<token>"'))
     parser.add_argument("--special-token-by-id",  action="append",      type=str,  help="Special token by id",
nargs=2, metavar=(' | '.join(token_names.keys()), '0'))
    parser.add_argument("--force",                action="store_true",            help="Bypass warnings without
confirmation")
      parser.add_argument("--verbose",                 action="store_true",              help="Increase output
verbosity")
    args = parser.parse_args(None if len(sys.argv) > 2 else ["--help"])

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

```
    new_metadata = {}
    remove_metadata = args.remove_metadata or []

    if args.general_name:
        new_metadata[gguf.Keys.General.NAME] = MetadataDetails(gguf.GGUFValueType.STRING, args.general_name)

    if args.general_description:
                    new_metadata[gguf.Keys.General.DESCRIPTION]  =  MetadataDetails(gguf.GGUFValueType.STRING,
args.general_description)

    if args.chat_template:
                new_metadata[gguf.Keys.Tokenizer.CHAT_TEMPLATE]  =  MetadataDetails(gguf.GGUFValueType.STRING,
json.loads(args.chat_template) if args.chat_template.startswith('[') else args.chat_template)

    if args.chat_template_config:
        with open(args.chat_template_config, 'r') as fp:
            config = json.load(fp)
            template = config.get('chat_template')
            if template:
                  new_metadata[gguf.Keys.Tokenizer.CHAT_TEMPLATE] = MetadataDetails(gguf.GGUFValueType.STRING,
template)

    if args.pre_tokenizer:
        new_metadata[gguf.Keys.Tokenizer.PRE] = MetadataDetails(gguf.GGUFValueType.STRING, args.pre_tokenizer)

    if remove_metadata:
        logger.warning('*** Warning *** Warning *** Warning **')
        logger.warning('* Most metadata is required for a fully functional GGUF file,')
        logger.warning('* removing crucial metadata may result in a corrupt output file!')

        if not args.force:
            logger.warning('* Enter exactly YES if you are positive you want to proceed:')
            response = input('YES, I am sure> ')
            if response != 'YES':
                logger.info("You didn't enter YES. Okay then, see ya!")
                sys.exit(0)

    logger.info(f'* Loading: {args.input}')
    reader = gguf.GGUFReader(args.input, 'r')

    arch = get_field_data(reader, gguf.Keys.General.ARCHITECTURE)

    token_list = get_field_data(reader, gguf.Keys.Tokenizer.LIST) or []

    for name, token in args.special_token or []:
        if name not in token_names:
            logger.warning(f'Unknown special token "{name}", ignoring...')
        else:
            ids = find_token(token_list, token)
            new_metadata[token_names[name]] = MetadataDetails(gguf.GGUFValueType.UINT32, ids[0], f'= {token}')

            if len(ids) > 1:
                            logger.warning(f'Multiple  "{token}"  tokens  found,  choosing  ID  {ids[0]},  use
--special-token-by-id if you want another:')
```

```
                logger.warning(', '.join(str(i) for i in ids))

    for name, id_string in args.special_token_by_id or []:
        if name not in token_names:
            logger.warning(f'Unknown special token "{name}", ignoring...')
        elif not id_string.isdecimal():
            raise LookupError(f'Token ID "{id_string}" is not a valid ID!')
        else:
            id_int = int(id_string)

            if id_int >= 0 and id_int < len(token_list):
                    new_metadata[token_names[name]] = MetadataDetails(gguf.GGUFValueType.UINT32, id_int, f'=
{token_list[id_int]}')
            else:
                raise LookupError(f'Token ID {id_int} is not within token list!')

    if os.path.isfile(args.output) and not args.force:
        logger.warning('*** Warning *** Warning *** Warning **')
        logger.warning(f'* The "{args.output}" GGUF file already exists, it will be overwritten!')
        logger.warning('* Enter exactly YES if you are positive you want to proceed:')
        response = input('YES, I am sure> ')
        if response != 'YES':
            logger.info("You didn't enter YES. Okay then, see ya!")
            sys.exit(0)

    logger.info(f'* Writing: {args.output}')
    writer = gguf.GGUFWriter(args.output, arch=arch, endianess=reader.endianess)

    alignment = get_field_data(reader, gguf.Keys.General.ALIGNMENT)
    if alignment is not None:
        logger.debug(f'Setting custom alignment: {alignment}')
        writer.data_alignment = alignment

    copy_with_new_metadata(reader, writer, new_metadata, remove_metadata)


if __name__ == '__main__':
    main()
```

```
==== gguf_prompt_digger.py ====
"""
LOGICSHREDDER :: gguf_prompt_digger.py
Purpose: Extract readable string chunks from .gguf binary model files, convert into belief fragments
"""

import struct
from pathlib import Path
import uuid, yaml, time

MODEL_DIR = Path("models")
FRAG_DIR = Path("fragments/core")
CONSUMED_DIR = MODEL_DIR / "parsed"

MODEL_DIR.mkdir(exist_ok=True)
```

```
FRAG_DIR.mkdir(parents=True, exist_ok=True)
CONSUMED_DIR.mkdir(exist_ok=True)

def write_fragment(claim, source):
    frag = {
        "id": str(uuid.uuid4())[:8],
        "claim": claim,
        "confidence": 0.77,
        "emotion": {},
        "timestamp": int(time.time()),
        "source": source
    }
    path = FRAG_DIR / f"{frag['id']}.yaml"
    with open(path, 'w', encoding='utf-8') as f:
        yaml.safe_dump(frag, f)

def extract_strings_from_gguf(path):
    logic_chunks = []
    try:
        with open(path, 'rb') as f:
            raw = f.read()
            strings = set()
            current = b""
            for byte in raw:
                if 32 <= byte <= 126:
                    current += bytes([byte])
                else:
                    if len(current) >= 20:
                        strings.add(current.decode(errors='ignore'))
                    current = b""
            logic_chunks = sorted(strings)
    except Exception as e:
        print(f"[gguf_digger] ? Failed to extract from {path.name}: {e}")
    return logic_chunks

def run_digger():
    models = list(MODEL_DIR.glob("*.gguf"))
    total = 0
    for model_path in models:
        logic = extract_strings_from_gguf(model_path)
        if logic:
            for line in logic:
                if len(line.split()) > 3 and not line.startswith("ggml"):
                    write_fragment(line.strip(), model_path.name)
            print(f"[gguf_digger] [OK] Extracted {len(logic)} strings from {model_path.name}")
            total += len(logic)
        else:
            print(f"[gguf_digger] WARNING No usable strings found in {model_path.name}")
    print(f"[gguf_digger] INFO Total beliefs extracted: {total}")

if __name__ == "__main__":
    run_digger()

==== gguf_reader.py ====
```

```
#
# GGUF file reading/modification support. For API usage information,
# please see the files scripts/ for some fairly simple examples.
#
from __future__ import annotations

import logging
import os
import sys
from collections import OrderedDict
from typing import Any, Literal, NamedTuple, TypeVar, Union

import numpy as np
import numpy.typing as npt

from .quants import quant_shape_to_byte_shape

if __name__ == "__main__":
    from pathlib import Path

    # Allow running file in package as a script.
    sys.path.insert(0, str(Path(__file__).parent.parent))

from gguf.constants import (
    GGML_QUANT_SIZES,
    GGUF_DEFAULT_ALIGNMENT,
    GGUF_MAGIC,
    GGUF_VERSION,
    GGMLQuantizationType,
    GGUFValueType,
    GGUFEndian,
)

logger = logging.getLogger(__name__)

READER_SUPPORTED_VERSIONS = [2, GGUF_VERSION]



class ReaderField(NamedTuple):
    # Offset to start of this field.
    offset: int

    # Name of the field (not necessarily from file data).
    name: str

    # Data parts. Some types have multiple components, such as strings
    # that consist of a length followed by the string data.
    parts: list[npt.NDArray[Any]] = []

    # Indexes into parts that we can call the actual data. For example
    # an array of strings will be populated with indexes to the actual
    # string data.
    data: list[int] = [-1]
```

```
    types: list[GGUFValueType] = []

    def contents(self, index_or_slice: int | slice = slice(None)) -> Any:
        if self.types:
            to_string = lambda x: str(x.tobytes(), encoding='utf-8') # noqa: E731
            main_type = self.types[0]

            if main_type == GGUFValueType.ARRAY:
                sub_type = self.types[-1]

                if sub_type == GGUFValueType.STRING:
                    indices = self.data[index_or_slice]

                    if isinstance(index_or_slice, int):
                        return to_string(self.parts[indices]) # type: ignore
                    else:
                        return [to_string(self.parts[idx]) for idx in indices] # type: ignore
                else:
                    # FIXME: When/if _get_field_parts() support multi-dimensional arrays, this must do so too

                    # Check if it's unsafe to perform slice optimization on data
                    # if any(True for idx in self.data if len(self.parts[idx]) != 1):
                    #     optim_slice = slice(None)
                    # else:
                    #     optim_slice = index_or_slice
                    #     index_or_slice = slice(None)

                    # if isinstance(optim_slice, int):
                    #     return self.parts[self.data[optim_slice]].tolist()[0]
                    # else:
                                        #         return [pv for idx in self.data[optim_slice] for pv in
self.parts[idx].tolist()][index_or_slice]

                    if isinstance(index_or_slice, int):
                        return self.parts[self.data[index_or_slice]].tolist()[0]
                    else:
                        return [pv for idx in self.data[index_or_slice] for pv in self.parts[idx].tolist()]

            if main_type == GGUFValueType.STRING:
                return to_string(self.parts[-1])
            else:
                return self.parts[-1].tolist()[0]

        return None


class ReaderTensor(NamedTuple):
    name: str
    tensor_type: GGMLQuantizationType
    shape: npt.NDArray[np.uint32]
    n_elements: int
    n_bytes: int
    data_offset: int
    data: npt.NDArray[Any]
```

```
    field: ReaderField


class GGUFReader:
    # I - same as host, S - swapped
    byte_order: Literal['I', 'S'] = 'I'
    alignment: int = GGUF_DEFAULT_ALIGNMENT
    data_offset: int

    # Note: Internal helper, API may change.
    gguf_scalar_to_np: dict[GGUFValueType, type[np.generic]] = {
        GGUFValueType.UINT8:   np.uint8,
        GGUFValueType.INT8:    np.int8,
        GGUFValueType.UINT16:  np.uint16,
        GGUFValueType.INT16:   np.int16,
        GGUFValueType.UINT32:  np.uint32,
        GGUFValueType.INT32:   np.int32,
        GGUFValueType.FLOAT32: np.float32,
        GGUFValueType.UINT64:  np.uint64,
        GGUFValueType.INT64:   np.int64,
        GGUFValueType.FLOAT64: np.float64,
        GGUFValueType.BOOL:    np.bool_,
    }

    def __init__(self, path: os.PathLike[str] | str, mode: Literal['r', 'r+', 'c'] = 'r'):
        self.data = np.memmap(path, mode = mode)
        offs = 0

        # Check for GGUF magic
        if self._get(offs, np.uint32, override_order = '<')[0] != GGUF_MAGIC:
            raise ValueError('GGUF magic invalid')
        offs += 4

        # Check GGUF version
        temp_version = self._get(offs, np.uint32)
        if temp_version[0] & 65535 == 0:
            # If we get 0 here that means it's (probably) a GGUF file created for
            # the opposite byte order of the machine this script is running on.
            self.byte_order = 'S'
            temp_version = temp_version.view(temp_version.dtype.newbyteorder(self.byte_order))
        version = temp_version[0]
        if version not in READER_SUPPORTED_VERSIONS:
            raise ValueError(f'Sorry, file appears to be version {version} which we cannot handle')
        if sys.byteorder == "little":
            # Host is little endian
            host_endian = GGUFEndian.LITTLE
            swapped_endian = GGUFEndian.BIG
        else:
            # Sorry PDP or other weird systems that don't use BE or LE.
            host_endian = GGUFEndian.BIG
            swapped_endian = GGUFEndian.LITTLE
        self.endianess = swapped_endian if self.byte_order == "S" else host_endian
        self.fields: OrderedDict[str, ReaderField] = OrderedDict()
        self.tensors: list[ReaderTensor] = []
```

```
                    offs   +=   self._push_field(ReaderField(offs,   'GGUF.version',   [temp_version],   [0],
[GGUFValueType.UINT32]))

        # Check tensor count and kv count
        temp_counts = self._get(offs, np.uint64, 2)
                  offs  +=  self._push_field(ReaderField(offs,  'GGUF.tensor_count',  [temp_counts[:1]],  [0],
[GGUFValueType.UINT64]))
                    offs   +=   self._push_field(ReaderField(offs,   'GGUF.kv_count',   [temp_counts[1:]],   [0],
[GGUFValueType.UINT64]))
        tensor_count, kv_count = temp_counts
        offs = self._build_fields(offs, kv_count)

        # Build Tensor Info Fields
        offs, tensors_fields = self._build_tensor_info(offs, tensor_count)
        new_align = self.fields.get('general.alignment')
        if new_align is not None:
            if new_align.types != [GGUFValueType.UINT32]:
                raise ValueError('Bad type for general.alignment field')
            self.alignment = new_align.parts[-1][0]
        padding = offs % self.alignment
        if padding != 0:
            offs += self.alignment - padding
        self.data_offset = offs
        self._build_tensors(offs, tensors_fields)

    _DT = TypeVar('_DT', bound = npt.DTypeLike)

    # Fetch a key/value metadata field by key.
    def get_field(self, key: str) -> Union[ReaderField, None]:
        return self.fields.get(key, None)

    # Fetch a tensor from the list by index.
    def get_tensor(self, idx: int) -> ReaderTensor:
        return self.tensors[idx]

    def _get(
         self, offset: int, dtype: npt.DTypeLike, count: int = 1, override_order: None | Literal['I', 'S', '<']
= None,
    ) -> npt.NDArray[Any]:
        count = int(count)
        itemsize = int(np.empty([], dtype = dtype).itemsize)
        end_offs = offset + itemsize * count
        arr = self.data[offset:end_offs].view(dtype=dtype)[:count]
        return arr.view(arr.dtype.newbyteorder(self.byte_order if override_order is None else override_order))

    def _push_field(self, field: ReaderField, skip_sum: bool = False) -> int:
        if field.name in self.fields:
            # TODO: add option to generate error on duplicate keys
            # raise KeyError(f'Duplicate {field.name} already in list at offset {field.offset}')

            logger.warning(f'Duplicate key {field.name} at offset {field.offset}')
            self.fields[field.name + '_{}'.format(field.offset)] = field
        else:
            self.fields[field.name] = field
```

```
    return 0 if skip_sum else sum(int(part.nbytes) for part in field.parts)

def _get_str(self, offset: int) -> tuple[npt.NDArray[np.uint64], npt.NDArray[np.uint8]]:
    slen = self._get(offset, np.uint64)
    return slen, self._get(offset + 8, np.uint8, slen[0])

def _get_field_parts(
    self, orig_offs: int, raw_type: int,
) -> tuple[int, list[npt.NDArray[Any]], list[int], list[GGUFValueType]]:
    offs = orig_offs
    types: list[GGUFValueType] = []
    gtype = GGUFValueType(raw_type)
    types.append(gtype)
    # Handle strings.
    if gtype == GGUFValueType.STRING:
        sparts: list[npt.NDArray[Any]] = list(self._get_str(offs))
        size = sum(int(part.nbytes) for part in sparts)
        return size, sparts, [1], types
    # Check if it's a simple scalar type.
    nptype = self.gguf_scalar_to_np.get(gtype)
    if nptype is not None:
        val = self._get(offs, nptype)
        return int(val.nbytes), [val], [0], types
    # Handle arrays.
    if gtype == GGUFValueType.ARRAY:
        raw_itype = self._get(offs, np.uint32)
        offs += int(raw_itype.nbytes)
        alen = self._get(offs, np.uint64)
        offs += int(alen.nbytes)
        aparts: list[npt.NDArray[Any]] = [raw_itype, alen]
        data_idxs: list[int] = []
        # FIXME: Handle multi-dimensional arrays properly instead of flattening
        for idx in range(alen[0]):
            curr_size, curr_parts, curr_idxs, curr_types = self._get_field_parts(offs, raw_itype[0])
            if idx == 0:
                types += curr_types
            idxs_offs = len(aparts)
            aparts += curr_parts
            data_idxs += (idx + idxs_offs for idx in curr_idxs)
            offs += curr_size
        return offs - orig_offs, aparts, data_idxs, types
    # We can't deal with this one.
    raise ValueError('Unknown/unhandled field type {gtype}')

def _get_tensor_info_field(self, orig_offs: int) -> ReaderField:
    offs = orig_offs

    # Get Tensor Name
    name_len, name_data = self._get_str(offs)
    offs += int(name_len.nbytes + name_data.nbytes)

    # Get Tensor Dimensions Count
    n_dims = self._get(offs, np.uint32)
    offs += int(n_dims.nbytes)
```

```
        # Get Tensor Dimension Array
        dims = self._get(offs, np.uint64, n_dims[0])
        offs += int(dims.nbytes)

        # Get Tensor Encoding Scheme Type
        raw_dtype = self._get(offs, np.uint32)
        offs += int(raw_dtype.nbytes)

        # Get Tensor Offset
        offset_tensor = self._get(offs, np.uint64)
        offs += int(offset_tensor.nbytes)

        return ReaderField(
            orig_offs,
            str(bytes(name_data), encoding = 'utf-8'),
            [name_len, name_data, n_dims, dims, raw_dtype, offset_tensor],
            [1, 3, 4, 5],
        )

    def _build_fields(self, offs: int, count: int) -> int:
        for _ in range(count):
            orig_offs = offs
            kv_klen, kv_kdata = self._get_str(offs)
            offs += int(kv_klen.nbytes + kv_kdata.nbytes)
            raw_kv_type = self._get(offs, np.uint32)
            offs += int(raw_kv_type.nbytes)
            parts: list[npt.NDArray[Any]] = [kv_klen, kv_kdata, raw_kv_type]
            idxs_offs = len(parts)
            field_size, field_parts, field_idxs, field_types = self._get_field_parts(offs, raw_kv_type[0])
            parts += field_parts
            self._push_field(ReaderField(
                orig_offs,
                str(bytes(kv_kdata), encoding = 'utf-8'),
                parts,
                [idx + idxs_offs for idx in field_idxs],
                field_types,
            ), skip_sum = True)
            offs += field_size
        return offs

    def _build_tensor_info(self, offs: int, count: int) -> tuple[int, list[ReaderField]]:
        tensor_fields = []
        for _ in range(count):
            field = self._get_tensor_info_field(offs)
            offs += sum(int(part.nbytes) for part in field.parts)
            tensor_fields.append(field)
        return offs, tensor_fields

    def _build_tensors(self, start_offs: int, fields: list[ReaderField]) -> None:
        tensors = []
        tensor_names = set() # keep track of name to prevent duplicated tensors
        for field in fields:
            _name_len, name_data, _n_dims, dims, raw_dtype, offset_tensor = field.parts
```

```
            # check if there's any tensor having same name already in the list
            tensor_name = str(bytes(name_data), encoding = 'utf-8')
            if tensor_name in tensor_names:
                raise ValueError(f'Found duplicated tensor with name {tensor_name}')
            tensor_names.add(tensor_name)
            ggml_type = GGMLQuantizationType(raw_dtype[0])
            n_elems = int(np.prod(dims))
            np_dims = tuple(reversed(dims.tolist()))
            block_size, type_size = GGML_QUANT_SIZES[ggml_type]
            n_bytes = n_elems * type_size // block_size
            data_offs = int(start_offs + offset_tensor[0])
            item_type: npt.DTypeLike
            if ggml_type == GGMLQuantizationType.F16:
                item_count = n_elems
                item_type = np.float16
            elif ggml_type == GGMLQuantizationType.F32:
                item_count = n_elems
                item_type = np.float32
            elif ggml_type == GGMLQuantizationType.F64:
                item_count = n_elems
                item_type = np.float64
            elif ggml_type == GGMLQuantizationType.I8:
                item_count = n_elems
                item_type = np.int8
            elif ggml_type == GGMLQuantizationType.I16:
                item_count = n_elems
                item_type = np.int16
            elif ggml_type == GGMLQuantizationType.I32:
                item_count = n_elems
                item_type = np.int32
            elif ggml_type == GGMLQuantizationType.I64:
                item_count = n_elems
                item_type = np.int64
            else:
                item_count = n_bytes
                item_type = np.uint8
                np_dims = quant_shape_to_byte_shape(np_dims, ggml_type)
            tensors.append(ReaderTensor(
                name = tensor_name,
                tensor_type = ggml_type,
                shape = dims,
                n_elements = n_elems,
                n_bytes = n_bytes,
                data_offset = data_offs,
                data = self._get(data_offs, item_type, item_count).reshape(np_dims),
                field = field,
            ))
        self.tensors = tensors

==== gguf_set_metadata.py ====
#!/usr/bin/env python3
import logging
import argparse
import os
```

```
import sys
from pathlib import Path

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

from gguf import GGUFReader  # noqa: E402

logger = logging.getLogger("gguf-set-metadata")


def minimal_example(filename: str) -> None:
    reader = GGUFReader(filename, 'r+')
    field = reader.fields['tokenizer.ggml.bos_token_id']
    if field is None:
        return
    part_index = field.data[0]
    field.parts[part_index][0] = 2  # Set tokenizer.ggml.bos_token_id to 2
    #
    # So what's this field.data thing? It's helpful because field.parts contains
    # _every_ part of the GGUF field. For example, tokenizer.ggml.bos_token_id consists
    # of:
    #
    #  Part index 0: Key length (27)
    #  Part index 1: Key data ("tokenizer.ggml.bos_token_id")
    #  Part index 2: Field type (4, the id for GGUFValueType.UINT32)
    #  Part index 3: Field value
    #
    # Note also that each part is an NDArray slice, so even a part that
    # is only a single value like the key length will be a NDArray of
    # the key length type (numpy.uint32).
    #
    # The .data attribute in the Field is a list of relevant part indexes
    # and doesn't contain internal GGUF details like the key length part.
    # In this case, .data will be [3] - just the part index of the
    # field value itself.


def set_metadata(reader: GGUFReader, args: argparse.Namespace) -> None:
    field = reader.get_field(args.key)
    if field is None:
        logger.error(f'! Field {repr(args.key)} not found')
        sys.exit(1)
    # Note that field.types is a list of types. This is because the GGUF
    # format supports arrays. For example, an array of UINT32 would
    # look like [GGUFValueType.ARRAY, GGUFValueType.UINT32]
    handler = reader.gguf_scalar_to_np.get(field.types[0]) if field.types else None
    if handler is None:
        logger.error(f'! This tool only supports changing simple values, {repr(args.key)} has unsupported type
{field.types}')
        sys.exit(1)
    current_value = field.parts[field.data[0]][0]
    new_value = handler(args.value)
```

```
    logger.info(f'* Preparing to change field {repr(args.key)} from {current_value} to {new_value}')
    if current_value == new_value:
        logger.info(f'- Key {repr(args.key)} already set to requested value {current_value}')
        sys.exit(0)
    if args.dry_run:
        sys.exit(0)
    if not args.force:
        logger.warning('*** Warning *** Warning *** Warning **')
        logger.warning('* Changing fields in a GGUF file can make it unusable. Proceed at your own risk.')
        logger.warning('* Enter exactly YES if you are positive you want to proceed:')
        response = input('YES, I am sure> ')
        if response != 'YES':
            logger.info("You didn't enter YES. Okay then, see ya!")
            sys.exit(0)
    field.parts[field.data[0]][0] = new_value
    logger.info('* Field changed. Successful completion.')


def main() -> None:
    parser = argparse.ArgumentParser(description="Set a simple value in GGUF file metadata")
    parser.add_argument("model",     type=str,            help="GGUF format model filename")
    parser.add_argument("key",       type=str,            help="Metadata key to set")
    parser.add_argument("value",     type=str,            help="Metadata value to set")
    parser.add_argument("--dry-run", action="store_true", help="Don't actually change anything")
    parser.add_argument("--force",   action="store_true", help="Change the field without confirmation")
    parser.add_argument("--verbose",      action="store_true",    help="increase output verbosity")

    args = parser.parse_args(None if len(sys.argv) > 1 else ["--help"])

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

    logger.info(f'* Loading: {args.model}')
    reader = GGUFReader(args.model, 'r' if args.dry_run else 'r+')
    set_metadata(reader, args)


if __name__ == '__main__':
    main()

==== gguf_writer.py ====
from __future__ import annotations

import logging
import os
import shutil
import struct
import tempfile
from dataclasses import dataclass
from enum import Enum, auto
from math import prod
from pathlib import Path
from io import BufferedWriter
from typing import IO, Any, Sequence, Mapping
from string import ascii_letters, digits
```

```
import numpy as np

from .constants import (
    GGUF_DEFAULT_ALIGNMENT,
    GGUF_MAGIC,
    GGUF_VERSION,
    GGMLQuantizationType,
    GGUFEndian,
    GGUFValueType,
    Keys,
    RopeScalingType,
    PoolingType,
    TokenType,
    ExpertGatingFuncType,
)

from .quants import quant_shape_from_byte_shape

logger = logging.getLogger(__name__)


SHARD_NAME_FORMAT = "{:s}-{:05d}-of-{:05d}.gguf"


@dataclass
class TensorInfo:
    shape: Sequence[int]
    dtype: GGMLQuantizationType
    nbytes: int
    tensor: np.ndarray[Any, Any] | None = None


@dataclass
class GGUFValue:
    value: Any
    type: GGUFValueType


class WriterState(Enum):
    NO_FILE = auto()
    EMPTY   = auto()
    HEADER  = auto()
    KV_DATA = auto()
    TI_DATA = auto()
    WEIGHTS = auto()


class GGUFWriter:
    fout: list[BufferedWriter] | None
    path: Path | None
    temp_file: tempfile.SpooledTemporaryFile[bytes] | None
    tensors: list[dict[str, TensorInfo]]
    kv_data: list[dict[str, GGUFValue]]
```

```
    state: WriterState
    _simple_value_packing = {
        GGUFValueType.UINT8:   "B",
        GGUFValueType.INT8:    "b",
        GGUFValueType.UINT16:  "H",
        GGUFValueType.INT16:   "h",
        GGUFValueType.UINT32:  "I",
        GGUFValueType.INT32:   "i",
        GGUFValueType.FLOAT32: "f",
        GGUFValueType.UINT64:  "Q",
        GGUFValueType.INT64:   "q",
        GGUFValueType.FLOAT64: "d",
        GGUFValueType.BOOL:    "?",
    }

    def __init__(
            self, path: os.PathLike[str] | str | None, arch: str, use_temp_file: bool = False, endianess:
GGUFEndian = GGUFEndian.LITTLE,
         split_max_tensors: int = 0, split_max_size: int = 0, dry_run: bool = False, small_first_shard: bool =
False
    ):
        self.fout = None
        self.path = Path(path) if path else None
        self.arch = arch
        self.endianess = endianess
        self.data_alignment = GGUF_DEFAULT_ALIGNMENT
        self.use_temp_file = use_temp_file
        self.temp_file = None
        self.tensors = [{}]
        self.kv_data = [{}]
        self.split_max_tensors = split_max_tensors
        self.split_max_size = split_max_size
        self.dry_run = dry_run
        self.small_first_shard = small_first_shard
        logger.info("gguf: This GGUF file is for {0} Endian only".format(
            "Big" if self.endianess == GGUFEndian.BIG else "Little",
        ))
        self.state = WriterState.NO_FILE

        if self.small_first_shard:
            self.tensors.append({})

        self.add_architecture()

    def get_total_parameter_count(self) -> tuple[int, int, int, int]:
        total_params = 0
        shared_params = 0
        expert_params = 0

        expert_sum = 0
        n_expert_tensors = 0

        last_lora_a: tuple[str, TensorInfo] | None = None
```

```
        for tensors in self.tensors:
            for name, info in tensors.items():

                shape = info.shape

                if name.endswith(".lora_a"):
                    last_lora_a = (name, info)
                    continue
                elif name.endswith(".lora_b"):
                    if last_lora_a is None or last_lora_a[0] != name[:-1] + "a":
                        # Bail when the LoRA pair can't be found trivially
                        logger.warning("can't measure LoRA size correctly, tensor order is unusual")
                        return 0, 0, 0, 0
                    else:
                        shape = (*shape[:-1], last_lora_a[1].shape[-1])

                size = prod(shape)

                if "_exps." in name:
                    expert_params += (size // shape[-3])
                    expert_sum += shape[-3]
                    n_expert_tensors += 1
                else:
                    shared_params += size

                total_params += size

        # Hopefully this should work even for variable-expert-count models
        expert_count = (expert_sum // n_expert_tensors) if n_expert_tensors > 0 else 0

        # Negate the total to signal it's likely not exact
        if last_lora_a is not None:
            total_params = -total_params

        # NOTE: keep the output in the same order as accepted by 'size_label' in gguf-py/gguf/utility.py
        return total_params, shared_params, expert_params, expert_count

    def format_shard_names(self, path: Path) -> list[Path]:
        if len(self.tensors) == 1:
            return [path]
        return [path.with_name(SHARD_NAME_FORMAT.format(path.stem, i + 1, len(self.tensors))) for i in range(len(self.tensors))]

    def open_output_file(self, path: Path | None = None) -> None:
        if self.state is WriterState.EMPTY and self.fout is not None and (path is None or path == self.path):
            # allow calling this multiple times as long as the path is the same
            return

        if self.state is not WriterState.NO_FILE:
            raise ValueError(f'Expected output file to be not yet opened, got {self.state}')

        if path is not None:
            self.path = path
```

```
        if self.path is not None:
            filenames = self.print_plan()
            self.fout = [open(filename, "wb") for filename in filenames]
            self.state = WriterState.EMPTY

    def print_plan(self) -> list[Path]:
        logger.info("Writing the following files:")
        assert self.path is not None
        filenames = self.format_shard_names(self.path)
        assert len(filenames) == len(self.tensors)
        for name, tensors in zip(filenames, self.tensors):
                                    logger.info(f"{name}:   n_tensors   =   {len(tensors)},   total_size   =
{GGUFWriter.format_n_bytes_to_str(sum(ti.nbytes for ti in tensors.values()))}")

        if self.dry_run:
            logger.info("Dry run, not writing files")
            for name in filenames:
                print(name)  # noqa: NP100
            exit()

        return filenames

    def add_shard_kv_data(self) -> None:
        if len(self.tensors) == 1:
            return

        total_tensors = sum(len(t) for t in self.tensors)
        assert self.fout is not None
        total_splits = len(self.fout)
        self.kv_data.extend({} for _ in range(len(self.kv_data), total_splits))
        for i, kv_data in enumerate(self.kv_data):
            kv_data[Keys.Split.LLM_KV_SPLIT_NO] = GGUFValue(i, GGUFValueType.UINT16)
            kv_data[Keys.Split.LLM_KV_SPLIT_COUNT] = GGUFValue(total_splits, GGUFValueType.UINT16)
            kv_data[Keys.Split.LLM_KV_SPLIT_TENSORS_COUNT] = GGUFValue(total_tensors, GGUFValueType.INT32)

    def write_header_to_file(self, path: Path | None = None) -> None:
        if len(self.tensors) == 1 and (self.split_max_tensors != 0 or self.split_max_size != 0):
            logger.warning("Model fails split requirements, not splitting")

        self.open_output_file(path)

        if self.state is not WriterState.EMPTY:
            raise ValueError(f'Expected output file to be empty, got {self.state}')

        assert self.fout is not None
        assert len(self.fout) == len(self.tensors)
        assert len(self.kv_data) == 1

        self.add_shard_kv_data()

        for fout, tensors, kv_data in zip(self.fout, self.tensors, self.kv_data):
            fout.write(self._pack("<I", GGUF_MAGIC, skip_pack_prefix = True))
            fout.write(self._pack("I", GGUF_VERSION))
            fout.write(self._pack("Q", len(tensors)))
```

```
            fout.write(self._pack("Q", len(kv_data)))
            fout.flush()
        self.state = WriterState.HEADER

    def write_kv_data_to_file(self) -> None:
        if self.state is not WriterState.HEADER:
            raise ValueError(f'Expected output file to contain the header, got {self.state}')
        assert self.fout is not None

        for fout, kv_data in zip(self.fout, self.kv_data):
            kv_bytes = bytearray()

            for key, val in kv_data.items():
                kv_bytes += self._pack_val(key, GGUFValueType.STRING, add_vtype=False)
                kv_bytes += self._pack_val(val.value, val.type, add_vtype=True)

            fout.write(kv_bytes)

        self.flush()
        self.state = WriterState.KV_DATA

    def write_ti_data_to_file(self) -> None:
        if self.state is not WriterState.KV_DATA:
            raise ValueError(f'Expected output file to contain KV data, got {self.state}')
        assert self.fout is not None

        for fout, tensors in zip(self.fout, self.tensors):
            ti_data = bytearray()
            offset_tensor = 0

            for name, ti in tensors.items():
                ti_data += self._pack_val(name, GGUFValueType.STRING, add_vtype=False)
                n_dims = len(ti.shape)
                ti_data += self._pack("I", n_dims)
                for j in range(n_dims):
                    ti_data += self._pack("Q", ti.shape[n_dims - 1 - j])
                ti_data += self._pack("I", ti.dtype)
                ti_data += self._pack("Q", offset_tensor)
                offset_tensor += GGUFWriter.ggml_pad(ti.nbytes, self.data_alignment)

            fout.write(ti_data)
            fout.flush()
        self.state = WriterState.TI_DATA

    def add_key_value(self, key: str, val: Any, vtype: GGUFValueType) -> None:
        if any(key in kv_data for kv_data in self.kv_data):
            raise ValueError(f'Duplicated key name {key!r}')

        self.kv_data[0][key] = GGUFValue(value=val, type=vtype)

    def add_uint8(self, key: str, val: int) -> None:
        self.add_key_value(key,val, GGUFValueType.UINT8)

    def add_int8(self, key: str, val: int) -> None:
```

```
    self.add_key_value(key, val, GGUFValueType.INT8)

def add_uint16(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.UINT16)

def add_int16(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.INT16)

def add_uint32(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.UINT32)

def add_int32(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.INT32)

def add_float32(self, key: str, val: float) -> None:
    self.add_key_value(key, val, GGUFValueType.FLOAT32)

def add_uint64(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.UINT64)

def add_int64(self, key: str, val: int) -> None:
    self.add_key_value(key, val, GGUFValueType.INT64)

def add_float64(self, key: str, val: float) -> None:
    self.add_key_value(key, val, GGUFValueType.FLOAT64)

def add_bool(self, key: str, val: bool) -> None:
    self.add_key_value(key, val, GGUFValueType.BOOL)

def add_string(self, key: str, val: str) -> None:
    if not val:
        return
    self.add_key_value(key, val, GGUFValueType.STRING)

def add_array(self, key: str, val: Sequence[Any]) -> None:
    if len(val) == 0:
        return
    self.add_key_value(key, val, GGUFValueType.ARRAY)

@staticmethod
def ggml_pad(x: int, n: int) -> int:
    return ((x + n - 1) // n) * n

def add_tensor_info(
    self, name: str, tensor_shape: Sequence[int], tensor_dtype: np.dtype,
    tensor_nbytes: int, raw_dtype: GGMLQuantizationType | None = None,
) -> None:
    if self.state is not WriterState.NO_FILE:
        raise ValueError(f'Expected output file to be not yet opened, got {self.state}')

    if any(name in tensors for tensors in self.tensors):
        raise ValueError(f'Duplicated tensor name {name!r}')

    if raw_dtype is None:
```

```
            if tensor_dtype == np.float16:
                dtype = GGMLQuantizationType.F16
            elif tensor_dtype == np.float32:
                dtype = GGMLQuantizationType.F32
            elif tensor_dtype == np.float64:
                dtype = GGMLQuantizationType.F64
            elif tensor_dtype == np.int8:
                dtype = GGMLQuantizationType.I8
            elif tensor_dtype == np.int16:
                dtype = GGMLQuantizationType.I16
            elif tensor_dtype == np.int32:
                dtype = GGMLQuantizationType.I32
            elif tensor_dtype == np.int64:
                dtype = GGMLQuantizationType.I64
            else:
                raise ValueError("Only F16, F32, F64, I8, I16, I32, I64 tensors are supported for now")
        else:
            dtype = raw_dtype
            if tensor_dtype == np.uint8:
                tensor_shape = quant_shape_from_byte_shape(tensor_shape, raw_dtype)

        # make sure there is at least one tensor before splitting
        if len(self.tensors[-1]) > 0:
            if (  # split when over tensor limit
                self.split_max_tensors != 0
                and len(self.tensors[-1]) >= self.split_max_tensors
            ) or (   # split when over size limit
                self.split_max_size != 0
                and sum(ti.nbytes for ti in self.tensors[-1].values()) + tensor_nbytes > self.split_max_size
            ):
                self.tensors.append({})

        self.tensors[-1][name] = TensorInfo(shape=tensor_shape, dtype=dtype, nbytes=tensor_nbytes)

    def add_tensor(
        self, name: str, tensor: np.ndarray[Any, Any], raw_shape: Sequence[int] | None = None,
        raw_dtype: GGMLQuantizationType | None = None,
    ) -> None:
        if self.endianess == GGUFEndian.BIG:
            tensor.byteswap(inplace=True)
        if self.use_temp_file and self.temp_file is None:
            fp = tempfile.SpooledTemporaryFile(mode="w+b", max_size=256 * 1024 * 1024)
            fp.seek(0)
            self.temp_file = fp

        shape: Sequence[int] = raw_shape if raw_shape is not None else tensor.shape
        self.add_tensor_info(name, shape, tensor.dtype, tensor.nbytes, raw_dtype=raw_dtype)

        if self.temp_file is None:
            self.tensors[-1][name].tensor = tensor
            return

        tensor.tofile(self.temp_file)
        self.write_padding(self.temp_file, tensor.nbytes)
```

```
    def write_padding(self, fp: IO[bytes], n: int, align: int | None = None) -> None:
        pad = GGUFWriter.ggml_pad(n, align if align is not None else self.data_alignment) - n
        if pad != 0:
            fp.write(bytes([0] * pad))

    def write_tensor_data(self, tensor: np.ndarray[Any, Any]) -> None:
        if self.state is not WriterState.TI_DATA and self.state is not WriterState.WEIGHTS:
            raise ValueError(f'Expected output file to contain tensor info or weights, got {self.state}')
        assert self.fout is not None

        if self.endianess == GGUFEndian.BIG:
            tensor.byteswap(inplace=True)

        file_id = -1
        for i, tensors in enumerate(self.tensors):
            if len(tensors) > 0:
                file_id = i
                break

        fout = self.fout[file_id]

        # pop the first tensor info
        # TODO: cleaner way to get the first key
        first_tensor_name = [name for name, _ in zip(self.tensors[file_id].keys(), range(1))][0]
        ti = self.tensors[file_id].pop(first_tensor_name)
        assert ti.nbytes == tensor.nbytes

        self.write_padding(fout, fout.tell())
        tensor.tofile(fout)
        self.write_padding(fout, tensor.nbytes)

        self.state = WriterState.WEIGHTS

    def write_tensors_to_file(self, *, progress: bool = False) -> None:
        self.write_ti_data_to_file()

        assert self.fout is not None

        for fout in self.fout:
            self.write_padding(fout, fout.tell())

        if self.temp_file is None:
            shard_bar = None
            bar = None

            if progress:
                from tqdm import tqdm

                total_bytes = sum(ti.nbytes for t in self.tensors for ti in t.values())

                if len(self.fout) > 1:
                            shard_bar = tqdm(desc=f"Shard (0/{len(self.fout)})", total=None, unit="byte",
unit_scale=True)
```

```
            bar = tqdm(desc="Writing", total=total_bytes, unit="byte", unit_scale=True)

        for i, (fout, tensors) in enumerate(zip(self.fout, self.tensors)):
            if shard_bar is not None:
                shard_bar.set_description(f"Shard ({i + 1}/{len(self.fout)})")
                total = sum(ti.nbytes for ti in tensors.values())
                shard_bar.reset(total=(total if total > 0 else None))

            # relying on the fact that Python dicts preserve insertion order (since 3.7)
            for ti in tensors.values():
                assert ti.tensor is not None  # can only iterate once over the tensors
                assert ti.tensor.nbytes == ti.nbytes
                ti.tensor.tofile(fout)
                if shard_bar is not None:
                    shard_bar.update(ti.nbytes)
                if bar is not None:
                    bar.update(ti.nbytes)
                self.write_padding(fout, ti.nbytes)
                ti.tensor = None
    else:
        self.temp_file.seek(0)

        shutil.copyfileobj(self.temp_file, self.fout[0 if not self.small_first_shard else 1])
        self.flush()
        self.temp_file.close()

    self.state = WriterState.WEIGHTS

def flush(self) -> None:
    assert self.fout is not None
    for fout in self.fout:
        fout.flush()

def close(self) -> None:
    if self.fout is not None:
        for fout in self.fout:
            fout.close()
        self.fout = None

def add_type(self, type_name: str) -> None:
    self.add_string(Keys.General.TYPE, type_name)

def add_architecture(self) -> None:
    self.add_string(Keys.General.ARCHITECTURE, self.arch)

def add_quantization_version(self, quantization_version: int) -> None:
    self.add_uint32(Keys.General.QUANTIZATION_VERSION, quantization_version)

def add_custom_alignment(self, alignment: int) -> None:
    self.data_alignment = alignment
    self.add_uint32(Keys.General.ALIGNMENT, alignment)

def add_file_type(self, ftype: int) -> None:
    self.add_uint32(Keys.General.FILE_TYPE, ftype)
```

```
def add_name(self, name: str) -> None:
    self.add_string(Keys.General.NAME, name)

def add_author(self, author: str) -> None:
    self.add_string(Keys.General.AUTHOR, author)

def add_version(self, version: str) -> None:
    self.add_string(Keys.General.VERSION, version)

def add_organization(self, organization: str) -> None:
    self.add_string(Keys.General.ORGANIZATION, organization)

def add_finetune(self, finetune: str) -> None:
    self.add_string(Keys.General.FINETUNE, finetune)

def add_basename(self, basename: str) -> None:
    self.add_string(Keys.General.BASENAME, basename)

def add_description(self, description: str) -> None:
    self.add_string(Keys.General.DESCRIPTION, description)

def add_quantized_by(self, quantized: str) -> None:
    self.add_string(Keys.General.QUANTIZED_BY, quantized)

def add_size_label(self, size_label: str) -> None:
    self.add_string(Keys.General.SIZE_LABEL, size_label)

def add_license(self, license: str) -> None:
    self.add_string(Keys.General.LICENSE, license)

def add_license_name(self, license: str) -> None:
    self.add_string(Keys.General.LICENSE_NAME, license)

def add_license_link(self, license: str) -> None:
    self.add_string(Keys.General.LICENSE_LINK, license)

def add_url(self, url: str) -> None:
    self.add_string(Keys.General.URL, url)

def add_doi(self, doi: str) -> None:
    self.add_string(Keys.General.DOI, doi)

def add_uuid(self, uuid: str) -> None:
    self.add_string(Keys.General.UUID, uuid)

def add_repo_url(self, repo_url: str) -> None:
    self.add_string(Keys.General.REPO_URL, repo_url)

def add_source_url(self, url: str) -> None:
    self.add_string(Keys.General.SOURCE_URL, url)

def add_source_doi(self, doi: str) -> None:
    self.add_string(Keys.General.SOURCE_DOI, doi)
```

```
def add_source_uuid(self, uuid: str) -> None:
    self.add_string(Keys.General.SOURCE_UUID, uuid)

def add_source_repo_url(self, repo_url: str) -> None:
    self.add_string(Keys.General.SOURCE_REPO_URL, repo_url)

def add_base_model_count(self, source_count: int) -> None:
    self.add_uint32(Keys.General.BASE_MODEL_COUNT, source_count)

def add_base_model_name(self, source_id: int, name: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_NAME.format(id=source_id), name)

def add_base_model_author(self, source_id: int, author: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_AUTHOR.format(id=source_id), author)

def add_base_model_version(self, source_id: int, version: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_VERSION.format(id=source_id), version)

def add_base_model_organization(self, source_id: int, organization: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_ORGANIZATION.format(id=source_id), organization)

def add_base_model_description(self, source_id: int, description: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_DESCRIPTION.format(id=source_id), description)

def add_base_model_url(self, source_id: int, url: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_URL.format(id=source_id), url)

def add_base_model_doi(self, source_id: int, doi: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_DOI.format(id=source_id), doi)

def add_base_model_uuid(self, source_id: int, uuid: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_UUID.format(id=source_id), uuid)

def add_base_model_repo_url(self, source_id: int, repo_url: str) -> None:
    self.add_string(Keys.General.BASE_MODEL_REPO_URL.format(id=source_id), repo_url)

def add_dataset_count(self, source_count: int) -> None:
    self.add_uint32(Keys.General.DATASET_COUNT, source_count)

def add_dataset_name(self, source_id: int, name: str) -> None:
    self.add_string(Keys.General.DATASET_NAME.format(id=source_id), name)

def add_dataset_author(self, source_id: int, author: str) -> None:
    self.add_string(Keys.General.DATASET_AUTHOR.format(id=source_id), author)

def add_dataset_version(self, source_id: int, version: str) -> None:
    self.add_string(Keys.General.DATASET_VERSION.format(id=source_id), version)

def add_dataset_organization(self, source_id: int, organization: str) -> None:
    self.add_string(Keys.General.DATASET_ORGANIZATION.format(id=source_id), organization)

def add_dataset_description(self, source_id: int, description: str) -> None:
    self.add_string(Keys.General.DATASET_DESCRIPTION.format(id=source_id), description)
```

```
def add_dataset_url(self, source_id: int, url: str) -> None:
    self.add_string(Keys.General.DATASET_URL.format(id=source_id), url)

def add_dataset_doi(self, source_id: int, doi: str) -> None:
    self.add_string(Keys.General.DATASET_DOI.format(id=source_id), doi)

def add_dataset_uuid(self, source_id: int, uuid: str) -> None:
    self.add_string(Keys.General.DATASET_UUID.format(id=source_id), uuid)

def add_dataset_repo_url(self, source_id: int, repo_url: str) -> None:
    self.add_string(Keys.General.DATASET_REPO_URL.format(id=source_id), repo_url)

def add_tags(self, tags: Sequence[str]) -> None:
    self.add_array(Keys.General.TAGS, tags)

def add_languages(self, languages: Sequence[str]) -> None:
    self.add_array(Keys.General.LANGUAGES, languages)

def add_tensor_data_layout(self, layout: str) -> None:
    self.add_string(Keys.LLM.TENSOR_DATA_LAYOUT.format(arch=self.arch), layout)

def add_vocab_size(self, size: int) -> None:
    self.add_uint32(Keys.LLM.VOCAB_SIZE.format(arch=self.arch), size)

def add_context_length(self, length: int) -> None:
    self.add_uint32(Keys.LLM.CONTEXT_LENGTH.format(arch=self.arch), length)

def add_embedding_length(self, length: int) -> None:
    self.add_uint32(Keys.LLM.EMBEDDING_LENGTH.format(arch=self.arch), length)

def add_features_length(self, length: int) -> None:
    self.add_uint32(Keys.LLM.FEATURES_LENGTH.format(arch=self.arch), length)

def add_posnet_embedding_length(self, length: int) -> None:
    self.add_uint32(Keys.PosNet.EMBEDDING_LENGTH.format(arch=self.arch), length)

def add_posnet_block_count(self, length: int) -> None:
    self.add_uint32(Keys.PosNet.BLOCK_COUNT.format(arch=self.arch), length)

def add_convnext_embedding_length(self, length: int) -> None:
    self.add_uint32(Keys.ConvNext.EMBEDDING_LENGTH.format(arch=self.arch), length)

def add_convnext_block_count(self, length: int) -> None:
    self.add_uint32(Keys.ConvNext.BLOCK_COUNT.format(arch=self.arch), length)

def add_block_count(self, length: int) -> None:
    self.add_uint32(Keys.LLM.BLOCK_COUNT.format(arch=self.arch), length)

def add_leading_dense_block_count(self, length: int) -> None:
    self.add_uint32(Keys.LLM.LEADING_DENSE_BLOCK_COUNT.format(arch=self.arch), length)

def add_feed_forward_length(self, length: int | Sequence[int]) -> None:
    if isinstance(length, int):
```

```
            self.add_uint32(Keys.LLM.FEED_FORWARD_LENGTH.format(arch=self.arch), length)
        else:
            self.add_array(Keys.LLM.FEED_FORWARD_LENGTH.format(arch=self.arch), length)

    def add_expert_feed_forward_length(self, length: int) -> None:
        self.add_uint32(Keys.LLM.EXPERT_FEED_FORWARD_LENGTH.format(arch=self.arch), length)

    def add_expert_shared_feed_forward_length(self, length: int) -> None:
        self.add_uint32(Keys.LLM.EXPERT_SHARED_FEED_FORWARD_LENGTH.format(arch=self.arch), length)

    def add_parallel_residual(self, use: bool) -> None:
        self.add_bool(Keys.LLM.USE_PARALLEL_RESIDUAL.format(arch=self.arch), use)

    def add_decoder_start_token_id(self, id: int) -> None:
        self.add_uint32(Keys.LLM.DECODER_START_TOKEN_ID.format(arch=self.arch), id)

    def add_head_count(self, count: int | Sequence[int]) -> None:
        if isinstance(count, int):
            self.add_uint32(Keys.Attention.HEAD_COUNT.format(arch=self.arch), count)
        else:
            self.add_array(Keys.Attention.HEAD_COUNT.format(arch=self.arch), count)

    def add_head_count_kv(self, count: int | Sequence[int]) -> None:
        if isinstance(count, int):
            self.add_uint32(Keys.Attention.HEAD_COUNT_KV.format(arch=self.arch), count)
        else:
            self.add_array(Keys.Attention.HEAD_COUNT_KV.format(arch=self.arch), count)

    def add_key_length(self, length: int) -> None:
        self.add_uint32(Keys.Attention.KEY_LENGTH.format(arch=self.arch), length)

    def add_value_length(self, length: int) -> None:
        self.add_uint32(Keys.Attention.VALUE_LENGTH.format(arch=self.arch), length)

    def add_max_alibi_bias(self, bias: float) -> None:
        self.add_float32(Keys.Attention.MAX_ALIBI_BIAS.format(arch=self.arch), bias)

    def add_clamp_kqv(self, value: float) -> None:
        self.add_float32(Keys.Attention.CLAMP_KQV.format(arch=self.arch), value)

    def add_logit_scale(self, value: float) -> None:
        self.add_float32(Keys.LLM.LOGIT_SCALE.format(arch=self.arch), value)

    def add_attn_logit_softcapping(self, value: float) -> None:
        self.add_float32(Keys.LLM.ATTN_LOGIT_SOFTCAPPING.format(arch=self.arch), value)

    def add_final_logit_softcapping(self, value: float) -> None:
        self.add_float32(Keys.LLM.FINAL_LOGIT_SOFTCAPPING.format(arch=self.arch), value)

    def add_expert_count(self, count: int) -> None:
        self.add_uint32(Keys.LLM.EXPERT_COUNT.format(arch=self.arch), count)

    def add_expert_used_count(self, count: int) -> None:
        self.add_uint32(Keys.LLM.EXPERT_USED_COUNT.format(arch=self.arch), count)
```

```
def add_expert_shared_count(self, count: int) -> None:
    self.add_uint32(Keys.LLM.EXPERT_SHARED_COUNT.format(arch=self.arch), count)

def add_expert_weights_scale(self, value: float) -> None:
    self.add_float32(Keys.LLM.EXPERT_WEIGHTS_SCALE.format(arch=self.arch), value)

def add_expert_weights_norm(self, value: bool) -> None:
    self.add_bool(Keys.LLM.EXPERT_WEIGHTS_NORM.format(arch=self.arch), value)

def add_expert_gating_func(self, value: ExpertGatingFuncType) -> None:
    self.add_uint32(Keys.LLM.EXPERT_GATING_FUNC.format(arch=self.arch), value.value)

def add_swin_norm(self, value: bool) -> None:
    self.add_bool(Keys.LLM.SWIN_NORM.format(arch=self.arch), value)

def add_rescale_every_n_layers(self, count: int) -> None:
    self.add_uint32(Keys.LLM.RESCALE_EVERY_N_LAYERS.format(arch=self.arch), count)

def add_time_mix_extra_dim(self, dim: int) -> None:
    self.add_uint32(Keys.LLM.TIME_MIX_EXTRA_DIM.format(arch=self.arch), dim)

def add_time_decay_extra_dim(self, dim: int) -> None:
    self.add_uint32(Keys.LLM.TIME_DECAY_EXTRA_DIM.format(arch=self.arch), dim)

def add_residual_scale(self, value: float) -> None:
    self.add_float32(Keys.LLM.RESIDUAL_SCALE.format(arch=self.arch), value)

def add_embedding_scale(self, value: float) -> None:
    self.add_float32(Keys.LLM.EMBEDDING_SCALE.format(arch=self.arch), value)

def add_wkv_head_size(self, size: int) -> None:
    self.add_uint32(Keys.WKV.HEAD_SIZE.format(arch=self.arch), size)

def add_token_shift_count(self, count: int) -> None:
    self.add_uint32(Keys.LLM.TOKEN_SHIFT_COUNT.format(arch=self.arch), count)

def add_interleave_moe_layer_step(self, value: int) -> None:
    self.add_uint32(Keys.LLM.INTERLEAVE_MOE_LAYER_STEP.format(arch=self.arch), value)

def add_layer_norm_eps(self, value: float) -> None:
    self.add_float32(Keys.Attention.LAYERNORM_EPS.format(arch=self.arch), value)

def add_layer_norm_rms_eps(self, value: float) -> None:
    self.add_float32(Keys.Attention.LAYERNORM_RMS_EPS.format(arch=self.arch), value)

def add_group_norm_eps(self, value: float) -> None:
    self.add_float32(Keys.Attention.GROUPNORM_EPS.format(arch=self.arch), value)

def add_group_norm_groups(self, value: int) -> None:
    self.add_uint32(Keys.Attention.GROUPNORM_GROUPS.format(arch=self.arch), value)

def add_causal_attention(self, value: bool) -> None:
    self.add_bool(Keys.Attention.CAUSAL.format(arch=self.arch), value)
```

```
def add_q_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.Q_LORA_RANK.format(arch=self.arch), length)

def add_kv_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.KV_LORA_RANK.format(arch=self.arch), length)

def add_decay_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.DECAY_LORA_RANK.format(arch=self.arch), length)

def add_iclr_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.ICLR_LORA_RANK.format(arch=self.arch), length)

def add_value_residual_mix_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.VALUE_RESIDUAL_MIX_LORA_RANK.format(arch=self.arch), length)

def add_gate_lora_rank(self, length: int) -> None:
    self.add_uint32(Keys.Attention.GATE_LORA_RANK.format(arch=self.arch), length)

def add_relative_attn_buckets_count(self, value: int) -> None:
    self.add_uint32(Keys.Attention.REL_BUCKETS_COUNT.format(arch=self.arch), value)

def add_sliding_window(self, value: int) -> None:
    self.add_uint32(Keys.Attention.SLIDING_WINDOW.format(arch=self.arch), value)

def add_attention_scale(self, value: float) -> None:
    self.add_float32(Keys.Attention.SCALE.format(arch=self.arch), value)

def add_pooling_type(self, value: PoolingType) -> None:
    self.add_uint32(Keys.LLM.POOLING_TYPE.format(arch=self.arch), value.value)

def add_rope_dimension_count(self, count: int) -> None:
    self.add_uint32(Keys.Rope.DIMENSION_COUNT.format(arch=self.arch), count)

def add_rope_dimension_sections(self, dims: Sequence[int]) -> None:
    self.add_array(Keys.Rope.DIMENSION_SECTIONS.format(arch=self.arch), dims)

def add_rope_freq_base(self, value: float) -> None:
    self.add_float32(Keys.Rope.FREQ_BASE.format(arch=self.arch), value)

def add_rope_scaling_type(self, value: RopeScalingType) -> None:
    self.add_string(Keys.Rope.SCALING_TYPE.format(arch=self.arch), value.value)

def add_rope_scaling_factor(self, value: float) -> None:
    self.add_float32(Keys.Rope.SCALING_FACTOR.format(arch=self.arch), value)

def add_rope_scaling_attn_factors(self, value: float) -> None:
    self.add_float32(Keys.Rope.SCALING_ATTN_FACTOR.format(arch=self.arch), value)

def add_rope_scaling_orig_ctx_len(self, value: int) -> None:
    self.add_uint32(Keys.Rope.SCALING_ORIG_CTX_LEN.format(arch=self.arch), value)

def add_rope_scaling_finetuned(self, value: bool) -> None:
    self.add_bool(Keys.Rope.SCALING_FINETUNED.format(arch=self.arch), value)
```

```
def add_rope_scaling_yarn_log_mul(self, value: float) -> None:
    self.add_float32(Keys.Rope.SCALING_YARN_LOG_MUL.format(arch=self.arch), value)

def add_ssm_conv_kernel(self, value: int) -> None:
    self.add_uint32(Keys.SSM.CONV_KERNEL.format(arch=self.arch), value)

def add_ssm_inner_size(self, value: int) -> None:
    self.add_uint32(Keys.SSM.INNER_SIZE.format(arch=self.arch), value)

def add_ssm_state_size(self, value: int) -> None:
    self.add_uint32(Keys.SSM.STATE_SIZE.format(arch=self.arch), value)

def add_ssm_time_step_rank(self, value: int) -> None:
    self.add_uint32(Keys.SSM.TIME_STEP_RANK.format(arch=self.arch), value)

def add_ssm_dt_b_c_rms(self, value: bool) -> None:
    self.add_bool(Keys.SSM.DT_B_C_RMS.format(arch=self.arch), value)

def add_tokenizer_model(self, model: str) -> None:
    self.add_string(Keys.Tokenizer.MODEL, model)

def add_tokenizer_pre(self, pre: str) -> None:
    self.add_string(Keys.Tokenizer.PRE, pre)

def add_token_list(self, tokens: Sequence[str] | Sequence[bytes] | Sequence[bytearray]) -> None:
    self.add_array(Keys.Tokenizer.LIST, tokens)

def add_token_merges(self, merges: Sequence[str] | Sequence[bytes] | Sequence[bytearray]) -> None:
    self.add_array(Keys.Tokenizer.MERGES, merges)

def add_token_types(self, types: Sequence[TokenType] | Sequence[int]) -> None:
    self.add_array(Keys.Tokenizer.TOKEN_TYPE, types)

def add_token_type_count(self, value: int) -> None:
    self.add_uint32(Keys.Tokenizer.TOKEN_TYPE_COUNT, value)

def add_token_scores(self, scores: Sequence[float]) -> None:
    self.add_array(Keys.Tokenizer.SCORES, scores)

def add_bos_token_id(self, id: int) -> None:
    self.add_uint32(Keys.Tokenizer.BOS_ID, id)

def add_eos_token_id(self, id: int) -> None:
    self.add_uint32(Keys.Tokenizer.EOS_ID, id)

def add_unk_token_id(self, id: int) -> None:
    self.add_uint32(Keys.Tokenizer.UNK_ID, id)

def add_sep_token_id(self, id: int) -> None:
    self.add_uint32(Keys.Tokenizer.SEP_ID, id)

def add_pad_token_id(self, id: int) -> None:
    self.add_uint32(Keys.Tokenizer.PAD_ID, id)
```

```
    def add_mask_token_id(self, id: int) -> None:
        self.add_uint32(Keys.Tokenizer.MASK_ID, id)

    def add_add_bos_token(self, value: bool) -> None:
        self.add_bool(Keys.Tokenizer.ADD_BOS, value)

    def add_add_eos_token(self, value: bool) -> None:
        self.add_bool(Keys.Tokenizer.ADD_EOS, value)

    def add_add_space_prefix(self, value: bool) -> None:
        self.add_bool(Keys.Tokenizer.ADD_PREFIX, value)

    def add_remove_extra_whitespaces(self, value: bool) -> None:
        self.add_bool(Keys.Tokenizer.REMOVE_EXTRA_WS, value)

    def add_precompiled_charsmap(self, charsmap: Sequence[bytes]) -> None:
        self.add_array(Keys.Tokenizer.PRECOMPILED_CHARSMAP, charsmap)

    def add_chat_template(self, value: str | Sequence[Mapping[str, str]]) -> None:
        if not isinstance(value, str):
            template_default = None
            template_names = set()

            for choice in value:
                name = choice.get('name', '')
                template = choice.get('template')

                    # Allowing non-alphanumerical characters in template name is probably not a good idea, so
filter it
                name = ''.join((c if c in ascii_letters + digits else '_' for c in name))

                if name and template is not None:
                    if name == 'default':
                        template_default = template
                    else:
                        template_names.add(name)
                        self.add_string(Keys.Tokenizer.CHAT_TEMPLATE_N.format(name=name), template)

            if template_names:
                self.add_array(Keys.Tokenizer.CHAT_TEMPLATES, list(template_names))

            if template_default is None:
                return

            value = template_default

        self.add_string(Keys.Tokenizer.CHAT_TEMPLATE, value)

    def add_eot_token_id(self, id: int) -> None:
        self.add_uint32(Keys.Tokenizer.EOT_ID, id)

    def add_eom_token_id(self, id: int) -> None:
        self.add_uint32(Keys.Tokenizer.EOM_ID, id)
```

```
def _pack(self, fmt: str, value: Any, skip_pack_prefix: bool = False) -> bytes:
    pack_prefix = ''
    if not skip_pack_prefix:
        pack_prefix = '<' if self.endianess == GGUFEndian.LITTLE else '>'
    return struct.pack(f'{pack_prefix}{fmt}', value)

def _pack_val(self, val: Any, vtype: GGUFValueType, add_vtype: bool) -> bytes:
    kv_data = bytearray()

    if add_vtype:
        kv_data += self._pack("I", vtype)

    pack_fmt = self._simple_value_packing.get(vtype)
    if pack_fmt is not None:
        kv_data += self._pack(pack_fmt, val, skip_pack_prefix = vtype == GGUFValueType.BOOL)
    elif vtype == GGUFValueType.STRING:
        encoded_val = val.encode("utf-8") if isinstance(val, str) else val
        kv_data += self._pack("Q", len(encoded_val))
        kv_data += encoded_val
    elif vtype == GGUFValueType.ARRAY:

        if not isinstance(val, Sequence):
            raise ValueError("Invalid GGUF metadata array, expecting sequence")

        if len(val) == 0:
            raise ValueError("Invalid GGUF metadata array. Empty array")

        if isinstance(val, bytes):
            ltype = GGUFValueType.UINT8
        else:
            ltype = GGUFValueType.get_type(val[0])
            if not all(GGUFValueType.get_type(i) is ltype for i in val[1:]):
                raise ValueError("All items in a GGUF array should be of the same type")
        kv_data += self._pack("I", ltype)
        kv_data += self._pack("Q", len(val))
        for item in val:
            kv_data += self._pack_val(item, ltype, add_vtype=False)
    else:
        raise ValueError("Invalid GGUF metadata value type or value")

    return kv_data

@staticmethod
def format_n_bytes_to_str(num: int) -> str:
    if num == 0:
        return "negligible - metadata only"
    fnum = float(num)
    for unit in ("", "K", "M", "G"):
        if abs(fnum) < 1000.0:
            return f"{fnum:3.1f}{unit}"
        fnum /= 1000.0
    return f"{fnum:.1f}T - over 1TB, split recommended"
```

```
==== glmedge-convert-image-encoder-to-gguf.py ====
import argparse
import os
import json
import re

import torch
import numpy as np
from gguf import *

TEXT = "clip.text"
VISION = "clip.vision"
from transformers import SiglipVisionModel, SiglipVisionConfig

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 name in (
        "vision_model.head.probe",
        "vision_model.head.attention.in_proj_weight",
        "vision_model.head.attention.in_proj_bias",
        "vision_model.head.attention.out_proj.weight",
        "vision_model.head.attention.out_proj.bias",
        "vision_model.head.layernorm.weight",
        "vision_model.head.layernorm.bias",
        "vision_model.head.mlp.fc1.weight",
        "vision_model.head.mlp.fc1.bias",
        "vision_model.head.mlp.fc2.weight",
        "vision_model.head.mlp.fc2.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:
    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("--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)")
ap.add_argument("--clip-model-is-openclip", action="store_true", required=False,
                help="The clip model is from openclip (for ViT-SO400M type))")
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","adapter"], default="adapter")
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 = [0.5, 0.5, 0.5]
default_image_std = [0.5, 0.5, 0.5]
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:
    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 = None

# 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

vision_config = SiglipVisionConfig(**v_hparams)
model = SiglipVisionModel(vision_config)
```

```
model.load_state_dict(torch.load(os.path.join(dir_model, "glm.clip")))

fname_middle = None
has_text_encoder = False
has_vision_encoder = True
has_glm_projector = True
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_glm_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")

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_glm_projector", has_glm_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 has_glm_projector:
    fout.add_description("image encoder for glm4v")
    fout.add_string("clip.projector_type", "adapter")
else:
    fout.add_description("two-tower CLIP model")

if has_text_encoder:
    assert t_hparams is not None
    assert tokens is not None
    # 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", t_hparams.get("projection_dim", config["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)

if has_vision_encoder:
    # 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", 0)
    fout.add_uint32(k(KEY_ATTENTION_HEAD_COUNT, VISION), v_hparams["num_attention_heads"])
    fout.add_float32(k(KEY_ATTENTION_LAYERNORM_EPS, VISION), 1e-6)
    fout.add_uint32(k(KEY_BLOCK_COUNT, VISION), v_hparams["num_hidden_layers"])

    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)

fout.add_bool("clip.use_gelu", True)


if has_glm_projector:
    # model.vision_model.encoder.layers.pop(-1)  # pyright: ignore[reportAttributeAccessIssue]
    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)
        if name.startswith("vision."):
            name=name.replace("vision.","")
        fout.add_tensor(name, data)
        print(f"Projector {name} - {data.dtype} - shape = {data.shape}")
        # print(f"Projector {name} tensors added\n")

state_dict = model.state_dict()  # pyright: ignore[reportAttributeAccessIssue]
for name, data in state_dict.items():
    if should_skip_tensor(name, has_text_encoder, has_vision_encoder, has_glm_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"siglip {name} - {data.dtype} - shape = {data.shape}")
    # 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)

==== glmedge-surgery.py ====
import argparse
import os
import torch
from transformers import AutoModel

ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", help="Path to GLM model")
args = ap.parse_args()

# find the model part that includes the the multimodal projector weights
model = AutoModel.from_pretrained(args.model, trust_remote_code=True, local_files_only=True)
checkpoint = model.state_dict()

# get a list of mm tensor names
mm_tensors = [k for k, v in checkpoint.items() if k.startswith("vision.adapter.")]

# store these tensors in a new dictionary and torch.save them
projector = {name: checkpoint[name].float() for name in mm_tensors}
torch.save(projector, f"{args.model}/glm.projector")

clip_tensors = [k for k, v in checkpoint.items() if k.startswith("vision.vit.model.vision_model.")]
if len(clip_tensors) > 0:
    clip = {name.replace("vision.vit.model.", ""): checkpoint[name].float() for name in clip_tensors}
    torch.save(clip, f"{args.model}/glm.clip")

    # added tokens should be removed to be able to convert Mistral models
    if os.path.exists(f"{args.model}/added_tokens.json"):
        with open(f"{args.model}/added_tokens.json", "w") as f:
            f.write("{}\n")

print("Done!")
print(f"Now you can convert {args.model} to a regular LLaMA GGUF file.")
```

```
print(f"Also, use {args.model}glm.projector to prepare a glm-encoder.gguf file.")

==== graph.py ====
#!/usr/bin/env python3
import matplotlib.pyplot as plt
import os
import csv

labels = []
numbers = []
numEntries = 1

rows = []


def bar_chart(numbers, labels, pos):
    plt.bar(pos, numbers, color='blue')
    plt.xticks(ticks=pos, labels=labels)
    plt.title("Jeopardy Results by Model")
    plt.xlabel("Model")
    plt.ylabel("Questions Correct")
    plt.show()


def calculatecorrect():
    directory = os.fsencode("./examples/jeopardy/results/")
    csv_reader = csv.reader(open("./examples/jeopardy/qasheet.csv", 'rt'), delimiter=',')
    for row in csv_reader:
        global rows
        rows.append(row)
    for listing in os.listdir(directory):
        filename = os.fsdecode(listing)
        if filename.endswith(".txt"):
            file = open("./examples/jeopardy/results/" + filename, "rt")
            global labels
            global numEntries
            global numbers
            labels.append(filename[:-4])
            numEntries += 1
            i = 1
            totalcorrect = 0
            for line in file.readlines():
                if line.strip() != "------":
                    print(line)
                else:
                    print("Correct answer: " + rows[i][2] + "\n")
                    i += 1
                    print("Did the AI get the question right? (y/n)")
                    if input() == "y":
                        totalcorrect += 1
            numbers.append(totalcorrect)


if __name__ == '__main__':
```

```
    calculatecorrect()
    pos = list(range(numEntries))
    labels.append("Human")
    numbers.append(48.11)
    bar_chart(numbers, labels, pos)
    print(labels)
    print(numbers)

==== guffifier_v2.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: guffifier_v2.py
Purpose: Ingest text files and model outputs, extract symbolic claims, write to incoming/,
         and distribute to helper modules for parallel digestion of massive models (540B and beyond)
"""

import os
import yaml
import threading
from utils import agent_profiler
# [PROFILER_INJECTED]
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()
from pathlib import Path
import time
import re
import uuid
import multiprocessing

IN_DIR = Path("input/")
OUT_DIR = Path("fragments/incoming")
CHUNK_DIR = Path("input/chunks")
CHUNK_DIR.mkdir(parents=True, exist_ok=True)
OUT_DIR.mkdir(parents=True, exist_ok=True)

class Guffifier:
    def __init__(self, agent_id="guffifier_01"):
        self.agent_id = agent_id

    def extract_claims(self, text):
        sentences = re.split(r'(?<=[.!?]) +', text)
        claims = [s.strip() for s in sentences if len(s.strip()) > 10]
        return claims

    def guffify(self, content, origin_path):
        claims = self.extract_claims(content)

        for claim in claims:
            fragment = {
                'id': str(uuid.uuid4())[:8],
                'origin': str(origin_path),
                'claim': claim,
                'emotion': {'neutral': 0.9},
                'confidence': 0.5,
                'timestamp': int(time.time())
```

```
            }
            out_path = OUT_DIR / f"{fragment['id']}.yaml"
            with open(out_path, 'w', encoding='utf-8') as out:
                yaml.safe_dump(fragment, out)

    def chunk_model_dump(self, path, chunk_size=50000):
        with open(path, 'r', encoding='utf-8') as f:
            text = f.read()

        base_name = path.stem
        chunks = [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]

        for i, chunk in enumerate(chunks):
            chunk_path = CHUNK_DIR / f"{base_name}_chunk_{i}.txt"
            with open(chunk_path, 'w', encoding='utf-8') as c:
                c.write(chunk)

    def batch_chunkify(self):
        raw_files = list(IN_DIR.glob("*.txt")) + list(IN_DIR.glob("*.md")) + list(IN_DIR.glob("*.json"))
        for f in raw_files:
            self.chunk_model_dump(f)

    def worker_guffify(self, chunk_path):
        with open(chunk_path, 'r', encoding='utf-8') as f:
            content = f.read()
        self.guffify(content, chunk_path)

    def run_parallel_guffifiers(self):
        chunk_paths = list(CHUNK_DIR.glob("*.txt"))
        with multiprocessing.Pool(processes=os.cpu_count() or 4) as pool:
            pool.map(self.worker_guffify, chunk_paths)

    def run(self):
        print(f"[{self.agent_id}] Chunkifying massive model files...")
        self.batch_chunkify()
        print(f"[{self.agent_id}] Distributing to guffifier helpers...")
        self.run_parallel_guffifiers()

if __name__ == "__main__":
    Guffifier().run()
# [CONFIG_PATCHED]

==== ima_gate3.py ====
"""Iterative memory attention model."""
import numpy as np
import keras.backend as K
import keras.layers as L
from keras.models import Model
import tensorflow as tf
from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
```

```
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  # Contextual embeddeding of symbols
  onehot_weights = np.eye(char_size)
  onehot_weights[0, 0] = 0 # Clear zero index
  onehot = L.Embedding(char_size, char_size,
                       trainable=False,
                       weights=[onehot_weights],
                       name='onehot')
  embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  embedded_q = onehot(query) # (?, chars, char_size)
  K.print_tensor(embedded_q)
  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
            embedded_ctx_preds      =      L.TimeDistributed(L.TimeDistributed(embed_pred,      name='nest1'),
name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel')
  att_dense = L.Dense(1, activation='sigmoid', name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
    rule_mask  =  L.Lambda(lambda  x:  K.cast(K.any(K.not_equal(x,  0),  axis=-1,  keepdims=True),  'float32'),
name='rule_mask')(embedded_rules)

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))
  gating = L.Dense(1, activation='sigmoid', name='gating')
  gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')
```