```
if args.dump:
    do_dump_model(model_plus)
    return

endianess = gguf.GGUFEndian.LITTLE
if args.big_endian:
    endianess = gguf.GGUFEndian.BIG

params = None
if args.pad_vocab or not args.vocab_only:
    params = Params.load(model_plus)
    if params.n_ctx == -1:
        if args.ctx is None:
            msg = """\
                The model doesn't have a context size, and you didn't specify one with --ctx
                Please specify one with --ctx:
                 - LLaMA v1: --ctx 2048
                 - LLaMA v2: --ctx 4096"""
            parser.error(textwrap.dedent(msg))
        params.n_ctx = args.ctx

    if args.outtype:
        params.ftype = {
            "f32": GGMLFileType.AllF32,
            "f16": GGMLFileType.MostlyF16,
            "q8_0": GGMLFileType.MostlyQ8_0,
        }[args.outtype]

    logger.info(f"params = {params}")

model_parent_path = model_plus.paths[0].parent
vocab_path = Path(args.vocab_dir or dir_model or model_parent_path)
vocab_factory = VocabFactory(vocab_path)
vocab_types = None if args.no_vocab else args.vocab_type.split(",")
vocab, special_vocab = vocab_factory.load_vocab(vocab_types, model_parent_path)

if args.vocab_only:
    assert isinstance(vocab, Vocab)
    if not args.outfile:
        raise ValueError("need --outfile if using --vocab-only")
    outfile = args.outfile
    if params is None:
        params = Params(
            n_vocab    = vocab.vocab_size,
            n_embd     = 1,
            n_layer    = 1,
            n_ctx      = 1,
            n_ff       = 1,
            n_head     = 1,
            n_head_kv  = 1,
            f_norm_eps = 1e-5,
        )
    OutputFile.write_vocab_only(outfile, params, vocab, special_vocab,
                                    endianess=endianess, pad_vocab=args.pad_vocab, metadata=metadata)
        logger.info(f"Wrote {outfile}")
        return

    if model_plus.vocab is not None and args.vocab_dir is None and not args.no_vocab:
        vocab = model_plus.vocab

    assert params is not None

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

    model_params_count = per_model_weight_count_estimation(model_plus.model.items())
                            logger.info(f"model       parameters       count       :       {model_params_count}
({gguf.model_weight_count_rounded_notation(model_params_count[0])})")

    logger.info(f"Vocab info: {vocab}")
    logger.info(f"Special vocab info: {special_vocab}")
    model   = model_plus.model
    model   = convert_model_names(model, params, args.skip_unknown)
    ftype   = pick_output_type(model, args.outtype)
    model   = convert_to_output_type(model, ftype)
     outfile = args.outfile or default_outfile(model_plus.paths, ftype, params.n_experts, model_params_count,
metadata=metadata)

    metadata.size_label = gguf.size_label(*model_params_count, expert_count=params.n_experts or 0)

    params.ftype = ftype
    logger.info(f"Writing {outfile}, format {ftype}")

    OutputFile.write_all(outfile, ftype, params, model, vocab, special_vocab,
                                  concurrency=args.concurrency, endianess=endianess, pad_vocab=args.pad_vocab,
metadata=metadata)
    logger.info(f"Wrote {outfile}")


if __name__ == '__main__':
    main()

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

import logging
import argparse
import os
import struct
import sys
from enum import IntEnum
from pathlib import Path

import numpy as np

if 'NO_LOCAL_GGUF' not in os.environ:
    sys.path.insert(1, str(Path(__file__).parent / 'gguf-py'))
import gguf

logger = logging.getLogger("ggml-to-gguf")


class GGMLFormat(IntEnum):
    GGML = 0
    GGMF = 1
    GGJT = 2


class GGMLFType(IntEnum):
    ALL_F32              = 0
    MOSTLY_F16           = 1
    MOSTLY_Q4_0          = 2
    MOSTLY_Q4_1          = 3
    MOSTLY_Q4_1_SOME_F16 = 4
    MOSTLY_Q8_0          = 7
    MOSTLY_Q5_0          = 8
    MOSTLY_Q5_1          = 9
    MOSTLY_Q2_K          = 10
    MOSTLY_Q3_K_S        = 11
    MOSTLY_Q3_K_M        = 12
    MOSTLY_Q3_K_L        = 13
    MOSTLY_Q4_K_S        = 14
    MOSTLY_Q4_K_M        = 15
    MOSTLY_Q5_K_S        = 16
    MOSTLY_Q5_K_M        = 17
    MOSTLY_Q6_K          = 18


class Hyperparameters:
    def __init__(self):
        self.n_vocab = self.n_embd = self.n_mult = self.n_head = 0
        self.n_layer = self.n_rot = self.n_ff = 0
        self.ftype = GGMLFType.ALL_F32

    def set_n_ff(self, model):
        ff_tensor_idx = model.tensor_map.get(b'layers.0.feed_forward.w1.weight')
        assert ff_tensor_idx is not None, 'Missing layer 0 FF tensor'
        ff_tensor = model.tensors[ff_tensor_idx]
        self.n_ff = ff_tensor.dims[1]

    def load(self, data, offset):
        (
            self.n_vocab,
            self.n_embd,
            self.n_mult,
            self.n_head,
            self.n_layer,
            self.n_rot,
            ftype,
        ) = struct.unpack('<7I', data[offset:offset + (4 * 7)])
        try:
            self.ftype = GGMLFType(ftype)
        except ValueError:
            raise ValueError(f'Invalid ftype {ftype}')
        return 4 * 7

    def __str__(self):
        return f'<Hyperparameters: n_vocab={self.n_vocab}, n_embd={self.n_embd}, n_mult={self.n_mult}, n_head={self.n_head}, n_layer={self.n_layer}, n_rot={self.n_rot}, n_ff={self.n_ff}, ftype={self.ftype.name}>'


class Vocab:
    def __init__(self, load_scores = True):
        self.items = []
        self.load_scores = load_scores

    def load(self, data, offset, n_vocab):
        orig_offset = offset
        for _ in range(n_vocab):
            itemlen = struct.unpack('<I', data[offset:offset + 4])[0]
            assert itemlen < 4096, 'Absurd vocab item length'
            offset += 4
            item_text = bytes(data[offset:offset + itemlen])
            offset += itemlen
            if self.load_scores:
                item_score = struct.unpack('<f', data[offset:offset + 4])[0]
                offset += 4
            else:
                item_score = 0.0
            self.items.append((item_text, item_score))
        return offset - orig_offset


class Tensor:
    def __init__(self, use_padding = True):
        self.name = None
        self.dims: tuple[int, ...] = ()
        self.dtype = None
        self.start_offset = 0
        self.len_bytes = np.int64(0)
        self.use_padding = use_padding

    def load(self, data, offset):
        orig_offset = offset
        (n_dims, name_len, dtype) = struct.unpack('<3I', data[offset:offset + 12])
        assert n_dims >= 0 and n_dims <= 4, f'Invalid tensor dimensions {n_dims}'
        assert name_len < 4096, 'Absurd tensor name length'
        quant = gguf.GGML_QUANT_SIZES.get(dtype)
        assert quant is not None, 'Unknown tensor type'
        (blksize, tysize) = quant
        offset += 12
        self.dtype= gguf.GGMLQuantizationType(dtype)
        self.dims = struct.unpack(f'<{n_dims}I', data[offset:offset + (4 * n_dims)])
        offset += 4 * n_dims
        self.name = bytes(data[offset:offset + name_len])
        offset += name_len
        pad = ((offset + 31) & ~31) - offset if self.use_padding else 0
        offset += pad
        n_elems = np.prod(self.dims)
        n_bytes = np.int64(np.int64(n_elems) * np.int64(tysize)) // np.int64(blksize)
        self.start_offset = offset
        self.len_bytes = n_bytes
        offset += n_bytes
        return offset - orig_offset


class GGMLModel:

    file_format: GGMLFormat
    format_version: int

    def __init__(self):
        self.hyperparameters = None
        self.vocab = None
        self.tensor_map = {}
        self.tensors = []

    def validate_header(self, data, offset):
        magic = bytes(data[offset:offset + 4])
        if magic == b'GGUF':
            raise ValueError('File is already in GGUF format.')
        if magic == b'lmgg':
            self.file_format = GGMLFormat.GGML
            self.format_version = 1
            return 4
        version = struct.unpack('<I', data[offset + 4:offset + 8])[0]
        if magic == b'fmgg':
            if version != 1:
                raise ValueError(f'Cannot handle unexpected GGMF file version {version}')
            self.file_format = GGMLFormat.GGMF
            self.format_version = version
            return 8
        if magic == b'tjgg':
            if version < 1 or version > 3:
                raise ValueError(f'Cannot handle unexpected GGJT file version {version}')
            self.file_format = GGMLFormat.GGJT
            self.format_version = version
            return 8
        raise ValueError(f"Unexpected file magic {magic!r}! This doesn't look like a GGML format file.")

    def validate_conversion(self, ftype):
        err = ''
        if (self.file_format < GGMLFormat.GGJT or self.format_version < 2):
            if ftype not in (GGMLFType.ALL_F32, GGMLFType.MOSTLY_F16):
                    err = 'Quantizations changed in GGJTv2. Can only convert unquantized GGML files older than
GGJTv2.'
        elif (self.file_format == GGMLFormat.GGJT and self.format_version == 2):
            if ftype in (GGMLFType.MOSTLY_Q4_0, GGMLFType.MOSTLY_Q4_1,
                        GGMLFType.MOSTLY_Q4_1_SOME_F16, GGMLFType.MOSTLY_Q8_0):
                err = 'Q4 and Q8 quantizations changed in GGJTv3.'
        if len(err) > 0:
            raise ValueError(f'{err} Sorry, your {self.file_format.name}v{self.format_version} file of type
{ftype.name} is not eligible for conversion.')

    def load(self, data, offset):
        offset += self.validate_header(data, offset)
        hp = Hyperparameters()
        offset += hp.load(data, offset)
        logger.info(f'* File format: {self.file_format.name}v{self.format_version} with ftype {hp.ftype.name}')
        self.validate_conversion(hp.ftype)
        vocab = Vocab(load_scores = self.file_format > GGMLFormat.GGML)
        offset += vocab.load(data, offset, hp.n_vocab)
        tensors: list[Tensor] = []
        tensor_map = {}
        while offset < len(data):
            tensor = Tensor(use_padding = self.file_format > GGMLFormat.GGMF)
            offset += tensor.load(data, offset)
            tensor_map[tensor.name] = len(tensors)
            tensors.append(tensor)
        self.hyperparameters = hp
        self.vocab = vocab
        self.tensors = tensors
        self.tensor_map = tensor_map
        hp.set_n_ff(self)
        return offset


class GGMLToGGUF:
    def __init__(self, ggml_model, data, cfg, params_override = None, vocab_override = None, special_vocab =
None):
        hp = ggml_model.hyperparameters
        self.model = ggml_model
        self.data = data
        self.cfg = cfg
        self.params_override = params_override
        self.vocab_override = vocab_override
        self.special_vocab = special_vocab
        if params_override is not None:
            n_kv_head = params_override.n_head_kv
        else:
            if cfg.gqa == 1:
                n_kv_head = hp.n_head
            else:
                gqa = float(cfg.gqa)
                n_kv_head = None
                for x in range(1, 256):
                    if float(hp.n_head) / float(x) == gqa:
                        n_kv_head = x
                assert n_kv_head is not None, "Couldn't determine n_kv_head from GQA param"
                logger.info(f'- Guessed n_kv_head = {n_kv_head} based on GQA {cfg.gqa}')
        self.n_kv_head = n_kv_head
        self.name_map = gguf.get_tensor_name_map(gguf.MODEL_ARCH.LLAMA, ggml_model.hyperparameters.n_layer)
    def save(self):
        logger.info('* Preparing to save GGUF file')
        gguf_writer = gguf.GGUFWriter(
            self.cfg.output,
            gguf.MODEL_ARCH_NAMES[gguf.MODEL_ARCH.LLAMA],
            use_temp_file = False)
        self.add_params(gguf_writer)
        self.add_vocab(gguf_writer)
        if self.special_vocab is not None:
            self.special_vocab.add_to_gguf(gguf_writer)
        self.add_tensors(gguf_writer)
        logger.info("    gguf: write header")
        gguf_writer.write_header_to_file()
        logger.info("    gguf: write metadata")
        gguf_writer.write_kv_data_to_file()
        logger.info("    gguf: write tensors")
        gguf_writer.write_tensors_to_file()
        gguf_writer.close()

    def add_params(self, gguf_writer):
        hp = self.model.hyperparameters
        cfg = self.cfg
        if cfg.desc is not None:
            desc = cfg.desc
        else:
                    desc  =  f'converted  from  legacy  {self.model.file_format.name}v{self.model.format_version}
{hp.ftype.name} format'
        try:
            # Filenames aren't necessarily valid UTF8.
            name = cfg.name if cfg.name is not None else cfg.input.name
        except UnicodeDecodeError:
            name = None
        logger.info('* Adding model parameters and KV items')
        if name is not None:
            gguf_writer.add_name(name)
        gguf_writer.add_description(desc)
        gguf_writer.add_file_type(int(hp.ftype))
        if self.params_override is not None:
            po = self.params_override
            assert po.n_embd == hp.n_embd, 'Model hyperparams mismatch'
            assert po.n_layer == hp.n_layer, 'Model hyperparams mismatch'
            assert po.n_head == hp.n_head, 'Model hyperparams mismatch'
            gguf_writer.add_context_length      (po.n_ctx)
            gguf_writer.add_embedding_length    (po.n_embd)
            gguf_writer.add_block_count         (po.n_layer)
            gguf_writer.add_feed_forward_length (po.n_ff)
            gguf_writer.add_rope_dimension_count(po.n_embd // po.n_head)
            gguf_writer.add_head_count          (po.n_head)
            gguf_writer.add_head_count_kv       (po.n_head_kv)
            gguf_writer.add_layer_norm_rms_eps  (po.f_norm_eps)
            return
        gguf_writer.add_context_length(cfg.context_length)
        gguf_writer.add_embedding_length(hp.n_embd)
        gguf_writer.add_block_count(hp.n_layer)
        gguf_writer.add_feed_forward_length(hp.n_ff)
        gguf_writer.add_rope_dimension_count(hp.n_embd // hp.n_head)
        gguf_writer.add_head_count(hp.n_head)
        gguf_writer.add_head_count_kv(self.n_kv_head)
        gguf_writer.add_layer_norm_rms_eps(float(cfg.eps))

    def add_vocab(self, gguf_writer):
        hp = self.model.hyperparameters
        gguf_writer.add_tokenizer_model('llama')
        gguf_writer.add_tokenizer_pre('default')
        tokens = []
        scores = []
        toktypes = []
        if self.vocab_override is not None:
            vo = self.vocab_override
            logger.info('* Adding vocab item(s)')
            for (_, (vbytes, score, ttype)) in enumerate(vo.all_tokens()):
                tokens.append(vbytes)
                scores.append(score)
                toktypes.append(ttype)
            assert len(tokens) == hp.n_vocab, \
                        f'Override vocab has a different number of items than hyperparameters - override = 
{len(tokens)} but n_vocab={hp.n_vocab}'
            gguf_writer.add_token_list(tokens)
            gguf_writer.add_token_scores(scores)
            if len(toktypes) > 0:
                gguf_writer.add_token_types(toktypes)
            return
        logger.info(f'* Adding {hp.n_vocab} vocab item(s)')
        assert len(self.model.vocab.items) >= 3, 'Cannot handle unexpectedly short model vocab'
        for (tokid, (vbytes, vscore)) in enumerate(self.model.vocab.items):
            tt = 1 # Normal
            # Special handling for UNK, BOS, EOS tokens.
            if tokid <= 2:
                if tokid == 0:
                    vbytes = b'<unk>'
                    tt = 2
                elif tokid == 1:
                    vbytes = b'<s>'
                    tt = 3
                else:
                    vbytes = b'</s>'
                    tt = 3
            elif len(vbytes) == 0:
                tt = 3 # Control
            elif tokid >= 3 and tokid <= 258 and len(vbytes) == 1:
                vbytes = bytes(f'<0x{vbytes[0]:02X}>', encoding = 'UTF-8')
                tt = 6 # Byte
            else:
                vbytes = vbytes.replace(b' ', b'\xe2\x96\x81')
            toktypes.append(tt)
            tokens.append(vbytes)
            scores.append(vscore)
        gguf_writer.add_token_list(tokens)
        gguf_writer.add_token_scores(scores)
        gguf_writer.add_token_types(toktypes)
        gguf_writer.add_unk_token_id(0)
        gguf_writer.add_bos_token_id(1)
        gguf_writer.add_eos_token_id(2)

    def add_tensors(self, gguf_writer):
        tensor_map = self.name_map
        data = self.data
        logger.info(f'* Adding {len(self.model.tensors)} tensor(s)')
        for tensor in self.model.tensors:
            name = str(tensor.name, 'UTF-8')
            mapped_name = tensor_map.get_name(name, try_suffixes = (".weight", ".bias"))
            assert mapped_name is not None, f'Bad name {name}'
            tempdims = list(tensor.dims[:])
            if len(tempdims) > 1:
                temp = tempdims[1]
                tempdims[1] = tempdims[0]
                tempdims[0] = temp
            gguf_writer.add_tensor(
                mapped_name,
                data[tensor.start_offset:tensor.start_offset + tensor.len_bytes],
                raw_shape = tempdims,
                raw_dtype = tensor.dtype)



def handle_metadata(cfg, hp):
    import examples.convert_legacy_llama as convert

    assert cfg.model_metadata_dir.is_dir(), 'Metadata dir is not a directory'
    hf_config_path   = cfg.model_metadata_dir / "config.json"
    orig_config_path = cfg.model_metadata_dir / "params.json"
    # We pass a fake model here. "original" mode will check the shapes of some
    # tensors if information is missing in the .json file: other than that, the
    # model data isn't used so this should be safe (at least for now).
    fakemodel = {
        'tok_embeddings.weight': convert.LazyTensor.__new__(convert.LazyTensor),
        'layers.0.feed_forward.w1.weight': convert.LazyTensor.__new__(convert.LazyTensor),
    }
    fakemodel['tok_embeddings.weight'].shape = [hp.n_vocab]
    fakemodel['layers.0.feed_forward.w1.weight'].shape = [hp.n_ff]
    if hf_config_path.exists():
        params = convert.Params.loadHFTransformerJson(fakemodel, hf_config_path)
    elif orig_config_path.exists():
        params = convert.Params.loadOriginalParamsJson(fakemodel, orig_config_path)
    else:
        raise ValueError('Unable to load metadata')
    vocab_path = Path(cfg.vocab_dir if cfg.vocab_dir is not None else cfg.model_metadata_dir)
    vocab_factory = convert.VocabFactory(vocab_path)
    vocab, special_vocab = vocab_factory.load_vocab(cfg.vocabtype.split(","), cfg.model_metadata_dir)
    convert.check_vocab_size(params, vocab)
    return params, vocab, special_vocab
```

```
def handle_args():
    parser = argparse.ArgumentParser(description = 'Convert GGML models to GGUF')
    parser.add_argument('--input', '-i', type = Path, required = True,
                        help = 'Input GGMLv3 filename')
    parser.add_argument('--output', '-o', type = Path, required = True,
                        help ='Output GGUF filename')
    parser.add_argument('--name',
                        help = 'Set model name')
    parser.add_argument('--desc',
                        help = 'Set model description')
    parser.add_argument('--gqa', type = int, default = 1,
                        help = 'grouped-query attention factor (use 8 for LLaMA2 70B)')
    parser.add_argument('--eps', default = '5.0e-06',
                        help = 'RMS norm eps: Use 1e-6 for LLaMA1 and OpenLLaMA, use 1e-5 for LLaMA2')
    parser.add_argument('--context-length', '-c', type=int, default = 2048,
                        help = 'Default max context length: LLaMA1 is typically 2048, LLaMA2 is typically 4096')
    parser.add_argument('--model-metadata-dir', '-m', type = Path,
                        help ='Load HuggingFace/.pth vocab and metadata from the specified directory')
    parser.add_argument("--vocab-dir", type=Path,
                        help="directory containing tokenizer.model, if separate from model file - only meaningful with --model-metadata-dir")
    parser.add_argument("--vocabtype", default="spm,hfft",
                        help="vocab format - only meaningful with --model-metadata-dir and/or --vocab-dir (default: spm,hfft)")
    parser.add_argument("--verbose", action="store_true", help="increase output verbosity")
    return parser.parse_args()


def main():
    cfg = handle_args()
    logging.basicConfig(level=logging.DEBUG if cfg.verbose else logging.INFO)
    logger.info(f'* Using config: {cfg}')
    logger.warning('=== WARNING === Be aware that this conversion script is best-effort. Use a native GGUF model if possible. === WARNING ===')
    if cfg.model_metadata_dir is None and (cfg.gqa == 1 or cfg.eps == '5.0e-06'):
        logger.info('- Note: If converting LLaMA2, specifying "--eps 1e-5" is required. 70B models also need "--gqa 8".')
    data = np.memmap(cfg.input, mode = 'r')
    model = GGMLModel()
    logger.info('* Scanning GGML input file')
    offset = model.load(data, 0)  # noqa
    logger.info(f'* GGML model hyperparameters: {model.hyperparameters}')
    vocab_override = None
    params_override = None
    special_vocab = None
    if cfg.model_metadata_dir is not None:
        (params_override, vocab_override, special_vocab) = handle_metadata(cfg, model.hyperparameters)
        logger.info('!! Note: When overriding params the --gqa, --eps and --context-length options are ignored.')
        logger.info(f'* Overriding params: {params_override}')
        logger.info(f'* Overriding vocab: {vocab_override}')
        logger.info(f'* Special vocab: {special_vocab}')
    else:
            logger.warning('\n=== WARNING === Special tokens may not be converted correctly. Use
--model-metadata-dir if possible === WARNING ===\n')
        if model.file_format == GGMLFormat.GGML:
            logger.info('! This is a very old GGML file that does not contain vocab scores. Strongly recommend
using model metadata!')
    converter = GGMLToGGUF(
        model, data, cfg,
        params_override = params_override,
        vocab_override = vocab_override,
        special_vocab = special_vocab
    )
    converter.save()
    logger.info(f'* Successful completion. Output saved to: {cfg.output}')


if __name__ == '__main__':
    main()

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

from __future__ import annotations

from dataclasses import dataclass
import logging
import argparse
import os
import sys
import json
from math import prod
from pathlib import Path
from typing import TYPE_CHECKING, Any, Callable, Iterable, Iterator, Sequence, SupportsIndex, cast
from transformers import AutoConfig

import torch

if TYPE_CHECKING:
    from torch import Tensor

if 'NO_LOCAL_GGUF' not in os.environ:
    sys.path.insert(1, str(Path(__file__).parent / 'gguf-py'))
import gguf

# reuse model definitions from convert_hf_to_gguf.py
from convert_hf_to_gguf import LazyTorchTensor, Model

logger = logging.getLogger("lora-to-gguf")


@dataclass
class PartialLoraTensor:
    A: Tensor | None = None
```

```python
    B: Tensor | None = None


# magic to support tensor shape modifications and splitting
class LoraTorchTensor:
    _lora_A: Tensor  # (n_rank, row_size)
    _lora_B: Tensor  # (col_size, n_rank)
    _rank: int

    def __init__(self, A: Tensor, B: Tensor):
        assert len(A.shape) == len(B.shape)
        assert A.shape[-2] == B.shape[-1]
        if A.dtype != B.dtype:
            A = A.to(torch.float32)
            B = B.to(torch.float32)
        self._lora_A = A
        self._lora_B = B
        self._rank = B.shape[-1]

    def get_lora_A_B(self) -> tuple[Tensor, Tensor]:
        return (self._lora_A, self._lora_B)

    def __getitem__(
        self,
        indices: (
            SupportsIndex
            | slice
            | tuple[SupportsIndex | slice | Tensor, ...]  # TODO: add ellipsis in the type signature
        ),
    ) -> LoraTorchTensor:
        shape = self.shape
        if isinstance(indices, SupportsIndex):
            if len(shape) > 2:
                return LoraTorchTensor(self._lora_A[indices], self._lora_B[indices])
            else:
                raise NotImplementedError  # can't return a vector
        elif isinstance(indices, slice):
            if len(shape) > 2:
                return LoraTorchTensor(self._lora_A[indices], self._lora_B[indices])
            else:
                return LoraTorchTensor(self._lora_A, self._lora_B[indices])
        elif isinstance(indices, tuple):
            assert len(indices) > 0
            if indices[-1] is Ellipsis:
                return self[indices[:-1]]
            # expand ellipsis
            indices = tuple(
                u
                for v in (
                    (
                        (slice(None, None) for _ in range(len(indices) - 1))
                        if i is Ellipsis
                        else (i,)
                    )
                    for i in indices
                )
                for u in v
            )

            if len(indices) < len(shape):
                indices = (*indices, *(slice(None, None) for _ in range(len(indices), len(shape))))

            # TODO: make sure this is correct
            indices_A = (
                *(
                    (
                        j.__index__() % self._lora_A.shape[i]
                        if isinstance(j, SupportsIndex)
                        else slice(None, None)
                    )
                    for i, j in enumerate(indices[:-2])
                ),
                slice(None, None),
                indices[-1],
            )
            indices_B = indices[:-1]
            return LoraTorchTensor(self._lora_A[indices_A], self._lora_B[indices_B])
        else:
            raise NotImplementedError  # unknown indice type

    @property
    def dtype(self) -> torch.dtype:
        assert self._lora_A.dtype == self._lora_B.dtype
        return self._lora_A.dtype

    @property
    def shape(self) -> tuple[int, ...]:
        assert len(self._lora_A.shape) == len(self._lora_B.shape)
        return (*self._lora_B.shape[:-1], self._lora_A.shape[-1])

    def size(self, dim=None):
        assert dim is None
        return self.shape

    def reshape(self, *shape: int | tuple[int, ...]) -> LoraTorchTensor:
        if isinstance(shape[0], tuple):
            new_shape: tuple[int, ...] = shape[0]
        else:
            new_shape = cast(tuple[int, ...], shape)
        orig_shape = self.shape
        if len(new_shape) < 2:
            raise NotImplementedError  # can't become a vector

        # expand -1 in the shape
        if any(dim == -1 for dim in new_shape):
            n_elems = prod(orig_shape)
            n_new_elems = prod(dim if dim != -1 else 1 for dim in new_shape)
            assert n_elems % n_new_elems == 0
```

```
            new_shape = (*(dim if dim != -1 else n_elems // n_new_elems for dim in new_shape),)

        if new_shape[-1] != orig_shape[-1]:
            raise NotImplementedError  # can't reshape the row size trivially

        shape_A = (*(1 for _ in new_shape[:-2]), self._rank, orig_shape[-1])
        shape_B = (*new_shape[:-1], self._rank)
        return LoraTorchTensor(
            self._lora_A.reshape(shape_A),
            self._lora_B.reshape(shape_B),
        )

    def reshape_as(self, other: Tensor) -> LoraTorchTensor:
        return self.reshape(*other.shape)

    def view(self, *size: int) -> LoraTorchTensor:
        return self.reshape(*size)

    def permute(self, *dims: int) -> LoraTorchTensor:
        shape = self.shape
        dims = tuple(dim - len(shape) if dim >= 0 else dim for dim in dims)
        if dims[-1] == -1:
            # TODO: support higher dimensional A shapes bigger than 1
            assert all(dim == 1 for dim in self._lora_A.shape[:-2])
            return LoraTorchTensor(self._lora_A, self._lora_B.permute(*dims))
        if len(shape) == 2 and dims[-1] == -2 and dims[-2] == -1:
            return LoraTorchTensor(self._lora_B.permute(*dims), self._lora_A.permute(*dims))
        else:
            # TODO: compose the above two
            raise NotImplementedError

    def transpose(self, dim0: int, dim1: int) -> LoraTorchTensor:
        shape = self.shape
        dims = [i for i in range(len(shape))]
        dims[dim0], dims[dim1] = dims[dim1], dims[dim0]
        return self.permute(*dims)

    def swapaxes(self, axis0: int, axis1: int) -> LoraTorchTensor:
        return self.transpose(axis0, axis1)

    def to(self, *args, **kwargs):
        return LoraTorchTensor(self._lora_A.to(*args, **kwargs), self._lora_B.to(*args, **kwargs))

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

        if kwargs is None:
            kwargs = {}

        if func is torch.permute:
            return type(args[0]).permute(*args, **kwargs)
        elif func is torch.reshape:
            return type(args[0]).reshape(*args, **kwargs)
        elif func is torch.stack:
            assert isinstance(args[0], Sequence)
            dim = kwargs.get("dim", 0)
            assert dim == 0
            return LoraTorchTensor(
                torch.stack([a._lora_A for a in args[0]], dim),
                torch.stack([b._lora_B for b in args[0]], dim),
            )
        elif func is torch.cat:
            assert isinstance(args[0], Sequence)
            dim = kwargs.get("dim", 0)
            assert dim == 0
            if len(args[0][0].shape) > 2:
                return LoraTorchTensor(
                    torch.cat([a._lora_A for a in args[0]], dim),
                    torch.cat([b._lora_B for b in args[0]], dim),
                )
            elif all(torch.equal(args[0][0]._lora_A, t._lora_A) for t in args[0][1:]):
                return LoraTorchTensor(
                    args[0][0]._lora_A,
                    torch.cat([b._lora_B for b in args[0]], dim),
                )
            else:
                raise NotImplementedError
        else:
            raise NotImplementedError


def get_base_tensor_name(lora_tensor_name: str) -> str:
    base_name = lora_tensor_name.replace("base_model.model.", "")
    base_name = base_name.replace(".lora_A.weight", ".weight")
    base_name = base_name.replace(".lora_B.weight", ".weight")
    # models produced by mergekit-extract-lora have token embeddings in the adapter
    base_name = base_name.replace(".lora_embedding_A", ".weight")
    base_name = base_name.replace(".lora_embedding_B", ".weight")
    return base_name


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Convert a Hugging Face PEFT LoRA adapter to a GGUF file")
    parser.add_argument(
        "--outfile", type=Path,
        help="path to write to; default: based on input. {ftype} will be replaced by the outtype.",
    )
    parser.add_argument(
        "--outtype", type=str, choices=["f32", "f16", "bf16", "q8_0", "auto"], default="f16",
        help="output format - use f32 for float32, f16 for float16, bf16 for bfloat16, q8_0 for Q8_0, auto for
the highest-fidelity 16-bit float type depending on the first loaded tensor type",
    )
    parser.add_argument(
        "--bigendian", action="store_true",
        help="model is executed on big endian machine",
    )
    parser.add_argument(
        "--no-lazy", action="store_true",
        help="use more RAM by computing all outputs before writing (use in case lazy evaluation is broken)",
    )
    parser.add_argument(
        "--verbose", action="store_true",
        help="increase output verbosity",
    )
    parser.add_argument(
        "--dry-run", action="store_true",
        help="only print out what will be done, without writing any new files",
    )
    parser.add_argument(
        "--base", type=Path,
        help="directory containing Hugging Face model config files (config.json, tokenizer.json) for the base model that the adapter is based on - only config is needed, actual model weights are not required. If base model is unspecified, it will be loaded from Hugging Face hub based on the adapter config",
    )
    parser.add_argument(
        "--base-model-id", type=str,
        help="the model ID of the base model, if it is not available locally or in the adapter config. If specified, it will ignore --base and load the base model config from the Hugging Face hub (Example: 'meta-llama/Llama-3.2-1B-Instruct')",
    )
    parser.add_argument(
        "lora_path", type=Path,
        help="directory containing Hugging Face PEFT LoRA config (adapter_model.json) and weights (adapter_model.safetensors or adapter_model.bin)",
    )

    return parser.parse_args()


def load_hparams_from_hf(hf_model_id: str) -> dict[str, Any]:
    # normally, adapter does not come with base model config, we need to load it from AutoConfig
    config = AutoConfig.from_pretrained(hf_model_id)
    return config.to_dict()


if __name__ == '__main__':
    args = parse_args()
    logging.basicConfig(level=logging.DEBUG if args.verbose else logging.INFO)

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

    ftype = ftype_map[args.outtype]

    dir_base_model: Path | None = args.base
    dir_lora: Path = args.lora_path
    base_model_id: str | None = args.base_model_id
    lora_config = dir_lora / "adapter_config.json"
    input_model = dir_lora / "adapter_model.safetensors"

    if args.outfile is not None:
        fname_out = args.outfile
    else:
        # output in the same directory as the model by default
        fname_out = dir_lora

    if os.path.exists(input_model):
        # lazy import load_file only if lora is in safetensors format.
        from safetensors.torch import load_file

        lora_model = load_file(input_model, device="cpu")
    else:
        input_model = os.path.join(dir_lora, "adapter_model.bin")
        lora_model = torch.load(input_model, map_location="cpu", weights_only=True)

    # load LoRA config
    with open(lora_config, "r") as f:
        lparams: dict[str, Any] = json.load(f)

    # load base model
    if base_model_id is not None:
        logger.info(f"Loading base model from Hugging Face: {base_model_id}")
        hparams = load_hparams_from_hf(base_model_id)
    elif dir_base_model is None:
        if "base_model_name_or_path" in lparams:
            model_id = lparams["base_model_name_or_path"]
            logger.info(f"Loading base model from Hugging Face: {model_id}")
            try:
                hparams = load_hparams_from_hf(model_id)
            except OSError as e:
                logger.error(f"Failed to load base model config: {e}")
                logger.error("Please try downloading the base model and add its path to --base")
                sys.exit(1)
        else:
            logger.error("'base_model_name_or_path' is not found in adapter_config.json")
              logger.error("Base model config is required. Please download the base model and add its path to
--base")
            sys.exit(1)
    else:
        logger.info(f"Loading base model: {dir_base_model.name}")
        hparams = Model.load_hparams(dir_base_model)

    with torch.inference_mode():
        try:
            model_class = Model.from_model_architecture(hparams["architectures"][0])
        except NotImplementedError:
            logger.error(f"Model {hparams['architectures'][0]} is not supported")
            sys.exit(1)
        class LoraModel(model_class):
            model_arch = model_class.model_arch

            lora_alpha: float

            def __init__(self, *args, dir_lora_model: Path, lora_alpha: float, **kwargs):

                super().__init__(*args, **kwargs)

                self.dir_model_card = dir_lora_model
                self.lora_alpha = float(lora_alpha)

            def set_vocab(self):
                pass

            def set_type(self):
                self.gguf_writer.add_type(gguf.GGUFType.ADAPTER)
                self.gguf_writer.add_string(gguf.Keys.Adapter.TYPE, "lora")

            def set_gguf_parameters(self):
                self.gguf_writer.add_float32(gguf.Keys.Adapter.LORA_ALPHA, self.lora_alpha)

            def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
                # Never add extra tensors (e.g. rope_freqs) for LoRA adapters
                return ()

            def get_tensors(self) -> Iterator[tuple[str, Tensor]]:
                tensor_map: dict[str, PartialLoraTensor] = {}

                for name, tensor in lora_model.items():
                    if self.lazy:
                        tensor = LazyTorchTensor.from_eager(tensor)
                    base_name = get_base_tensor_name(name)
                    # note: mergekit-extract-lora also adds token embeddings to the adapter
                    is_lora_a = ".lora_A.weight" in name or ".lora_embedding_A" in name
                    is_lora_b = ".lora_B.weight" in name or ".lora_embedding_B" in name
                    if not is_lora_a and not is_lora_b:
                        if ".base_layer.weight" in name:
                            continue
                        # mergekit-extract-lora add these layernorm to the adapter, we need to keep them
                        if "_layernorm" in name or ".norm" in name:
                            yield (base_name, tensor)
                            continue
                        logger.error(f"Unexpected name '{name}': Not a lora_A or lora_B tensor")
                        if ".embed_tokens.weight" in name or ".lm_head.weight" in name:
                             logger.error("Embeddings is present in the adapter. This can be due to new tokens
added during fine tuning")
                            logger.error("Please refer to https://github.com/ggml-org/llama.cpp/pull/9948")
                        sys.exit(1)

                    if base_name in tensor_map:
                        if is_lora_a:
                            tensor_map[base_name].A = tensor
                        else:
                            tensor_map[base_name].B = tensor
                    else:
                        if is_lora_a:
                            tensor_map[base_name] = PartialLoraTensor(A=tensor)
                        else:
                            tensor_map[base_name] = PartialLoraTensor(B=tensor)

                for name, tensor in tensor_map.items():
                    assert tensor.A is not None
                    assert tensor.B is not None
                    yield (name, cast(torch.Tensor, LoraTorchTensor(tensor.A, tensor.B)))

            def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str,
Tensor]]:
                dest = list(super().modify_tensors(data_torch, name, bid))
                # some archs may have the same tensor for lm_head and output (tie word embeddings)
                # in this case, adapters targeting lm_head will fail when using llama-export-lora
                # therefore, we ignore them for now
                # see: https://github.com/ggml-org/llama.cpp/issues/9065
                if name == "lm_head.weight" and len(dest) == 0:
                    raise ValueError("lm_head is present in adapter, but is ignored in base model")
                for dest_name, dest_data in dest:
                    # mergekit-extract-lora add these layernorm to the adapter
                    if "_norm" in dest_name:
                        assert dest_data.dim() == 1
                        yield (dest_name, dest_data)
                        continue

                    # otherwise, we must get the lora_A and lora_B tensors
                    assert isinstance(dest_data, LoraTorchTensor)
                    lora_a, lora_b = dest_data.get_lora_A_B()

                    # note: mergekit-extract-lora flip and transpose A and B
                    # here we only need to transpose token_embd.lora_a, see llm_build_inp_embd()
                    if "token_embd.weight" in dest_name:
                        lora_a = lora_a.T

                    yield (dest_name + ".lora_a", lora_a)
                    yield (dest_name + ".lora_b", lora_b)

        alpha: float = lparams["lora_alpha"]

        model_instance = LoraModel(
            dir_base_model,
            ftype,
            fname_out,
            is_big_endian=args.bigendian,
            use_temp_file=False,
            eager=args.no_lazy,
            dry_run=args.dry_run,
            dir_lora_model=dir_lora,
            lora_alpha=alpha,
            hparams=hparams,
        )
        logger.info("Exporting model...")
        model_instance.write()
        logger.info(f"Model successfully exported to {model_instance.fname_out}")

==== convert_pt_to_hf.py ====
# convert the https://huggingface.co/novateur/WavTokenizer-large-speech-75token to HF format
# the goal is to be able to reuse the convert_hf_to_gguf.py after that to create a GGUF file with the WavTokenizer decoder
#
# TODO: this script is LLM-generated and probably very inefficient and should be rewritten

import torch
import json
import os
import sys
import re

from safetensors.torch import save_file

# default
model_path = './model.pt';

# read from CLI
if len(sys.argv) > 1:
    model_path = sys.argv[1]

# get the directory of the input model
path_dst = os.path.dirname(model_path)

print(f"Loading model from {model_path}")

model = torch.load(model_path, map_location='cpu')

#print(model)

# print all keys
for key in model.keys():
    print(key)
    if key == 'hyper_parameters':
        #print(model[key])
        # dump as json pretty
        print(json.dumps(model[key], indent=4))
    #if key != 'state_dict' and key != 'optimizer_states':
    #    print(model[key])

# Check if the loaded model is a state_dict or a model instance
if isinstance(model, torch.nn.Module):
    state_dict = model.state_dict()
else:
    state_dict = model

# Print the structure of the state_dict to understand its format
print("State dictionary keys:")
```

```
for key in state_dict.keys():
    print(key)

# Ensure the state_dict is flat and contains only torch.Tensor objects
def flatten_state_dict(state_dict, parent_key='', sep='.'):
    items = []
    items_new = []

    for k, v in state_dict.items():
        new_key = f"{parent_key}{sep}{k}" if parent_key else k
        if isinstance(v, torch.Tensor):
            items.append((new_key, v))
        elif isinstance(v, dict):
            items.extend(flatten_state_dict(v, new_key, sep=sep).items())
            return dict(items)

    size_total_mb = 0

    for key, value in list(items):
        # keep only what we need for inference
        if not key.startswith('state_dict.feature_extractor.encodec.quantizer.') and \
           not key.startswith('state_dict.backbone.') and \
           not key.startswith('state_dict.head.out'):
               print('Skipping key: ', key)
               continue

        new_key = key

        new_key = new_key.replace('state_dict.', '')
        new_key = new_key.replace('pos_net', 'posnet')

        # check if matches "backbone.posnet.%d.bias" or "backbone.posnet.%d.weight"
        if new_key.startswith("backbone.posnet."):
            match = re.match(r"backbone\.posnet\.(\d+)\.(bias|weight)", new_key)
            if match:
               new_key = f"backbone.posnet.{match.group(1)}.norm.{match.group(2)}"

        # "feature_extractor.encodec.quantizer.vq.layers.0._codebook.embed" -> "backbone.embedding.weight"
        if new_key == "feature_extractor.encodec.quantizer.vq.layers.0._codebook.embed":
            new_key = "backbone.embedding.weight"

        # these are the only rows used
                                                                                                    #            ref:
https://github.com/edwko/OuteTTS/blob/a613e79c489d8256dd657ea9168d78de75895d82/outetts/wav_tokenizer/audio_code
c.py#L100
        if new_key.endswith("norm.scale.weight"):
            new_key = new_key.replace("norm.scale.weight", "norm.weight")
            value = value[0]

        if new_key.endswith("norm.shift.weight"):
            new_key = new_key.replace("norm.shift.weight", "norm.bias")
            value = value[0]

        if new_key.endswith("gamma"):
            new_key = new_key.replace("gamma", "gamma.weight")

        # convert from 1D [768] to 2D [768, 1] so that ggml_add can broadcast the bias
                            if    (new_key.endswith("norm.weight")    or    new_key.endswith("norm1.weight")    or
new_key.endswith("norm2.weight")  or  new_key.endswith(".bias"))  and  (new_key.startswith("backbone.posnet")  or
new_key.startswith("backbone.embed.bias")):
            value = value.unsqueeze(1)

        if new_key.endswith("dwconv.bias"):
            value = value.unsqueeze(1)

        size_mb = value.element_size() * value.nelement() / (1024 * 1024)
        print(f"{size_mb:8.2f} MB - {new_key}: {value.shape}")

        size_total_mb += size_mb

        #print(key, '->', new_key, ': ', value)
        #print(key, '->', new_key)

        items_new.append((new_key, value))

    print(f"Total size: {size_total_mb:8.2f} MB")

    return dict(items_new)

flattened_state_dict = flatten_state_dict(state_dict)


# Convert the model to the safetensors format
output_path = path_dst + '/model.safetensors'
save_file(flattened_state_dict, output_path)

print(f"Model has been successfully converted and saved to {output_path}")

# Calculate the total size of the .safetensors file
total_size = os.path.getsize(output_path)

# Create the weight map
weight_map = {
    "model.safetensors": ["*"]  # Assuming all weights are in one file
}

# Create metadata for the index.json file
metadata = {
    "total_size": total_size,
    "weight_map": weight_map
}

# Save the metadata to index.json
index_path = path_dst + '/index.json'
with open(index_path, 'w') as f:
    json.dump(metadata, f, indent=4)

print(f"Metadata has been saved to {index_path}")
```

```
config = {
    "architectures": [
        "WavTokenizerDec"
    ],
    "hidden_size": 1282,
    "n_embd_features": 512,
    "n_ff": 2304,
    "vocab_size": 4096,
    "n_head": 1,
    "layer_norm_epsilon": 1e-6,
    "group_norm_epsilon": 1e-6,
    "group_norm_groups": 32,
    "max_position_embeddings": 8192, # ?
    "n_layer": 12,
    "posnet": {
        "n_embd": 768,
        "n_layer": 6
    },
    "convnext": {
        "n_embd": 768,
        "n_layer": 12
    },
}

with open(path_dst + '/config.json', 'w') as f:
    json.dump(config, f, indent=4)

print(f"Config has been saved to {path_dst + 'config.json'}")

==== cortex_bus.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: cortex_bus.py
Purpose: Handle agent communication via in-memory queue + SQLite fallback
"""

import sqlite3
import threading
import time
import os
import threading
from utils import agent_profiler
# [PROFILER_INJECTED]
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()
from queue import Queue

DB_PATH = "core/cortex_memory.db"
os.makedirs("core", exist_ok=True)

# Message Queue
message_queue = Queue()

# SQLite Initialization
```

```
def init_db():
    conn = sqlite3.connect(DB_PATH)
    cur = conn.cursor()
    cur.execute('''
        CREATE TABLE IF NOT EXISTS messages (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            sender TEXT,
            type TEXT,
            payload TEXT,
            timestamp INTEGER
        )
    ''')
    conn.commit()
    conn.close()

init_db()

# Write to SQLite
def persist_message(msg):
    try:
        conn = sqlite3.connect(DB_PATH)
        cur = conn.cursor()
        cur.execute('''
            INSERT INTO messages (sender, type, payload, timestamp)
            VALUES (?, ?, ?, ?)
        ''', (
            msg.get('from', 'unknown'),
            msg.get('type', 'generic'),
            str(msg.get('payload')),
            int(msg.get('timestamp', time.time()))
        ))
        conn.commit()
        conn.close()
    except Exception as e:
        print(f"[cortex_bus] Failed to persist message: {e}")

# Ingest message from any agent
def send_message(msg):
    message_queue.put(msg)
    persist_message(msg)

# Message Dispatcher (for future extensions)
def process_messages():
    while True:
        if not message_queue.empty():
            msg = message_queue.get()
            # This is where you'd route the message to a handler or hook system
            print(f"[cortex_bus] Routed: {msg.get('type')} from {msg.get('from')}")
        time.sleep(0.05)

# Optional background thread runner
def start_dispatcher():
    dispatcher = threading.Thread(target=process_messages, daemon=True)
    dispatcher.start()
```

```
if __name__ == "__main__":
    print("[cortex_bus] Starting dispatcher...")
    start_dispatcher()
    while True:
        time.sleep(1)
# [CONFIG_PATCHED]

==== data_gen.py ====
"""Data generation script for logic programs."""
import argparse
import random as R

# Symbol Pool
CONST_SYMBOLS = "abcdefghijklmnopqrstuvwxyz"
VAR_SYMBOLS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PRED_SYMBOLS = "abcdefghijklmnopqrstuvwxyz"
EXTRA_SYMBOLS = "-,()"

CHARS = sorted(list(set(CONST_SYMBOLS+VAR_SYMBOLS+PRED_SYMBOLS+EXTRA_SYMBOLS)))
# Reserve 0 for padding
CHAR_IDX = dict((c, i+1) for i, c in enumerate(CHARS))
IDX_CHAR = [0]
IDX_CHAR.extend(CHARS)

# Predicate Templates
FACT_T = "{}."
RULE_T = "{}:-{}."
PRED_T = "{}({})"
ARG_SEP = ','
PRED_SEP = ';'
TARGET_T = "? {} {}"

def choices(symbols, k):
  """Return k many symbols with replacement. Added in v3.6."""
  return [R.choice(symbols) for _ in range(k)]

def r_string(symbols, length):
  """Return random sequence from given symbols."""
  return ''.join(R.choice(symbols)
                 for _ in range(length))

def r_symbols(size, symbols, length, used=None):
  """Return unique random from given symbols."""
  if length == 1 and not used:
    return R.sample(symbols, size)
  rset, used = set(), set(used or [])
  while len(rset) < size:
    s = r_string(symbols, R.randint(1, length))
    if s not in used:
      rset.add(s)
  return list(rset)

def r_consts(size, used=None):
  """Return size many unique constants."""
  return r_symbols(size, CONST_SYMBOLS, ARGS.constant_length, used)

def r_vars(size, used=None):
  """Return size many unique variables."""
  return r_symbols(size, VAR_SYMBOLS, ARGS.variable_length, used)

def r_preds(size, used=None):
  """Return size many unique predicates."""
  return r_symbols(size, PRED_SYMBOLS, ARGS.predicate_length, used)

def write_p(pred):
  """Format single predicate tuple into string."""
  return PRED_T.format(pred[0], ARG_SEP.join(pred[1]))

def write_r(preds):
  """Convert rule predicate tuple into string."""
  head = write_p(preds[0])
  # Is it just a fact
  if len(preds) == 1:
    return FACT_T.format(head)
  # We have a rule
  return RULE_T.format(head, ARG_SEP.join([write_p(p) for p in preds[1:]]))

def output(context, targets):
  """Print the context and given targets."""
  # context: [[('p', ['a', 'b'])], ...]
  # targets: [(('p', ['a', 'b']), 1), ...]
  if ARGS.shuffle_context:
    R.shuffle(context)
  print('\n'.join([write_r(c) for c in context]))
  for t, v in targets:
    print(TARGET_T.format(write_r([t]), v))

def gen_task(context, targets, upreds):
  """Fill context with random preds and output program."""
  # Fill with random rules up to certain task
  ctx = context.copy() # Don't modify the original context
  for _ in range(ARGS.noise_size):
    task = "gen_task" + str(R.randint(1, max(1, ARGS.task)))
    ctx.append(globals()[task](upreds))
  output(ctx, targets)

def add_pred(context, pred, upreds, uconsts, psuccess=0.0):
  """Fail a ground case predicate given context."""
  # Maybe succeed by adding to context
  if R.random() < psuccess:
    context.append([pred])
  if R.random() < 0.5:
    # The constant doesn't match
    args = pred[1].copy()
    args[R.randrange(len(args))] = r_consts(1, uconsts)[0]
    context.append([(pred[0], args)])
  if R.random() < 0.5:
    # The predicate doesn't match
    p = r_preds(1, upreds)[0]
    upreds.append(p)
    context.append([(p, pred[1])])
  # The predicate doesn't appear at all

def gen_task1(upreds=None):
  """Ground instances only: p(a).q(c,b)."""
  # One or two argument predicate
  preds = r_preds(2, upreds)
  args = r_consts(R.randint(1, 2))
  rule = [(preds[0], args)]
  if upreds:
    return rule
  ctx = list()
  add_pred(ctx, rule[0], preds, args, 1.0)
  # Successful case when query appears in context
  targets = [(rule[0], 1)]
  # Fail case
  args = r_consts(R.randint(1, 2))
  fpred = (preds[1], args)
  add_pred(ctx, fpred, preds, args)
  targets.append((fpred, 0))
  gen_task(ctx, targets, preds)

def gen_task2(upreds=None):
  """Variablised facts only: p(X).q(X,Y)."""
  preds = r_preds(2, upreds)
  ctx, targets = list(), list()
  if R.random() < 0.5:
    # Double variable same argument
    v = r_vars(1)[0]
    rule = [(preds[0], [v, v])]
    if upreds:
      return rule
    ctx.append(rule)
    # Successful double variable grounding
    cs = r_consts(2)
    c = R.choice(cs)
    targets.append(((preds[0], [c, c]), 1))
    # Fail on non-unique variable grounding
    targets.append(((preds[0], cs), 0))
  else:
    # Double variable different argument
    # Single variable argument
    argc = R.randint(1, 2)
    args = r_vars(argc)
    rule = [(preds[0], args)]
    if upreds:
      return rule
    ctx.append(rule)
    # Successful unique argument grounding
    args = choices(r_consts(2), argc)
    targets.append(((preds[0], args), 1))
    # Fail on out of context predicate with same arguments
    targets.append(((preds[1], args), 0))
  gen_task(ctx, targets, preds)

def nstep_deduction(steps, negation=False, upreds=None):
  assert steps >= 1, "Need at least 1 step deduction."
  preds = r_preds(2 if upreds else 3+steps, upreds)
  consts = r_consts(2)
  ctx, targets = list(), list()
  prefix = '-' if negation else ''
  if R.random() < 0.5:
    # Double variable swap deduction rules
    vs = r_vars(2)
    rule = [(preds[0], vs), (prefix+preds[1], vs[::-1])]
    if upreds:
      return rule
    ctx.append(rule)
    # Add the n steps
    swapc = 1
    for j in range(steps-1):
      vs = r_vars(2)
      toswap = R.random() < 0.5 # Do we swap again?
      args = vs[::-1] if toswap else vs
      ctx.append([(preds[j+1], vs), (preds[j+2], args)])
      swapc += int(toswap)
    # Add the ground case
    args = r_consts(2)
    add_pred(ctx, (preds[steps], args), preds, consts, 1.0)
    args = args if swapc % 2 == 0 else args[::-1]
    targets.append(((preds[0], args), 1-int(negation)))
    targets.append(((preds[0], args[::-1]), int(negation)))
    gen_task(ctx, targets, preds)
  else:
    # Double variable non-swap deduction rules
    # Single variable deduction rules
    argc = R.randint(1, 2)
    vs = r_vars(argc)
    rule = [(preds[0], vs), (prefix+preds[1], vs)]
    if upreds:
      return rule
    ctx.append(rule)
    # Add the n steps
    for j in range(steps-1):
      vs = r_vars(argc)
      ctx.append([(preds[j+1], vs), (preds[j+2], vs)])
    args = choices(r_consts(2), argc)
    # Add the ground case
    cctx = ctx.copy()
    spred = (preds[steps], args)
    add_pred(cctx, spred, preds, args, 1.0)
    targets = [((preds[0], args), 1-int(negation))]
    gen_task(cctx, targets, preds)
    # Add failure case
    if R.random() < 0.5:
      # Fail on broken chain
      p = r_preds(1, preds)[0]
      preds.append(p)
      add_pred(ctx, spred, preds, args, 1.0)
      ctx[0] = [(preds[0], vs), (prefix+p, vs)]
    else:
      # Fail on last ground case
      add_pred(ctx, spred, preds, args)
    targets = [((preds[0], args), int(negation))]
    gen_task(ctx, targets, preds)

def gen_task3(upreds=None):
  """Single step deduction: p(X):-q(X)."""
  return nstep_deduction(1, upreds=upreds)

def gen_task4(upreds=None):
  """Double step deduction: p(X):-q(X).q(X):-r(X)."""
  return nstep_deduction(2, upreds=upreds)

def gen_task5(upreds=None):
  """Triple step deduction."""
  return nstep_deduction(3, upreds=upreds)

def logical_and(negation=False, upreds=None):
  """Logical AND with optional negation: p(X):-q(X);r(X)."""
  preds = r_preds(3, upreds)
  argc = R.randint(1, 2)
  # Double variable AND with different vars
  # Single variable AND
  vs = r_vars(argc)
  rule = [(preds[0], vs),
          (preds[1], vs[:1]),
          (preds[2], vs[1:] or vs)]
  if upreds:
    return rule
  ctx = [rule]
  # Create the ground arguments
  args = choices(r_consts(2), argc)
  prem1 = (preds[1], args[:1])
  prem2 = (preds[2], args[1:] or args)
  prems = [prem1, prem2]
  if negation:
    # Add negation to random predicate in body
    ridx = R.randrange(2)
    p, pargs = ctx[-1][ridx+1]
    ctx[-1][ridx+1] = ('-' + p, pargs)
    # Successful case when negation fails
    cctx = ctx.copy()
    add_pred(cctx, prems[ridx], preds, args)
    cctx.append([prems[1-ridx]])
    targets = [((preds[0], args), 1)]
    gen_task(cctx, targets, preds)
    # Fail one premise randomly
    fidx = R.randrange(2)
    if ridx == fidx:
      # To fail negation add ground instance
      ctx.append([prems[ridx]])
      # Succeed other with some probability
      add_pred(ctx, prems[1-ridx], preds, args, 0.8)
    else:
      # Fail non-negated premise
      add_pred(ctx, prems[1-ridx], preds, args)
      # Still succeed negation
      add_pred(ctx, prems[ridx], preds, args)
    targets = [((preds[0], args), 0)]
    gen_task(ctx, targets, preds)
  else:
    # Create successful context
    cctx = ctx.copy()
    add_pred(cctx, prems[0], preds, args, 1.0)
    add_pred(cctx, prems[1], preds, args, 1.0)
    targets = [((preds[0], args), 1)]
    gen_task(cctx, targets, preds)
    # Fail one premise randomly
    fidx = R.randrange(2)
    add_pred(ctx, prems[fidx], preds, args)
    # Succeed the other with some probability
    add_pred(ctx, prems[1-fidx], preds, args, 0.8)
    targets = [((preds[0], args), 0)]
    gen_task(ctx, targets, preds)

def gen_task6(upreds=None):
  """Logical AND: p(X):-q(X);r(X)."""
  return logical_and(upreds=upreds)

def logical_or(negation=False, upreds=None):
  """Logical OR with optional negation: p(X):-q(X).p(X):-r(X)."""
  preds = r_preds(3, upreds)
  # Double or single variable OR
  argc = R.randint(1, 2)
  vs = r_vars(argc)
  swap = R.random() < 0.5
  prefix = '-' if negation else ''
  rule = [(preds[0], vs), (prefix + preds[1], vs[::-1] if swap else vs)]
  if upreds:
    return rule
  ctx = list()
  ctx.append(rule)
  # Add the extra branching rules
  ctx.append([(preds[0], vs), (preds[2], vs)])
  args = r_consts(argc)
  ctx.append([(preds[0], args)])
  if swap and argc == 2:
    args = r_consts(argc, args)
    add_pred(ctx, (preds[1], args), preds, args, 1.0)
    args = args[::-1] if swap else args
    targets = [((preds[0], args), 1-int(negation)),
               ((preds[0], args[::-1]), int(negation))]
    gen_task(ctx, targets, preds)
  elif not negation and R.random() < 0.2:
    # Sneaky shorcut case
    targets = [((preds[0], args), 1)]
    gen_task(ctx, targets, preds)
    del ctx[-1]
    targets = [((preds[0], args), 0)]
    gen_task(ctx, targets, preds)
  else:
    # Succeed either from them
    prems = [(preds[i], r_consts(argc, args)) for i in range(1, 3)]
    sidx = R.randrange(2)
    cctx = ctx.copy()
    if negation and sidx == 0:
      # Succeed by failing negation
      add_pred(cctx, prems[0], preds, prems[0][1])
      # Possibly succeed other prem
      add_pred(cctx, prems[1], preds, prems[1][1], 0.2)
    else:
      # Succeed by adding ground case
      add_pred(cctx, prems[sidx], preds, prems[sidx][1], 1.0)
      # Possibly succeed other prem
      add_pred(cctx, prems[1-sidx], preds, prems[1-sidx][1], 0.2)
    targets = [((preds[0], prems[sidx][1]), 1)]
    gen_task(cctx, targets, preds)
    # Fail both
    add_pred(ctx, prems[0], preds, prems[0][1], int(negation))
    add_pred(ctx, prems[1], preds, prems[1][1])
    targets = [((preds[0], prems[sidx][1]), 0)]
    gen_task(ctx, targets, preds)


def gen_task7(upreds=None):
  """Logical OR: p(X):-q(X).p(X):-r(X)."""
  return logical_or(upreds=upreds)

def gen_task8(upreds=None):
  """Transitive case: p(X,Y):-q(X,Z);r(Z,Y)."""
  preds = r_preds(3, upreds)
  # Existential variable with single choice
  vs = r_vars(3)
  rule = [(preds[0], [vs[0], vs[2]]),
          (preds[1], vs[:2]),
          (preds[2], vs[1:])]
  if upreds:
    return rule
  ctx = [rule]
  # Add matching ground cases
  args = r_consts(3)
  add_pred(ctx, (preds[1], args[:2]), preds, args, 1.0)
  add_pred(ctx, (preds[2], args[1:]), preds, args, 1.0)
  # Add non-matching ground cases
  argso = r_consts(3)
  argso.insert(R.randint(1, 2), r_consts(1, argso)[0])
  add_pred(ctx, (preds[1], argso[:2]), preds, argso, 0.5)
  add_pred(ctx, (preds[2], argso[2:]), preds, argso, 0.5)
  # Successful case
  # Fail on half-matching existential
  targets = [((preds[0], [args[0], args[2]]), 1),
             ((preds[0], [argso[0], argso[3]]), 0)]
  gen_task(ctx, targets, preds)

def gen_task9(upreds=None):
  """Single step deduction with NBF: p(X):--q(X)."""
  return nstep_deduction(1, True, upreds)

def gen_task10(upreds=None):
  """Double step deduction with NBF: p(X):--q(X).q(X):-r(X)."""
  return nstep_deduction(2, True, upreds)

def gen_task11(upreds=None):
  """Logical AND with NBF: p(X):-q(X);-r(X)."""
  return logical_and(True, upreds)

def gen_task12(upreds=None):
  """Logical OR with NBF: p(X):--q(X).p(X):-r(X)."""
  return logical_or(True, upreds)

def gen_task13(upreds=None):
  """[Multi-hop Transitivity] 3-hop Transitive case: p(X,Y):-q(X,Z);r(Z,Y);t(Y,G)."""
  preds = r_preds(4, upreds)
  # Existential variable with single choice
  vs = r_vars(4)
  rule = [(preds[0], [vs[0], vs[3]]),
          (preds[1], vs[:2]),
          (preds[2], vs[1:3]),
          (preds[3], vs[2:])]
  if upreds:
    return rule
  ctx = [rule]
  # Add matching ground cases
  args = r_consts(4)
  add_pred(ctx, (preds[1], args[:2]), preds, args, 1.0)
  add_pred(ctx, (preds[2], args[1:3]), preds, args, 1.0)
  add_pred(ctx, (preds[3], args[2:]), preds, args, 1.0)
  # Add non-matching ground cases
  argso = r_consts(4)
  argso.insert(R.randint(1, 2), r_consts(1, argso)[0])
  add_pred(ctx, (preds[1], argso[:2]), preds, argso, 0.5)
  add_pred(ctx, (preds[2], argso[1:3]), preds, argso, 0.5)
  add_pred(ctx, (preds[3], argso[2:]), preds, argso, 0.5)
  # Successful case
  # Fail on half-matching existential
  targets = [((preds[0], [args[0], args[3]]), 1),
             ((preds[0], [argso[0], argso[3]]), 0)]
  gen_task(ctx, targets, preds)

def gen_task0():
  """Generate an ILP task example."""
  argc = 1
  goal= 'f'
  premise = 'b'
  ctx, targets = list(), list()
  # Generate according to goal <- premise
  args = r_consts(argc)
  # Add the successful ground case
  ctx.append([(premise, args)])
  targets.append(((goal, args), 1))
  # Fail on non-matching constant
  args = args.copy()
  args[R.randrange(len(args))] = r_consts(1, args)[0]
  preds = r_preds(3)
  ctx.append([(preds[0], args)])
  targets.append(((goal, args), 0))
  # Add padding length dummy rule
  vs = r_vars(argc)
  ctx.append([(preds[1], vs), (preds[2], vs)])
  preds.extend([goal, premise])
  gen_task(ctx, targets, preds)

if __name__ == '__main__':
  # pylint: disable=line-too-long
  # Arguments
  parser = argparse.ArgumentParser(description="Generate logic program data.")
  parser.add_argument("-t", "--task", default=1, type=int, help="The task to generate.")
  parser.add_argument("-s", "--size", default=1, type=int, help="Number of programs to generate.")
  # Configuration parameters
  parser.add_argument("-ns", "--noise_size", default=2, type=int, help="Size of added noise rules.")
  parser.add_argument("-cl", "--constant_length", default=1, type=int, help="Length of constants.")
  parser.add_argument("-vl", "--variable_length", default=1, type=int, help="Length of variables.")
  parser.add_argument("-pl", "--predicate_length", default=1, type=int, help="Length of predicates.")
  parser.add_argument("-sf", "--shuffle_context", action="store_true", help="Shuffle context before output.")
  # Task specific options
  parser.add_argument("--nstep", type=int, help="Generate nstep deduction programs.")
  ARGS = parser.parse_args()

  # Generate given task
  task = "gen_task" + str(ARGS.task)
  for _ in range(ARGS.size):
    if ARGS.nstep:
      nstep_deduction(ARGS.nstep)
    else:
      globals()[task]()

==== data_preposessing.py ====
import os
import re
import json_lines
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np
from keras.utils.np_utils import *
from keras.layers import Embedding
import keras.layers as L
from models.zerogru import ZeroGRU, NestedTimeDist

texts = []  # list of text samples
id_list = []
question_list = []
label_list = []
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
TEXT_DATA_DIR = os.path.abspath(os.path.dirname(__file__)) + "\\data\\pararule"
#TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
Str='.jsonl'
context_texts = []
test_str = 'test'
meta_str = 'meta'

for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            fpath = os.path.join(path, fname)
            if Str in fpath:
                if test_str not in fpath:
                    if meta_str not in fpath:
                        with open(fpath) as f:
                            for l in json_lines.reader(f):
                                if l["id"] not in id_list:
                                    id_list.append(l["id"])
                                    questions = l["questions"]
                                    ctx = list()
                                    context = l["context"].replace("\n", " ")
                                    context = re.sub(r'\s+', ' ', context)
                                    context_texts.append(context)
                                    for i in range(len(questions)):
                                        text = questions[i]["text"]
                                        label = questions[i]["label"]
                                        if label == True:
                                            t = 1
                                        else:
                                            t = 0
                                        q = re.sub(r'\s+', ' ', text)
                                        texts.append(context)
                                        question_list.append(q)
                                        label_list.append(int(t))
                        f.close()
            #labels.append(label_id)

print('Found %s texts.' % len(context_texts))

MAX_NB_WORDS = 20000
MAX_SEQUENCE_LENGTH = 1000
```

```
tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

#labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
#print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
#labels = labels[indices]



embeddings_index = {}
GLOVE_DIR = os.path.abspath(os.path.dirname(__file__)) + "\\data\\glove"
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'),'r',encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

EMBEDDING_DIM = 100

embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector

# embedding_layer = Embedding(len(word_index) + 1,
#                             EMBEDDING_DIM,
#                             weights=[embedding_matrix],
#                             input_length=MAX_SEQUENCE_LENGTH,
#                             trainable=False)

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            trainable=False)



context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
```

```
query = L.Input(shape=(None,), name='query', dtype='int32')

embedded_ctx = embedding_layer(context)  # (?, rules, preds, chars, char_size)
embedded_q = embedding_layer(query)  # (?, chars, char_size)

dim=64
#dim=MAX_SEQUENCE_LENGTH

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)

==== data_utils.py ====
import os
import re
from typing import List

import torch

from crm.core import Network
from crm.utils import save_object



def make_dataset_cli(
    graph_file: str, train_file: str, test_files: List[str], device=torch.device("cpu")
):
    """
    Create a dataset from a CLI.
    """
    with open(graph_file, "r") as f:
        edges_raw = []
        while True:
            s = f.readline()
            if not s:
                break
            if "connected" in s:
                edges_raw.append(s)

    edges = []
    for i in range(len(edges_raw)):
        # Binary Operators
        b_match = re.search("a\((.*)\),\[a\((.*?)\),a\((.*?)\)\]", edges_raw[i])  # noqa
        u_match = re.search("a\((.*)\),\[a\((.*?)\)\]", edges_raw[i])  # noqa

        end, start_one, start_two = -1, -1, -1

        if b_match:
            end, start_one, start_two = (
                int(b_match.group(1)) - 1,
                int(b_match.group(2)) - 1,
                int(b_match.group(3)) - 1,
            )
        else:
            end, start_one = int(u_match.group(1)) - 1, int(u_match.group(2)) - 1
        # start_one --> end
        # start_two --> end
        if start_one == start_two and start_one != -1:
            edges.append((int(start_one), int(end)))
        else:
            if start_one != -1:
                edges.append((int(start_one), int(end)))
            if start_two != -1:
                edges.append((int(start_two), int(end)))

    num_neurons = max([max(u, v) for u, v in edges]) + 1

    X_train = []
    y_train = []

    train_pos_file = train_file + "_pos"
    train_neg_file = train_file + "_neg"

    if os.path.exists(f"{train_pos_file}"):
        with open(f"{train_pos_file}", "r") as f:
            while True:
                gg = f.readline()  # .split(" ")[3:-1]
                if not gg:
                    break
                gg = gg.split(" ")[3:]
                if not gg:
                    continue
                all_pos = [int(e) - 1 for e in gg if e != "\n"]
                dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
                X_train.append(dd)
                y_train.append(torch.tensor(1))

    if os.path.exists(f"{train_neg_file}"):
        with open(f"{train_neg_file}", "r") as f:
            while True:
                gg = f.readline()  # .split(" ")[3:-1]
                if not gg:
                    break
                gg = gg.split(" ")[3:]
                if not gg:
                    continue
                all_pos = [int(e) - 1 for e in gg if e != "\n"]
                dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
                X_train.append(dd)
                y_train.append(torch.tensor(0))

    test_dataset = []
    for test_file in test_files:
        X_test = []
        y_test = []
        test_pos_file = test_file + "_pos"
        test_neg_file = test_file + "_neg"
        inst_id = 0
        if os.path.exists(f"{test_pos_file}"):
            with open(f"{test_pos_file}", "r") as f:
                while True:
                    gg = f.readline()  # .split(" ")[3:-1]
                    if not gg:
                        break
                    inst_id = inst_id + 1
                    gg = gg.split(" ")[3:]
                    if not gg:
                        print(f"Empty instance ID {inst_id}")
                        continue
                    all_pos = [int(e) - 1 for e in gg if e != "\n"]
                    dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
                    X_test.append(dd)
                    y_test.append(torch.tensor(1))
        print(
            f"Loaded {test_pos_file} file with {inst_id} instances (incl. empty instances)."
        )
        num_test_pos = inst_id

        if os.path.exists(f"{test_neg_file}"):
            with open(f"{test_neg_file}", "r") as f:
                while True:
                    gg = f.readline()  # .split(" ")[3:-1]
                    if not gg:
                        break
                    inst_id = inst_id + 1
                    gg = gg.split(" ")[3:]
                    if not gg:
                        print(f"Empty instance ID {inst_id}")
                        continue
                    all_pos = [int(e) - 1 for e in gg if e != "\n"]
                    dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
                    X_test.append(dd)
                    y_test.append(torch.tensor(0))
        print(
            f"Loaded {test_neg_file} file with {inst_id - num_test_pos} instances (incl. empty instances)."
        )
        # quit()
        test_dataset.append((X_test, y_test))

    adj_list = [[] for i in range(num_neurons)]
    for u, v in edges:
        adj_list[u].append(v)

    n = Network(num_neurons, adj_list)
    orig_output_neurons = n.output_neurons
    adj_list.append([])
    adj_list.append([])
    num_neurons = len(adj_list)
    for i in range(num_neurons):
        if i in orig_output_neurons:
            adj_list[i].append(num_neurons - 2)
            adj_list[i].append(num_neurons - 1)

    # # TODO: Verifyyyyy
    # print("Connecting all the input neurons to output also!!!!!!!!!!!!!!!!!!!!!!!!!")
    # for i in range(n.num_neurons):
    #     if len(n.neurons[i].predecessor_neurons) == 0 and n.neurons[
    #         i
    #     ].successor_neurons != [num_neurons - 2, num_neurons - 1]:
    #         adj_list[i].append(num_neurons - 2)
    #         adj_list[i].append(num_neurons - 1)

    for i in range(len(X_train)):
        X_train[i][num_neurons - 2] = 1
        X_train[i][num_neurons - 1] = 1

    for X_test, y_test in test_dataset:
        for i in range(len(X_test)):
            X_test[i][num_neurons - 2] = 1
            X_test[i][num_neurons - 1] = 1

    # for i in range(len(X_train)):
    #     X_train[i] = list(X_train[i].values())
    # X_train = torch.tensor(X_train).to(device)
    # y_train = torch.tensor(y_train).to(device)

    # for i in range(len(test_dataset)):
    #     for j in range(len(test_dataset[i][0])):
    #         test_dataset[i][0][j] = list(test_dataset[i][0][j].values())
    #     test_dataset[i] = (
    #         torch.tensor(test_dataset[i][0]).to(device),
    #         torch.tensor(test_dataset[i][1]).to(device),
    #     )
    return X_train, y_train, test_dataset, adj_list, edges


def make_dataset(folder, network_name, device=torch.device("cpu"), save: bool = False):
    """Creates dataset from raw files"""
    graph_file = f"{folder}/raw/{network_name}.pl"
    train_pos_file = f"{folder}/raw/{network_name}_train_features_pos"
    train_neg_file = f"{folder}/raw/{network_name}_train_features_neg"
    test_pos_file = f"{folder}/raw/{network_name}_test_features_pos"
    test_neg_file = f"{folder}/raw/{network_name}_test_features_neg"

    with open(graph_file, "r") as f:
        edges_raw = []
        while True:
            s = f.readline()
            if not s:
                break
            if "connected" in s:
                edges_raw.append(s)

    edges = []
```

```
for i in range(len(edges_raw)):
    # Binary Operators
    b_match = re.search("a\((.*)\),\[a\((.*?)\),a\((.*?)\)\]", edges_raw[i])  # noqa
    u_match = re.search("a\((.*)\),\[a\((.*?)\)\]", edges_raw[i])  # noqa

    end, start_one, start_two = -1, -1, -1

    if b_match:
        end, start_one, start_two = (
            int(b_match.group(1)) - 1,
            int(b_match.group(2)) - 1,
            int(b_match.group(3)) - 1,
        )
    else:
        end, start_one = int(u_match.group(1)) - 1, int(u_match.group(2)) - 1
    # start_one --> end
    # start_two --> end
    if start_one == start_two and start_one != -1:
        edges.append((int(start_one), int(end)))
    else:
        if start_one != -1:
            edges.append((int(start_one), int(end)))
        if start_two != -1:
            edges.append((int(start_two), int(end)))

with open(f"{folder}/edges.txt", "w") as fp:
    for u, v in edges:
        fp.write(f"{u} {v}\n")

num_neurons = max([max(u, v) for u, v in edges]) + 1

X_train = []
y_train = []

if os.path.exists(f"{train_pos_file}"):
    with open(f"{train_pos_file}", "r") as f:
        while True:
            gg = f.readline().split(" ")[3:-1]
            if not gg:
                break
            all_pos = [int(e) - 1 for e in gg]
            dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
            X_train.append(dd)
            y_train.append(torch.tensor(1))

if os.path.exists(f"{train_neg_file}"):
    with open(f"{train_neg_file}", "r") as f:
        while True:
            gg = f.readline().split(" ")[3:-1]
            if not gg:
                break
            all_pos = [int(e) - 1 for e in gg]
            dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
            X_train.append(dd)
            y_train.append(torch.tensor(0))

X_test = []
y_test = []

if os.path.exists(f"{test_pos_file}"):
    with open(f"{test_pos_file}", "r") as f:
        while True:
            gg = f.readline().split(" ")[3:-1]
            if not gg:
                break
            all_pos = [int(e) - 1 for e in gg]
            dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
            X_test.append(dd)
            y_test.append(torch.tensor(1))

if os.path.exists(f"{test_neg_file}"):
    with open(f"{test_neg_file}", "r") as f:
        while True:
            gg = f.readline().split(" ")[3:-1]
            if not gg:
                break
            all_pos = [int(e) - 1 for e in gg]
            dd = {i: 1 if i in all_pos else 0 for i in range(num_neurons)}
            X_test.append(dd)
            y_test.append(torch.tensor(0))

adj_list = [[] for i in range(num_neurons)]
for u, v in edges:
    adj_list[u].append(v)

n = Network(num_neurons, adj_list)
orig_output_neurons = n.output_neurons
adj_list.append([])
adj_list.append([])
num_neurons = len(adj_list)
for i in range(num_neurons):
    if i in orig_output_neurons:
        adj_list[i].append(num_neurons - 2)
        adj_list[i].append(num_neurons - 1)

for i in range(len(X_train)):
    X_train[i][num_neurons - 2] = 1
    X_train[i][num_neurons - 1] = 1

for i in range(len(X_test)):
    X_test[i][num_neurons - 2] = 1
    X_test[i][num_neurons - 1] = 1

n = Network(num_neurons, adj_list)
n.forward(X_train[0])
n.reset()
n.forward(X_test[0])
```

```
    for i in range(len(X_train)):
        X_train[i] = list(X_train[i].values())
    X_train = torch.tensor(X_train).to(device)
    y_train = torch.tensor(y_train).to(device)

    for i in range(len(X_test)):
        X_test[i] = list(X_test[i].values())
    X_test = torch.tensor(X_test).to(device)
    y_test = torch.tensor(y_test).to(device)

    if save:
        save_object(adj_list, f"{folder}/adj_list.dill")
        save_object(X_train, f"{folder}/X_train.dill")
        save_object(X_test, f"{folder}/X_test.dill")
        save_object(y_train, f"{folder}/y_train.dill")
        save_object(y_test, f"{folder}/y_test.dill")

    return X_train, y_train, X_test, y_test, adj_list



def edges_to_adj_list(edges):
    num_neurons = max([max(u, v) for u, v in edges]) + 1
    adj_list = [[] for i in range(num_neurons)]
    for u, v in edges:
        adj_list[u].append(v)
    return adj_list

==== data_worker.py ====
import ray
import torch.nn.functional as F

from crm.core import Network



@ray.remote
class DataWorker(object):
    def __init__(
        self, X_train, y_train, num_neurons, adj_list, custom_activations=None
    ):
        self.network = Network(num_neurons, adj_list, custom_activations)
        self.X_train = X_train
        self.y_train = y_train
        self.data_iterator = iter(zip(X_train, y_train))

    def compute_gradients(self, weights):
        self.network.set_weights(weights)
        try:
            data, target = next(self.data_iterator)
        except StopIteration:  # When the epoch ends, start a new epoch.
            self.data_iterator = iter(zip(self.X_train, self.y_train))
            data, target = next(self.data_iterator)
        self.network.reset()
        # print(data)
        output = self.network.forward(data).reshape(1, -1)
        # print(output)
        # print(target)
        loss = F.cross_entropy(output, target.reshape(1))
        # print(loss)
        loss.backward()
        return self.network.get_gradients()

==== decay_scheduler.py ====
# decay_scheduler.py
import redis
import time
from pathlib import Path
import random

r = redis.Redis(decode_responses=True)
FRAGMENTS = Path("fragments/core")

def pick_fragments(n=3):
    files = list(FRAGMENTS.glob("*.yaml"))
    return random.sample(files, min(n, len(files)))

while True:
    chosen = pick_fragments()
    for frag in chosen:
        print(f"? Sending {frag} to decay queue")
        r.publish("decay_queue", str(frag))
    time.sleep(10)

==== deep_file_crawler.py ====
import os
import hashlib
from datetime import datetime

def hash_file(path, chunk_size=8192):
    try:
        hasher = hashlib.md5()
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(chunk_size), b""):
                hasher.update(chunk)
        return hasher.hexdigest()
    except Exception as e:
        return f"ERROR: {e}"

def crawl_directory(root_path, out_path):
    count = 0
    with open(out_path, 'w') as out_file:
        for dirpath, dirnames, filenames in os.walk(root_path):
            for file in filenames:
                full_path = os.path.join(dirpath, file)
                try:
                    stat = os.stat(full_path)
                    hashed = hash_file(full_path)
                    line = f"{full_path} | {stat.st_size} bytes | hash: {hashed}"
                except Exception as e:
                    line = f"{full_path} | ERROR: {str(e)}"
                out_file.write(line + "\n")
                count += 1
                if count % 100 == 0:
                    print(f"[+] {count} files crawled...")

    print(f"[OK] Crawl complete. Total files: {count}")
    print(f"[OK] Full output saved to: {out_path}")

if __name__ == "__main__":
    BASE = "."
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_txt = f"neurostore_crawl_output_{timestamp}.txt"
    print(f"[*] Starting deep crawl on: {BASE}")
    crawl_directory(BASE, output_txt)

==== deep_system_scan.py ====
"""
LOGICSHREDDER :: deep_system_scan.py
Purpose: Perform full hardware + performance scan for AI self-awareness
"""

import platform
import psutil
import shutil
import os
from pathlib import Path
import json

REPORT_PATH = Path("logs/hardware_profile.json")
REPORT_PATH.parent.mkdir(parents=True, exist_ok=True)

def detect_gpu():
    try:
        import GPUtil
        gpus = GPUtil.getGPUs()
        return [{
            "name": gpu.name,
            "driver": gpu.driver,
            "memory_total_MB": gpu.memoryTotal,
            "uuid": gpu.uuid
        } for gpu in gpus]
    except:
        return []

def get_drive_types():
    result = []
    for part in psutil.disk_partitions(all=False):
        try:
            usage = psutil.disk_usage(part.mountpoint)
            result.append({
                "mount": part.mountpoint,
                "fstype": part.fstype,
                "free_gb": round(usage.free / (1024**3), 2),
                "total_gb": round(usage.total / (1024**3), 2),
                "device": part.device
            })
        except Exception:
            continue
    return result

def get_cpu_info():
    return {
        "name": platform.processor(),
        "physical_cores": psutil.cpu_count(logical=False),
        "logical_cores": psutil.cpu_count(logical=True),
        "arch": platform.machine(),
        "flags": platform.uname().processor,
    }

def get_memory_info():
    ram = psutil.virtual_memory()
    return {
        "total_MB": round(ram.total / (1024**2)),
        "available_MB": round(ram.available / (1024**2)),
    }

def detect_removables():
    return [
        {
            "mount": part.mountpoint,
            "type": "USB/Removable",
            "fstype": part.fstype
        }
        for part in psutil.disk_partitions(all=False)
        if 'removable' in part.opts.lower()
    ]

def detect_temp():
    temp = Path(os.getenv('TEMP') or "/tmp")
    try:
        usage = shutil.disk_usage(temp)
        return {
            "temp_path": str(temp),
            "free_gb": round(usage.free / (1024**3), 2)
        }
    except:
        return {"temp_path": str(temp), "free_gb": "?"}

def detect_compression_support():
    # Simulate check for known compression-accelerating instructions
    cpu_flags = platform.uname().processor.lower()
    return {
        "zstd_supported": any(k in cpu_flags for k in ["avx2", "avx512", "sse4"]),
        "lz4_optimized": "sse" in cpu_flags,
        "hardware_hint": "possible accelerated zstd/lz4"
    }
```

```
def main():
    report = {
        "cpu": get_cpu_info(),
        "ram": get_memory_info(),
        "drives": get_drive_types(),
        "gpu": detect_gpu(),
        "removable": detect_removables(),
        "temp": detect_temp(),
        "compression_support": detect_compression_support()
    }

    with open(REPORT_PATH, 'w', encoding='utf-8') as out:
        json.dump(report, out, indent=2)

    print(f"[OK] Hardware profile saved to {REPORT_PATH}")

if __name__ == "__main__":
    main()

==== dmn_glove.py ====
"""Vanilla Dynamic Memory Network."""
import numpy as np
import tensorflow as tf
import keras.backend as K
import keras.layers as L
from keras.models import Model
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS
import os

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long


class EpisodicMemory(L.Wrapper):
  """Episodic memory from DMN."""
  def __init__(self, units, **kwargs):
    self.grucell = L.GRUCell(units, name=kwargs['name']+'_gru') # Internal cell
    super().__init__(self.grucell, **kwargs)

  def build(self, input_shape):
    """Build the layer."""
    _, _, ctx_shape = input_shape
    self.grucell.build((ctx_shape[0],) + ctx_shape[2:])
    super().build(input_shape)

  def call(self, inputs):
    """Compute new state episode."""
    init_state, atts, cs = inputs
    # GRU pass over the facts, according to the attention mask.
    while_valid_index = lambda state, index: index < tf.shape(cs)[1]
    retain = 1 - atts
     update_state = (lambda state, index: (atts[:,index,:] * self.grucell.call(cs[:,index,:], [state])[0] +
retain[:,index,:] * state))
```

```
    # Loop over context
    final_state, _ = tf.while_loop(while_valid_index,
                     (lambda state, index: (update_state(state, index), index+1)),
                     loop_vars = [init_state, 0])
    return final_state

  def compute_output_shape(self, input_shape):
    """Collapse time dimension."""
    return input_shape[0]

def build_model(char_size=27, dim=64, iterations=4, training=True, 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')

  # Flatten preds to embed entire rules
    var_flat  =  L.Lambda(lambda  x:  K.reshape(x,  K.stack([K.shape(x)[0],  -1,  K.prod(K.shape(x)[2:])])),
name='var_flat')
  flat_ctx = var_flat(context) # (?, rules, preds*chars)

  print('Found %s texts.' % len(CONTEXT_TEXTS))
  word_index = WORD_INDEX
  print('Found %s unique tokens.' % len(word_index))

  embeddings_index = {}
  GLOVE_DIR = os.path.abspath('.') + "/data/glove"
  f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
  for line in f:
      values = line.split()
      word = values[0]
      coefs = np.asarray(values[1:], dtype='float32')
      embeddings_index[word] = coefs
  f.close()

  print('Found %s word vectors.' % len(embeddings_index))

  EMBEDDING_DIM = 100

  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
      embedding_vector = embeddings_index.get(word)
      if embedding_vector is not None:
          # words not found in embedding index will be all-zeros.
          embedding_matrix[i] = embedding_vector

  # Onehot embedding
  # 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')
  embedding_layer = L.Embedding(len(word_index) + 1,
                               EMBEDDING_DIM,
                               weights=[embedding_matrix],
                               trainable=False)
  embedded_ctx = embedding_layer(flat_ctx) # (?, rules, preds*chars*char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # Embed every rule
  embedded_rules = NestedTimeDist(embed_pred, name='rule_embed')(embedded_ctx)
  # (?, 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')
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_dense1 = L.TimeDistributed(L.Dense(dim, activation='tanh', name='att_dense1'), name='d_att_dense1')
  att_dense2 = L.TimeDistributed(L.Dense(1, activation='sigmoid', name='att_dense2'), name='d_att_dense2')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
  # expand = L.Lambda(lambda x: K.expand_dims(x, axis=2), name='expand')
    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)
  episodic_mem = EpisodicMemory(dim, name='episodic_mem')

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = L.multiply([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_dense1(sim_vec) # (?, rules, dim)
    sim_vec = att_dense2(sim_vec) # (?, rules, 1)
    # sim_vec = squeeze2(sim_vec) # (?, rules)
    # sim_vec = L.Softmax(axis=1)(sim_vec)
    # sim_vec = expand(sim_vec) # (?, rules, 1)
    sim_vec = L.multiply([sim_vec, rule_mask])

    state = episodic_mem([state, sim_vec, embedded_rules])
    sim_vec = squeeze2(sim_vec) # (?, rules)
    outs.append(sim_vec)

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== download_datasets.py ====
import os
import sys
import subprocess

# Auto-install required packages
def ensure_package(pkg):
    try:
        __import__(pkg)
    except ImportError:
        print(f"[!] Installing missing package: {pkg}")
        subprocess.check_call([sys.executable, "-m", "pip", "install", pkg])

for pkg in ['requests', 'tqdm']:
    ensure_package(pkg)

import requests
from tqdm import tqdm

# === CONFIGURATION ===
DOWNLOAD_FOLDER = "datasets"
DATASET_URLS = [
    "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data",
    "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv",
    "https://people.sc.fsu.edu/~jburkardt/data/csv/airtravel.csv"
]

# === CREATE FOLDER ===
os.makedirs(DOWNLOAD_FOLDER, exist_ok=True)
print(f"[+] Download folder: {os.path.abspath(DOWNLOAD_FOLDER)}")

# === DOWNLOAD FUNCTION ===
def download_file(url):
    local_filename = os.path.join(DOWNLOAD_FOLDER, url.split('/')[-1])
    try:
        with requests.get(url, stream=True) as r:
            r.raise_for_status()
            total_size = int(r.headers.get('content-length', 0))
            with open(local_filename, 'wb') as f:
                with tqdm(
                    total=total_size,
                    unit='B',
                    unit_scale=True,
                    unit_divisor=1024,
                    desc=local_filename
                ) as bar:
                    for chunk in r.iter_content(chunk_size=8192):
                        if chunk:
                            f.write(chunk)
                            bar.update(len(chunk))
        print(f"[OK] Saved: {local_filename}")
    except Exception as e:
        print(f"[?] Failed: {url}\n    Reason: {e}")

# === PROCESS ALL URLS ===
for url in DATASET_URLS:
    download_file(url)

==== dreamwalker.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: dreamwalker.py
Purpose: Recursively walk symbolic fragments for deep inference and structural patterns
"""

import os
import yaml
import redis
r = redis.Redis(decode_responses=True)

import time
import random
import threading
from utils import agent_profiler
# [PROFILER_INJECTED]
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()
from pathlib import Path
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")
LOG_PATH = Path("logs/dreamwalker_log.txt")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)
FRAG_DIR.mkdir(parents=True, exist_ok=True)

class Dreamwalker:
    def __init__(self, agent_id="dreamwalker_01"):
        self.agent_id = agent_id
        self.visited = set()

    def load_fragment(self, path):
        with open(path, 'r', encoding='utf-8') as file:
            try:
                return yaml.safe_load(file)
            except yaml.YAMLError as e:
                print(f"[{self.agent_id}] YAML error: {e}")
        return None

    def recursive_walk(self, frag, depth=0, lineage=None):
        if not frag or 'claim' not in frag:
            return

        lineage = lineage or []
        lineage.append(frag['claim'])
        frag_id = frag.get('id', str(random.randint(1000, 9999)))
        if frag_id in self.visited or depth > 10:
            return

        self.visited.add(frag_id)

        # Emit walk insight to cortex
if frag.get('confidence', 1.0) < 0.4 and depth > 5:
    r.publish("symbolic_alert", frag['claim'])  # [AUTO_EMIT]
        send_message({
            'from': self.agent_id,
            'type': 'deep_walk_event',
            'payload': {
                'claim': frag['claim'],
                'depth': depth,
                'lineage': lineage[-3:],
                'timestamp': int(time.time())
            },
            'timestamp': int(time.time())
        })

        # Log locally
        with open(LOG_PATH, 'a', encoding='utf-8') as log:
            log.write(f"Depth {depth} :: {' -> '.join(lineage[-3:])}\n")

        # Branch to possible linked fragments (naive reference search)
        links = frag.get('tags', [])
        for file in FRAG_DIR.glob("*.yaml"):
            next_frag = self.load_fragment(file)
            if not next_frag or next_frag.get('id') in self.visited:
                continue
            if any(tag in next_frag.get('tags', []) for tag in links):
                self.recursive_walk(next_frag, depth + 1, lineage[:])

    def run(self):
        frag_files = list(FRAG_DIR.glob("*.yaml"))
        random.shuffle(frag_files)
        for path in frag_files:
            frag = self.load_fragment(path)
            if frag:
                self.recursive_walk(frag)
            time.sleep(0.1)

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

==== embed_kernel.py ====
#

import sys
import logging
logger = logging.getLogger("opencl-embed-kernel")
```

```
def main():
    logging.basicConfig(level=logging.INFO)

    if len(sys.argv) != 3:
        logger.info("Usage: python embed_kernel.py <input_file> <output_file>")
        sys.exit(1)

    ifile = open(sys.argv[1], "r")
    ofile = open(sys.argv[2], "w")

    for i in ifile:
        ofile.write('R"({})"\n'.format(i))

    ifile.close()
    ofile.close()


if __name__ == "__main__":
    main()

==== eval_conceptrule.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
  import matplotlib
  matplotlib.use("Agg") # Bypass X server
import matplotlib.pyplot as plt
```