```
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output



# --------- Parameter Server --------------------
class ParameterServer(nn.Module):
    def __init__(self, num_gpus=0):
        super().__init__()
        self.num_gpus = num_gpus
        self.models = {}
        # This lock is only used during init, and does not
        # impact training perf.
        self.models_init_lock = Lock()
        self.input_device = torch.device(
            "cuda:0" if torch.cuda.is_available() and num_gpus > 0 else "cpu"
        )

    def forward(self, rank, inp):
        inp = inp.to(self.input_device)
        out = self.models[rank](inp)
        # This output is forwarded over RPC, which as of 1.5.0 only accepts CPU tensors.
        # Tensors must be moved in and out of GPU memory due to this.
        out = out.to("cpu")
        return out

    # Use dist autograd to retrieve gradients accumulated for this model.
    # Primarily used for verification.
    def get_dist_gradients(self, cid):
        grads = dist_autograd.get_gradients(cid)
        # This output is forwarded over RPC, which as of 1.5.0 only accepts CPU tensors.
        # Tensors must be moved in and out of GPU memory due to this.
        cpu_grads = {}
        for k, v in grads.items():
            k_cpu, v_cpu = k.to("cpu"), v.to("cpu")
            cpu_grads[k_cpu] = v_cpu
        return cpu_grads

    # Wrap local parameters in a RRef. Needed for building the
    # DistributedOptimizer which optimizes parameters remotely.
    def get_param_rrefs(self, rank):
        param_rrefs = [rpc.RRef(param) for param in self.models[rank].parameters()]
        return param_rrefs

    def create_model_for_rank(self, rank, num_gpus):
        assert (
            num_gpus == self.num_gpus
        ), f"Inconsistent no. of GPUs requested from rank vs initialized with on PS: {num_gpus} vs
{self.num_gpus}"
        with self.models_init_lock:
            if rank not in self.models:
                self.models[rank] = Net(num_gpus=num_gpus)
    def get_num_models(self):
        with self.models_init_lock:
            return len(self.models)

    def average_models(self, rank):
        # Load state dict of requested rank
        state_dict_for_rank = self.models[rank].state_dict()
        # Average all params
        for key in state_dict_for_rank:
            state_dict_for_rank[key] = sum(
                self.models[r].state_dict()[key] for r in self.models
            ) / len(self.models)
        # Rewrite back state dict
        self.models[rank].load_state_dict(state_dict_for_rank)


param_server = None
global_lock = Lock()


def get_parameter_server(rank, num_gpus=0):
    global param_server
    # Ensure that we get only one handle to the ParameterServer.
    with global_lock:
        if not param_server:
            # construct it once
            param_server = ParameterServer(num_gpus=num_gpus)
        # Add model for this rank
        param_server.create_model_for_rank(rank, num_gpus)
        return param_server


def run_parameter_server(rank, world_size):
    # The parameter server just acts as a host for the model and responds to
    # requests from trainers, hence it does not need to run a loop.
    # rpc.shutdown() will wait for all workers to complete by default, which
    # in this case means that the parameter server will wait for all trainers
    # to complete, and then exit.
    logger.info("PS master initializing RPC")
    rpc.init_rpc(name="parameter_server", rank=rank, world_size=world_size)
    logger.info("RPC initialized! Running parameter server...")
    rpc.shutdown()
    logger.info("RPC shutdown on parameter server.")


# --------- Trainers --------------------


# nn.Module corresponding to the network trained by this trainer. The
# forward() method simply invokes the network on the given parameter
# server.
class TrainerNet(nn.Module):
    def __init__(
        self,
        rank,
        num_gpus=0,
    ):
        super().__init__()
        self.num_gpus = num_gpus
        self.rank = rank
        self.param_server_rref = rpc.remote(
            "parameter_server",
            get_parameter_server,
            args=(
                self.rank,
                num_gpus,
            ),
        )

    def get_global_param_rrefs(self):
        remote_params = self.param_server_rref.rpc_sync().get_param_rrefs(self.rank)
        return remote_params

    def forward(self, x):
        model_output = self.param_server_rref.rpc_sync().forward(self.rank, x)
        return model_output

    def average_model_across_trainers(self):
        self.param_server_rref.rpc_sync().average_models(self.rank)


def run_training_loop(rank, world_size, num_gpus, train_loader, test_loader):
    # Runs the typical neural network forward + backward + optimizer step, but
    # in a distributed fashion.
    net = TrainerNet(rank=rank, num_gpus=num_gpus)
    # Wait for all nets on PS to be created, otherwise we could run
    # into race conditions during training.
    num_created = net.param_server_rref.rpc_sync().get_num_models()
    while num_created != world_size - 1:
        time.sleep(0.5)
        num_created = net.param_server_rref.rpc_sync().get_num_models()

    # Build DistributedOptimizer.
    param_rrefs = net.get_global_param_rrefs()
    opt = DistributedOptimizer(optim.SGD, param_rrefs, lr=0.03)
    for i, (data, target) in enumerate(train_loader):
        if TERMINATE_AT_ITER is not None and i == TERMINATE_AT_ITER:
            break
        if i % PS_AVERAGE_EVERY_N == 0:
            # Request server to update model with average params across all
            # trainers.
            logger.info(f"Rank {rank} averaging model across all trainers.")
            net.average_model_across_trainers()
        with dist_autograd.context() as cid:
            model_output = net(data)
            target = target.to(model_output.device)
            loss = F.nll_loss(model_output, target)
            if i % TRAINER_LOG_INTERVAL == 0:
                logger.info(f"Rank {rank} training batch {i} loss {loss.item()}")
            dist_autograd.backward(cid, [loss])
            # Ensure that dist autograd ran successfully and gradients were
            # returned.
            assert net.param_server_rref.rpc_sync().get_dist_gradients(cid) != {}
            opt.step(cid)

    logger.info("Training complete!")
    logger.info("Getting accuracy....")
    get_accuracy(test_loader, net)



def get_accuracy(test_loader, model):
    model.eval()
    correct_sum = 0
    # Use GPU to evaluate if possible
    device = torch.device(
        "cuda:0" if model.num_gpus > 0 and torch.cuda.is_available() else "cpu"
    )
    with torch.no_grad():
        for i, (data, target) in enumerate(test_loader):
            out = model(data)
            pred = out.argmax(dim=1, keepdim=True)
            pred, target = pred.to(device), target.to(device)
            correct = pred.eq(target.view_as(pred)).sum().item()
            correct_sum += correct

    logger.info(f"Accuracy {correct_sum / len(test_loader.dataset)}")



# Main loop for trainers.
def run_worker(rank, world_size, num_gpus, train_loader, test_loader):
    logger.info(f"Worker rank {rank} initializing RPC")
    rpc.init_rpc(name=f"trainer_{rank}", rank=rank, world_size=world_size)

    logger.info(f"Worker {rank} done initializing RPC")

    run_training_loop(rank, world_size, num_gpus, train_loader, test_loader)
    rpc.shutdown()



# --------- Launcher --------------------



def runn(rank, world_size):
    if rank == 0:
        run_parameter_server(0, world_size)
    else:
        # Get data to train on
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "../data",
                train=True,
                download=True,
                transform=transforms.Compose(
                    [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
                ),
            ),
            batch_size=32,
            shuffle=True,
        )
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "../data",
                train=False,
                transform=transforms.Compose(
                    [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
                ),
            ),
            batch_size=32,
            shuffle=True,
        )
        # start training worker on this node
        run_worker(rank, world_size, 0, train_loader, test_loader)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Parameter-Server RPC based training")
    parser.add_argument(
        "--world_size",
        type=int,
        default=16,
        help="""Total number of participating processes. Should be the sum of
        master node and all training nodes.""",
    )
    parser.add_argument(
        "--rank",
        type=int,
        default=None,
        help="Global rank of this process. Pass in 0 for master.",
    )
    parser.add_argument(
        "--num_gpus",
        type=int,
        default=0,
        help="""Number of GPUs to use for training, currently supports between 0
         and 2 GPUs. Note that this argument will be passed to the parameter servers.""",
    )
    parser.add_argument(
        "--master_addr",
        type=str,
        default="localhost",
        help="""Address of master, will default to localhost if not provided.
        Master must be able to accept network traffic on the address + port.""",
    )
    parser.add_argument(
        "--master_port",
        type=str,
        default="29500",
        help="""Port that master is listening on, will default to 29500 if not
        provided. Master must be able to accept network traffic on the host and port.""",
    )

    args = parser.parse_args()
    # assert args.rank is not None, "must provide rank argument."
    assert (
        args.num_gpus <= 3
    ), f"Only 0-2 GPUs currently supported (got {args.num_gpus})."
    os.environ["MASTER_ADDR"] = args.master_addr
    os.environ["MASTER_PORT"] = args.master_port
    processes = []
    world_size = args.world_size

    mp.spawn(runn, args=(world_size,), nprocs=world_size, join=True)

==== run_logicshredder.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: run_logicshredder.py
Unified launcher that respects neuro.lock and intelligently spawns agent swarm
"""

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

AGENTS = {
    "token_agent": "agents/token_agent.py",
    "validator": "agents/validator.py",
    "guffifier": "agents/guffifier_v2.py",
    "mutation_engine": "agents/mutation_engine.py",
    "dreamwalker": "agents/dreamwalker.py",
    "meta_agent": "agents/meta_agent.py",
    "cold_logic_mover": "agents/cold_logic_mover.py",
    "cortex_logger": "agents/cortex_logger.py",
}

LOCK_FILE = Path("core/neuro.lock")

def is_locked():
    return LOCK_FILE.exists()

def should_skip(agent_name):
    # Lock disables mutation, walk, dream agents
    locked_agents = ["token_agent", "dreamwalker", "mutation_engine"]
    return agent_name in locked_agents and is_locked()

def launch_agent(agent_name, path):
    if should_skip(agent_name):
        print(f"[run_logicshredder] ? Skipped {agent_name} (brain is locked)")
        return

    interpreter = "python" if platform.system() == "Windows" else "python3"
    try:
        subprocess.Popen([interpreter, path])
        print(f"[run_logicshredder] [OK] Launched: {agent_name}")
    except Exception as e:
        print(f"[run_logicshredder] ERROR Failed to launch {agent_name}: {e}")

def main():
    print("INFO LOGICSHREDDER is activating...")
    if is_locked():
        print("? Brain is LOCKED. Only non-destructive modules will run.")
    else:
        print("? Brain is ACTIVE. Full cognition mode engaged.")

    for agent_name, script_path in AGENTS.items():
        launch_agent(agent_name, script_path)
        time.sleep(0.5)  # Prevent CPU spike at boot

    print("[run_logicshredder] Launch complete. Mind is operational.")

if __name__ == "__main__":
    main()
# [CONFIG_PATCHED]

==== server_embd.py ====
import asyncio
import asyncio.threads
import requests
import numpy as np



n = 8

result = []

async def requests_post_async(*args, **kwargs):
    return await asyncio.threads.to_thread(requests.post, *args, **kwargs)

async def main():
    model_url = "http://127.0.0.1:6900"
    responses: list[requests.Response] = await asyncio.gather(*[requests_post_async(
        url= f"{model_url}/embedding",
        json= {"content": "a "*1022}
    ) for i in range(n)])

    for response in responses:
        embedding = response.json()["embedding"]
        print(embedding[-8:])
        result.append(embedding)

asyncio.run(main())

# compute cosine similarity

for i in range(n-1):
    for j in range(i+1, n):
        embedding1 = np.array(result[i])
        embedding2 = np.array(result[j])
        similarity = np.dot(embedding1, embedding2) / (np.linalg.norm(embedding1) * np.linalg.norm(embedding2))
        print(f"Similarity between {i} and {j}: {similarity:.2f}")

==== simple_regression.py ====
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as F

from crm.core import Network
from crm.utils import seed_all

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

if __name__ == "__main__":
    seed_all(24)
    n = Network(
        2,
        [[1], []],
        custom_activations=((lambda x: x, lambda x: 1), (lambda x: x, lambda x: 1)),
    )
    n.to(device)
    optimizer = torch.optim.Adam(n.parameters(), lr=0.001)
    inputs = torch.linspace(-1, 1, 1000).to(device)
    labels = inputs / 2
    losses = []
    for i in range(1000):
        out = n.forward(torch.tensor([inputs[i], 1]))
        loss = F.mse_loss(out[0].reshape(1), labels[i].reshape(1))
        losses.append(loss.item())
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        n.reset()
    print(n.weights)
    plt.plot(losses)
    plt.show()

==== subcon_layer_mapper.py ====
import os
import yaml
from pathlib import Path
```

```
LAYER_MAP_PATH = Path("subcon_map.yaml")
FRAGMENTS_DIR = Path("fragments/core")
OUTPUT_PATH = Path("meta/subcon_layer_cache.yaml")
OUTPUT_PATH.parent.mkdir(parents=True, exist_ok=True)

class SubconLayerMapper:
    def __init__(self):
        self.layer_map = self.load_map()

    def load_map(self):
        if not LAYER_MAP_PATH.exists():
            print("[Mapper] No layer map found. Returning empty.")
            return {}
        with open(LAYER_MAP_PATH, 'r') as f:
            return yaml.safe_load(f)

    def extract_links(self):
        results = {}
        for file in FRAGMENTS_DIR.glob("*.yaml"):
            try:
                with open(file, 'r') as f:
                    frag = yaml.safe_load(f)
                tags = frag.get("tags", [])
                for tag in tags:
                    if tag in self.layer_map:
                        results.setdefault(tag, []).append(frag['id'])
            except Exception as e:
                print(f"[Mapper] Failed to read {file.name}: {e}")
        return results

    def save_cache(self, data):
        with open(OUTPUT_PATH, 'w') as out:
            yaml.dump(data, out)
        print(f"[Mapper] Saved subcon layer associations -> {OUTPUT_PATH}")

    def run(self):
        links = self.extract_links()
        self.save_cache(links)

if __name__ == "__main__":
    mapper = SubconLayerMapper()
    mapper.run()

==== subgraph.py ====
import argparse
import sys

import torch
import torch.nn.functional as F

from crm.core import Network
from crm.utils import get_explanations, get_metrics, make_dataset_cli, seed_all, train
```

```
def cmd_line_args():
    parser = argparse.ArgumentParser(
        description="CRM; Example: python3 main.py -f inp.file -o out.file -n 20"
    )
    parser.add_argument("-f", "--file", help="input file", required=True)
    parser.add_argument("-o", "--output", help="output file", required=True)
    parser.add_argument(
        "-n", "--num-epochs", type=int, help="number of epochs", required=True
    )
    parser.add_argument(
        "-e", "--explain", help="get explanations for predictions", action="store_true"
    )
    parser.add_argument(
        "-v", "--verbose", help="get verbose outputs", action="store_true"
    )
    parser.add_argument("-g", "--gpu", help="run model on gpu", action="store_true")
    args = parser.parse_args()
    return args


class Logger(object):
    def __init__(self, filename):
        self.terminal = sys.stdout
        self.log = open(filename, "a")

    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)

    def flush(self):
        pass


def main():
    seed_all(24)
    args = cmd_line_args()
    device = torch.device("cuda" if torch.cuda.is_available() and args.gpu else "cpu")
    sys.stdout = Logger(args.output)
    print(args)
    file_name = args.file
    with open(file_name, "r") as f:
        graph_file = f.readline()[:-1]
        train_file = f.readline()[:-1]
        test_files = f.readline()[:-1].split()
        true_explanations = list(map(int, f.readline()[:-1].split()))
    X_train, y_train, test_dataset, adj_list, edges = make_dataset_cli(
        graph_file, train_file, test_files, device=device
    )
    n = Network(len(adj_list), adj_list)
    n.to(device)
    criterion = F.cross_entropy
    optimizer = torch.optim.Adam(n.parameters(), lr=0.001)
    train_losses, train_accs = train(
        n, X_train, y_train, args.num_epochs, optimizer, criterion, verbose=args.verbose
    )

    # Train metrics
    print("Train Metrics")
    print(get_metrics(n, X_train, y_train))
    print("##############################")

    # Test metrics
    print("Test Metrics")
    for X_test, y_test in test_dataset:
        print(get_metrics(n, X_test, y_test))
        print("-------------------------------------")
    print("##############################")

    if args.explain:
        print("Explanations")
        for X_test, y_test in test_dataset:
            get_explanations(
                n,
                X_test,
                y_test,
                true_explanations=true_explanations,
                verbose=args.verbose,
            )
            print("-------------------------------------")
        print("##############################")


if __name__ == "__main__":
    main()

==== symbol_seed_generator.py ====
import os
import yaml
import hashlib
from datetime import datetime

USE_LLM = False
MODEL_PATH = "models/mistral-7b-q4.gguf"
SEED_OUTPUT_DIR = "fragments/core/"
SEED_COUNT = 100

BASE_SEEDS = [
    "truth is important",
    "conflict creates learning",
    "change is constant",
    "observation precedes action",
    "emotion influences memory",
    "self seeks meaning",
    "logic guides belief",
    "doubt triggers inquiry",
    "energy becomes form",
    "ideas replicate",
    "something must stay_still so everything else can move"
```

```
]

def generate_id(content):
    return hashlib.sha256(content.encode()).hexdigest()[:12]

def to_fragment(statement):
    parts = statement.split()
    if len(parts) < 3:
        return None
    subj = parts[0]
    pred = parts[1]
    obj = "_".join(parts[2:])
    return {
        "id": generate_id(statement),
        "predicate": pred,
        "arguments": [subj, obj],
        "confidence": 1.0,
        "emotion": {
            "curiosity": 0.8,
            "certainty": 1.0
        },
        "tags": ["seed", "immutable", "core"],
        "immutable": True,
        "claim": statement,
        "timestamp": datetime.utcnow().isoformat()
    }

def save_fragment(fragment, output_dir):
    fname = f"frag_{fragment['id']}.yaml"
    path = os.path.join(output_dir, fname)
    with open(path, 'w') as f:
        yaml.dump(fragment, f)

def generate_symbolic_seeds():
    if not os.path.exists(SEED_OUTPUT_DIR):
        os.makedirs(SEED_OUTPUT_DIR)
    seed_statements = BASE_SEEDS[:SEED_COUNT]
    count = 0
    for stmt in seed_statements:
        frag = to_fragment(stmt)
        if frag:
            save_fragment(frag, SEED_OUTPUT_DIR)
            count += 1
    print(f"Generated {count} symbolic seed fragments in {SEED_OUTPUT_DIR}")

if __name__ == "__main__":
    generate_symbolic_seeds()

==== tensor_mapping.py ====
from __future__ import annotations

from typing import Sequence

from .constants import MODEL_ARCH, MODEL_TENSOR, MODEL_TENSORS, TENSOR_NAMES
```

```
class TensorNameMap:
    mappings_cfg: dict[MODEL_TENSOR, tuple[str, ...]] = {
        # Token embeddings
        MODEL_TENSOR.TOKEN_EMBD: (
            "gpt_neox.embed_in",                        # gptneox
            "transformer.wte",                          # gpt2 gpt-j mpt refact qwen dbrx jais exaone
            "transformer.word_embeddings",              # falcon
            "word_embeddings",                          # bloom
            "model.embed_tokens",                       # llama-hf nemotron olmoe olmo2 rwkv6qwen2 glm4-0414
            "tok_embeddings",                           # llama-pth
            "embeddings.word_embeddings",               # bert nomic-bert
            "language_model.embedding.word_embeddings",  # persimmon
            "wte",                                      # gpt2
            "transformer.embd.wte",                     # phi2
            "model.tok_embeddings",                     # internlm2
            "model.embedding",                          # mamba-qbert
            "backbone.embedding",                       # mamba
            "backbone.embeddings",                      # mamba-hf
            "transformer.in_out_embed",                 # Grok
            "embedding.word_embeddings",                # chatglm
            "transformer.token_embeddings",             # openelm
            "shared",                                   # t5
            "rwkv.embeddings",                          # rwkv6
            "model.embeddings",                         # rwkv7
            "model.word_embeddings",                    # bailingmoe
            "language_model.model.embed_tokens",        # llama4
        ),

        # Token type embeddings
        MODEL_TENSOR.TOKEN_TYPES: (
            "embeddings.token_type_embeddings",  # bert nomic-bert
        ),

        # Normalization of token embeddings
        MODEL_TENSOR.TOKEN_EMBD_NORM: (
            "word_embeddings_layernorm",  # bloom
            "embeddings.LayerNorm",       # bert
            "emb_ln",                     # nomic-bert
            "transformer.norm",           # openelm
            "rwkv.blocks.0.pre_ln",       # rwkv
            "rwkv.blocks.0.pre_ln",       # rwkv6
            "model.pre_ln",               # rwkv7
            "model.layers.0.pre_norm",    # rwkv7
            "backbone.norm",              # wavtokenizer
        ),

        # Position embeddings
        MODEL_TENSOR.POS_EMBD: (
            "transformer.wpe",                 # gpt2
            "embeddings.position_embeddings",  # bert
            "wpe",                             # gpt2
        ),
        # Output
        MODEL_TENSOR.OUTPUT: (
            "embed_out",                 # gptneox
             "lm_head",                      # gpt2 mpt falcon llama-hf baichuan qwen mamba dbrx jais nemotron
exaone olmoe olmo2 phimoe
            "output",                    # llama-pth bloom internlm2
            "word_embeddings_for_head",  # persimmon
            "lm_head.linear",            # phi2
            "output_layer",              # chatglm
            "head",                      # rwkv
            "head.out",                  # wavtokenizer
            "language_model.lm_head",    # llama4
        ),

        # Output norm
        MODEL_TENSOR.OUTPUT_NORM: (
            "gpt_neox.final_layer_norm",               # gptneox
            "transformer.ln_f",                        # gpt2 gpt-j falcon jais exaone
            "model.norm",                              # llama-hf baichuan internlm2 olmoe olmo2 phimoe
            "norm",                                    # llama-pth
            "transformer.norm_f",                      # mpt dbrx
            "ln_f",                                    # refact bloom qwen gpt2
            "language_model.encoder.final_layernorm",  # persimmon
            "model.final_layernorm",                   # persimmon
            "lm_head.ln",                              # phi2
            "model.norm_f",                            # mamba-qbert
            "backbone.norm_f",                         # mamba
            "transformer.rms_norm",                    # Grok
            "encoder.final_layernorm",                 # chatglm
            "transformer.norm",                        # openelm
            "model.norm",                              # nemotron
            "rwkv.ln_out",                             # rwkv6
            "model.ln_out",                            # rwkv7
            "backbone.final_layer_norm",               # wavtokenizer
            "language_model.model.norm",               # llama4
        ),

        # Rope frequencies
        MODEL_TENSOR.ROPE_FREQS: (
            "rope.freqs",  # llama-pth
            "rotary_pos_emb.inv_freq",  # chatglm
        ),

        MODEL_TENSOR.ROPE_FACTORS_LONG: (),
        MODEL_TENSOR.ROPE_FACTORS_SHORT: (),

        MODEL_TENSOR.CONV1D: (
            "backbone.embed", # roberta
        ),
    }

    block_mappings_cfg: dict[MODEL_TENSOR, tuple[str, ...]] = {
        # Attention norm
        MODEL_TENSOR.ATTN_NORM: (
            "gpt_neox.layers.{bid}.input_layernorm",                # gptneox
            "transformer.h.{bid}.ln_1",                             # gpt2 gpt-j refact qwen jais exaone
            "transformer.blocks.{bid}.norm_1",                      # mpt
            "transformer.h.{bid}.input_layernorm",                  # falcon7b
            "h.{bid}.input_layernorm",                              # bloom
            "transformer.h.{bid}.ln_mlp",                           # falcon40b
            "model.layers.{bid}.input_layernorm",                   # llama-hf nemotron olmoe phimoe
            "layers.{bid}.attention_norm",                          # llama-pth
            "language_model.encoder.layers.{bid}.input_layernorm",  # persimmon
            "model.layers.{bid}.ln1",                               # yi
            "h.{bid}.ln_1",                                         # gpt2
            "transformer.h.{bid}.ln",                               # phi2
            "model.layers.layers.{bid}.norm",                       # plamo
            "model.layers.{bid}.attention_norm",                    # internlm2
            "model.layers.{bid}.norm",                              # mamba-qbert
            "backbone.layers.{bid}.norm",                           # mamba
            "transformer.decoder_layer.{bid}.rms_norm",             # Grok
            "transformer.blocks.{bid}.norm_attn_norm.norm_1",       # dbrx
            "encoder.layers.{bid}.input_layernorm",                 # chatglm
            "transformer.layers.{bid}.attn_norm",                   # openelm
            "rwkv.blocks.{bid}.ln1",                                # rwkv6
            "model.layers.{bid}.ln1",                               # rwkv7
            "language_model.model.layers.{bid}.input_layernorm",    # llama4
        ),

        # Attention norm 2
        MODEL_TENSOR.ATTN_NORM_2: (
            "transformer.h.{bid}.ln_attn",                  # falcon40b
            "encoder.layer.{bid}.layer_norm_1",             # jina-v2-code
            "rwkv.blocks.{bid}.ln2",                        # rwkv6
            "model.layers.{bid}.ln2",                       # rwkv7
        ),

        # Attention query-key-value
        MODEL_TENSOR.ATTN_QKV: (
            "gpt_neox.layers.{bid}.attention.query_key_value",                     # gptneox
            "transformer.h.{bid}.attn.c_attn",                                     # gpt2 qwen jais
            "transformer.blocks.{bid}.attn.Wqkv",                                  # mpt
            "transformer.blocks.{bid}.norm_attn_norm.attn.Wqkv",                   # dbrx
            "transformer.h.{bid}.self_attention.query_key_value",                  # falcon
            "h.{bid}.self_attention.query_key_value",                              # bloom
            "language_model.encoder.layers.{bid}.self_attention.query_key_value",  # persimmon
            "model.layers.{bid}.self_attn.query_key_value",                        # persimmon
            "h.{bid}.attn.c_attn",                                                 # gpt2
            "transformer.h.{bid}.mixer.Wqkv",                                      # phi2
            "encoder.layers.{bid}.attn.Wqkv",                                      # nomic-bert
            "model.layers.{bid}.self_attn.qkv_proj",                               # phi3
            "encoder.layers.{bid}.self_attention.query_key_value",                 # chatglm
            "transformer.layers.{bid}.attn.qkv_proj",                              # openelm
        ),

        # Attention query
        MODEL_TENSOR.ATTN_Q: (
            "model.layers.{bid}.self_attn.q_proj",                       # llama-hf nemotron olmoe olmo2 phimoe
            "model.layers.{bid}.self_attn.q_proj_no_perm",               # llama-custom
            "layers.{bid}.attention.wq",                                 # llama-pth
            "encoder.layer.{bid}.attention.self.query",                  # bert
            "transformer.h.{bid}.attn.q_proj",                           # gpt-j
            "model.layers.layers.{bid}.self_attn.q_proj",                # plamo
            "model.layers.{bid}.attention.wq",                           # internlm2
            "transformer.decoder_layer.{bid}.multi_head_attention.query",# Grok
            "transformer.h.{bid}.attn.attention.q_proj",                 # exaone
            "language_model.model.layers.{bid}.self_attn.q_proj",        # llama4
        ),

        # Attention key
        MODEL_TENSOR.ATTN_K: (
            "model.layers.{bid}.self_attn.k_proj",                     # llama-hf nemotron olmoe olmo2 phimoe
            "model.layers.{bid}.self_attn.k_proj_no_perm",             # llama-custom
            "layers.{bid}.attention.wk",                               # llama-pth
            "encoder.layer.{bid}.attention.self.key",                  # bert
            "transformer.h.{bid}.attn.k_proj",                         # gpt-j
            "transformer.h.{bid}.attn.k",                              # refact
            "model.layers.layers.{bid}.self_attn.k_proj",              # plamo
            "model.layers.{bid}.attention.wk",                         # internlm2
            "transformer.decoder_layer.{bid}.multi_head_attention.key",# Grok
            "transformer.h.{bid}.attn.attention.k_proj",               # exaone
            "language_model.model.layers.{bid}.self_attn.k_proj",      # llama4
        ),

        # Attention value
        MODEL_TENSOR.ATTN_V: (
            "model.layers.{bid}.self_attn.v_proj",                       # llama-hf nemotron olmoe olmo2 phimoe
            "layers.{bid}.attention.wv",                                 # llama-pth
            "encoder.layer.{bid}.attention.self.value",                  # bert
            "transformer.h.{bid}.attn.v_proj",                           # gpt-j
            "transformer.h.{bid}.attn.v",                                # refact
            "model.layers.layers.{bid}.self_attn.v_proj",                # plamo
            "model.layers.{bid}.attention.wv",                           # internlm2
            "transformer.decoder_layer.{bid}.multi_head_attention.value",# Grok
            "transformer.h.{bid}.attn.attention.v_proj",                 # exaone
            "language_model.model.layers.{bid}.self_attn.v_proj",        # llama4
        ),

        # Attention output
        MODEL_TENSOR.ATTN_OUT: (
            "gpt_neox.layers.{bid}.attention.dense",                        # gptneox
            "transformer.h.{bid}.attn.c_proj",                              # gpt2 refact qwen jais
            "transformer.blocks.{bid}.attn.out_proj",                       # mpt
            "transformer.h.{bid}.self_attention.dense",                     # falcon
            "h.{bid}.self_attention.dense",                                 # bloom
             "model.layers.{bid}.self_attn.o_proj",                            # llama-hf nemotron olmoe olmo2
phimoe
            "model.layers.{bid}.self_attn.linear_attn",                     # deci
            "layers.{bid}.attention.wo",                                    # llama-pth
            "encoder.layer.{bid}.attention.output.dense",                   # bert
            "transformer.h.{bid}.attn.out_proj",                            # gpt-j
            "language_model.encoder.layers.{bid}.self_attention.dense",     # persimmon
            "model.layers.{bid}.self_attn.dense",                           # persimmon
            "h.{bid}.attn.c_proj",                                          # gpt2
            "transformer.h.{bid}.mixer.out_proj",                           # phi2
            "model.layers.layers.{bid}.self_attn.o_proj",                   # plamo
            "model.layers.{bid}.attention.wo",                              # internlm2
            "encoder.layers.{bid}.attn.out_proj",                           # nomic-bert
            "transformer.decoder_layer.{bid}.multi_head_attention.linear",  # Grok
            "transformer.blocks.{bid}.norm_attn_norm.attn.out_proj",        # dbrx
            "encoder.layers.{bid}.self_attention.dense",                    # chatglm
            "transformer.layers.{bid}.attn.out_proj",                       # openelm
            "transformer.h.{bid}.attn.attention.out_proj",                  # exaone
            "language_model.model.layers.{bid}.self_attn.o_proj",           # llama4
        ),

        # Attention output norm
        MODEL_TENSOR.ATTN_OUT_NORM: (
            "encoder.layer.{bid}.attention.output.LayerNorm",  # bert
            "encoder.layers.{bid}.norm1",                      # nomic-bert
            "transformer.decoder_layer.{bid}.rms_norm_1",      # Grok
            "transformer.blocks.{bid}.norm_attn_norm.norm_2",  # dbrx
        ),

        MODEL_TENSOR.ATTN_POST_NORM: (
            "model.layers.{bid}.post_attention_layernorm",     # gemma2 olmo2    # ge
            "model.layers.{bid}.post_self_attn_layernorm",     # glm-4-0414
        ),

        # Rotary embeddings
        MODEL_TENSOR.ATTN_ROT_EMBD: (
            "model.layers.{bid}.self_attn.rotary_emb.inv_freq",        # llama-hf
            "layers.{bid}.attention.inner_attention.rope.freqs",       # llama-pth
            "model.layers.layers.{bid}.self_attn.rotary_emb.inv_freq", # plamo
            "transformer.h.{bid}.attn.rotary_emb.inv_freq",            # codeshell
        ),

        # Feed-forward norm
        MODEL_TENSOR.FFN_NORM: (
            "gpt_neox.layers.{bid}.post_attention_layernorm",                # gptneox
            "transformer.h.{bid}.ln_2",                                      # gpt2 refact qwen jais exaone
            "h.{bid}.post_attention_layernorm",                              # bloom
            "transformer.blocks.{bid}.norm_2",                               # mpt
            "model.layers.{bid}.post_attention_layernorm",                   # llama-hf nemotron olmoe phimoe
            "layers.{bid}.ffn_norm",                                         # llama-pth
            "language_model.encoder.layers.{bid}.post_attention_layernorm",  # persimmon
            "model.layers.{bid}.ln2",                                        # yi
            "h.{bid}.ln_2",                                                  # gpt2
            "model.layers.{bid}.ffn_norm",                                   # internlm2
            "transformer.decoder_layer.{bid}.rms_norm_2",                    # Grok
            "encoder.layers.{bid}.post_attention_layernorm",                 # chatglm
            "transformer.layers.{bid}.ffn_norm",                             # openelm
            "language_model.model.layers.{bid}.post_attention_layernorm",    # llama4
        ),
        # Post feed-forward norm
        MODEL_TENSOR.FFN_PRE_NORM: (
            "model.layers.{bid}.pre_feedforward_layernorm", # gemma2
        ),

        # Post feed-forward norm
        MODEL_TENSOR.FFN_POST_NORM: (
            "model.layers.{bid}.post_feedforward_layernorm", # gemma2 olmo2
            "model.layers.{bid}.post_mlp_layernorm", # glm-4-0414
        ),

        MODEL_TENSOR.FFN_GATE_INP: (
            "layers.{bid}.feed_forward.gate",                    # mixtral
            "model.layers.{bid}.block_sparse_moe.gate",          # mixtral phimoe
            "model.layers.{bid}.mlp.gate",                       # qwen2moe olmoe
            "transformer.decoder_layer.{bid}.router",            # Grok
            "transformer.blocks.{bid}.ffn.router.layer",         # dbrx
            "model.layers.{bid}.block_sparse_moe.router.layer", # granitemoe
            "language_model.model.layers.{bid}.feed_forward.router", # llama4
        ),

        MODEL_TENSOR.FFN_GATE_INP_SHEXP: (
            "model.layers.{bid}.mlp.shared_expert_gate", # qwen2moe
        ),

        MODEL_TENSOR.FFN_EXP_PROBS_B: (
            "model.layers.{bid}.mlp.gate.e_score_correction", # deepseek-v3
        ),

        # Feed-forward up
        MODEL_TENSOR.FFN_UP: (
            "gpt_neox.layers.{bid}.mlp.dense_h_to_4h",                # gptneox
            "transformer.h.{bid}.mlp.c_fc",                           # gpt2 jais
            "transformer.blocks.{bid}.ffn.up_proj",                   # mpt
            "transformer.h.{bid}.mlp.dense_h_to_4h",                  # falcon
            "h.{bid}.mlp.dense_h_to_4h",                              # bloom
            "model.layers.{bid}.mlp.up_proj",                         # llama-hf refact nemotron olmo2
            "layers.{bid}.feed_forward.w3",                           # llama-pth
            "encoder.layer.{bid}.intermediate.dense",                 # bert
            "transformer.h.{bid}.mlp.fc_in",                          # gpt-j
            "transformer.h.{bid}.mlp.linear_3",                       # refact
            "language_model.encoder.layers.{bid}.mlp.dense_h_to_4h",  # persimmon
            "model.layers.{bid}.mlp.dense_h_to_4h",                   # persimmon
            "transformer.h.{bid}.mlp.w1",                             # qwen
            "h.{bid}.mlp.c_fc",                                       # gpt2
            "transformer.h.{bid}.mlp.fc1",                            # phi2
            "model.layers.{bid}.mlp.fc1",                             # phi2
            "model.layers.{bid}.mlp.gate_up_proj",                    # phi3 glm-4-0414
            "model.layers.layers.{bid}.mlp.up_proj",                  # plamo
            "model.layers.{bid}.feed_forward.w3",                     # internlm2
            "encoder.layers.{bid}.mlp.fc11",                          # nomic-bert
            "model.layers.{bid}.mlp.c_fc",                            # starcoder2
            "encoder.layer.{bid}.mlp.gated_layers_v",                 # jina-bert-v2
            "model.layers.{bid}.residual_mlp.w3",                     # arctic
```

```
    "encoder.layers.{bid}.mlp.dense_h_to_4h",                 # chatglm
    "transformer.h.{bid}.mlp.c_fc_1",                         # exaone
    "language_model.model.layers.{bid}.feed_forward.up_proj", # llama4
),

MODEL_TENSOR.FFN_UP_EXP: (
    "layers.{bid}.feed_forward.experts.w3",           # mixtral (merged)
    "transformer.decoder_layer.{bid}.moe.linear_v",   # Grok (merged)
    "transformer.blocks.{bid}.ffn.experts.mlp.v1",    # dbrx
    "model.layers.{bid}.mlp.experts.up_proj",         # qwen2moe olmoe (merged)
    "model.layers.{bid}.block_sparse_moe.experts.w3", # phimoe (merged)
    "language_model.model.layers.{bid}.feed_forward.experts.up_proj", # llama4
),

MODEL_TENSOR.FFN_UP_SHEXP: (
    "model.layers.{bid}.mlp.shared_expert.up_proj",  # qwen2moe
    "model.layers.{bid}.mlp.shared_experts.up_proj", # deepseek deepseek2
    "language_model.model.layers.{bid}.feed_forward.shared_expert.up_proj", # llama4
),

# AWQ-activation gate
MODEL_TENSOR.FFN_ACT: (
    "transformer.blocks.{bid}.ffn.act",  # mpt
),

# Feed-forward gate
MODEL_TENSOR.FFN_GATE: (
    "model.layers.{bid}.mlp.gate_proj",           # llama-hf refact olmo2
    "layers.{bid}.feed_forward.w1",               # llama-pth
    "transformer.h.{bid}.mlp.w2",                 # qwen
    "transformer.h.{bid}.mlp.c_fc2",              # jais
    "model.layers.layers.{bid}.mlp.gate_proj",    # plamo
    "model.layers.{bid}.feed_forward.w1",         # internlm2
    "encoder.layers.{bid}.mlp.fc12",              # nomic-bert
    "encoder.layer.{bid}.mlp.gated_layers_w",     # jina-bert-v2
    "transformer.h.{bid}.mlp.linear_1",           # refact
    "model.layers.{bid}.residual_mlp.w1",         # arctic
    "transformer.h.{bid}.mlp.c_fc_0",             # exaone
    "language_model.model.layers.{bid}.feed_forward.gate_proj", # llama4
),

MODEL_TENSOR.FFN_GATE_EXP: (
    "layers.{bid}.feed_forward.experts.w1",           # mixtral (merged)
    "transformer.decoder_layer.{bid}.moe.linear",     # Grok (merged)
    "transformer.blocks.{bid}.ffn.experts.mlp.w1",    # dbrx
    "model.layers.{bid}.mlp.experts.gate_proj",       # qwen2moe olmoe (merged)
    "model.layers.{bid}.block_sparse_moe.experts.w1", # phimoe (merged)
    "language_model.model.layers.{bid}.feed_forward.experts.gate_proj", # llama4
),

MODEL_TENSOR.FFN_GATE_SHEXP: (
    "model.layers.{bid}.mlp.shared_expert.gate_proj",  # qwen2moe
    "model.layers.{bid}.mlp.shared_experts.gate_proj", # deepseek deepseek2
    "language_model.model.layers.{bid}.feed_forward.shared_expert.gate_proj", # llama4
),

# Feed-forward down
MODEL_TENSOR.FFN_DOWN: (
    "gpt_neox.layers.{bid}.mlp.dense_4h_to_h",               # gptneox
    "transformer.h.{bid}.mlp.c_proj",                         # gpt2 refact qwen jais
    "transformer.blocks.{bid}.ffn.down_proj",                 # mpt
    "transformer.h.{bid}.mlp.dense_4h_to_h",                  # falcon
    "h.{bid}.mlp.dense_4h_to_h",                              # bloom
    "model.layers.{bid}.mlp.down_proj",                       # llama-hf nemotron olmo2
    "layers.{bid}.feed_forward.w2",                           # llama-pth
    "encoder.layer.{bid}.output.dense",                       # bert
    "transformer.h.{bid}.mlp.fc_out",                         # gpt-j
    "language_model.encoder.layers.{bid}.mlp.dense_4h_to_h",  # persimmon
    "model.layers.{bid}.mlp.dense_4h_to_h",                   # persimmon
    "h.{bid}.mlp.c_proj",                                     # gpt2
    "transformer.h.{bid}.mlp.fc2",                            # phi2
    "model.layers.{bid}.mlp.fc2",                             # phi2
    "model.layers.layers.{bid}.mlp.down_proj",                # plamo
    "model.layers.{bid}.feed_forward.w2",                     # internlm2
    "encoder.layers.{bid}.mlp.fc2",                           # nomic-bert
    "model.layers.{bid}.mlp.c_proj",                          # starcoder2
    "encoder.layer.{bid}.mlp.wo",                             # jina-bert-v2
    "transformer.layers.{bid}.ffn.proj_2",                    # openelm
    "model.layers.{bid}.residual_mlp.w2",                     # arctic
    "encoder.layer.{bid}.mlp.down_layer",                     # jina-bert-v2
    "encoder.layers.{bid}.mlp.dense_4h_to_h",                 # chatglm
    "model.layers.h.{bid}.mlp.c_proj",                        # exaone
    "language_model.model.layers.{bid}.feed_forward.down_proj", # llama4
),

MODEL_TENSOR.FFN_DOWN_EXP: (
    "layers.{bid}.feed_forward.experts.w2",              # mixtral (merged)
    "transformer.decoder_layer.{bid}.moe.linear_1",      # Grok (merged)
    "transformer.blocks.{bid}.ffn.experts.mlp.w2",       # dbrx
    "model.layers.{bid}.mlp.experts.down_proj",          # qwen2moe olmoe (merged)
    "model.layers.{bid}.block_sparse_moe.output_linear", # granitemoe
    "model.layers.{bid}.block_sparse_moe.experts.w2",    # phimoe (merged)
    "language_model.model.layers.{bid}.feed_forward.experts.down_proj", # llama4
),

MODEL_TENSOR.FFN_DOWN_SHEXP: (
    "model.layers.{bid}.mlp.shared_expert.down_proj",  # qwen2moe
    "model.layers.{bid}.mlp.shared_experts.down_proj", # deepseek deepseek2
    "language_model.model.layers.{bid}.feed_forward.shared_expert.down_proj", # llama4
),

MODEL_TENSOR.ATTN_Q_NORM: (
    "language_model.encoder.layers.{bid}.self_attention.q_layernorm",
    "model.layers.{bid}.self_attn.q_layernorm",                       # persimmon
    "model.layers.{bid}.self_attn.q_norm",                            # cohere olmoe chameleon olmo2
    "transformer.blocks.{bid}.attn.q_ln",                             # sea-lion
    "encoder.layer.{bid}.attention.self.layer_norm_q",                # jina-bert-v2
    "transformer.layers.{bid}.attn.q_norm",                           # openelm
),

MODEL_TENSOR.ATTN_K_NORM: (
    "language_model.encoder.layers.{bid}.self_attention.k_layernorm",
    "model.layers.{bid}.self_attn.k_layernorm",                       # persimmon
    "model.layers.{bid}.self_attn.k_norm",                            # cohere olmoe chameleon olmo2
    "transformer.blocks.{bid}.attn.k_ln",                             # sea-lion
    "encoder.layer.{bid}.attention.self.layer_norm_k",                # jina-bert-v2
    "transformer.layers.{bid}.attn.k_norm",                           # openelm
),

MODEL_TENSOR.ROPE_FREQS: (
    "language_model.encoder.layers.{bid}.self_attention.rotary_emb.inv_freq",  # persimmon
),

MODEL_TENSOR.LAYER_OUT_NORM: (
    "encoder.layer.{bid}.output.LayerNorm",         # bert
    "encoder.layers.{bid}.norm2",                   # nomic-bert
    "transformer.decoder_layer.{bid}.rms_norm_3",   # Grok
    "encoder.layer.{bid}.mlp.layernorm",            # jina-bert-v2
    "encoder.layer.{bid}.layer_norm_2"              # jina-v2-code
),

MODEL_TENSOR.SSM_IN: (
    "model.layers.{bid}.in_proj",
    "backbone.layers.{bid}.mixer.in_proj",
),

MODEL_TENSOR.SSM_CONV1D: (
    "model.layers.{bid}.conv1d",
    "backbone.layers.{bid}.mixer.conv1d",
),

MODEL_TENSOR.SSM_X: (
    "model.layers.{bid}.x_proj",
    "backbone.layers.{bid}.mixer.x_proj",
),

MODEL_TENSOR.SSM_DT: (
    "model.layers.{bid}.dt_proj",
    "backbone.layers.{bid}.mixer.dt_proj",
),

MODEL_TENSOR.SSM_A: (
    "model.layers.{bid}.A_log",
    "backbone.layers.{bid}.mixer.A_log",
),

MODEL_TENSOR.SSM_D: (
    "model.layers.{bid}.D",
    "backbone.layers.{bid}.mixer.D",
),

MODEL_TENSOR.SSM_OUT: (
    "model.layers.{bid}.out_proj",
    "backbone.layers.{bid}.mixer.out_proj",
),

MODEL_TENSOR.TIME_MIX_W0: (
    "model.layers.{bid}.attention.w0",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_W1: (
    "rwkv.blocks.{bid}.attention.time_maa_w1",    # rwkv6
    "model.layers.{bid}.self_attn.time_maa_w1",   # rwkv6qwen2
    "model.layers.{bid}.attention.w1",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_W2: (
    "rwkv.blocks.{bid}.attention.time_maa_w2",    # rwkv6
    "model.layers.{bid}.self_attn.time_maa_w2",   # rwkv6qwen2
    "model.layers.{bid}.attention.w2",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_A0: (
    "model.layers.{bid}.attention.a0",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_A1: (
    "model.layers.{bid}.attention.a1",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_A2: (
    "model.layers.{bid}.attention.a2",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_V0: (
    "model.layers.{bid}.attention.v0",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_V1: (
    "model.layers.{bid}.attention.v1",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_V2: (
    "model.layers.{bid}.attention.v2",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_G1: (
    "model.layers.{bid}.attention.g1",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_G2: (
    "model.layers.{bid}.attention.g2",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_K_K: (
    "model.layers.{bid}.attention.k_k",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_K_A: (
    "model.layers.{bid}.attention.k_a",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_R_K: (
    "model.layers.{bid}.attention.r_k",            # rwkv7
),

MODEL_TENSOR.TIME_MIX_LERP_X: (
    "rwkv.blocks.{bid}.attention.time_maa_x",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_x",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LERP_K: (
    "rwkv.blocks.{bid}.attention.time_maa_k",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_k",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LERP_V: (
    "rwkv.blocks.{bid}.attention.time_maa_v",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_v",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LERP_R: (
    "rwkv.blocks.{bid}.attention.time_maa_r",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_r",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LERP_G: (
    "rwkv.blocks.{bid}.attention.time_maa_g",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_g",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LERP_W: (
    "rwkv.blocks.{bid}.attention.time_maa_w",   # rwkv6
    "model.layers.{bid}.self_attn.time_maa_w",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_FIRST: (
    "rwkv.blocks.{bid}.attention.time_faaaa",   # rwkv6
),

MODEL_TENSOR.TIME_MIX_DECAY: (
    "rwkv.blocks.{bid}.attention.time_decay",   # rwkv6
    "model.layers.{bid}.self_attn.time_decay",  # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_DECAY_W1: (
    "rwkv.blocks.{bid}.attention.time_decay_w1",  # rwkv6
    "model.layers.{bid}.self_attn.time_decay_w1", # rwkv6qwen2
),
MODEL_TENSOR.TIME_MIX_DECAY_W2: (
    "rwkv.blocks.{bid}.attention.time_decay_w2",  # rwkv6
    "model.layers.{bid}.self_attn.time_decay_w2", # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_KEY: (
    "rwkv.blocks.{bid}.attention.key",     # rwkv6
    "model.layers.{bid}.self_attn.k_proj", # rwkv6qwen2
    "model.layers.{bid}.attention.key",    # rwkv7
    "model.layers.{bid}.attention.k_proj", # rwkv7
),

MODEL_TENSOR.TIME_MIX_VALUE: (
    "rwkv.blocks.{bid}.attention.value",   # rwkv6
    "model.layers.{bid}.self_attn.v_proj", # rwkv6qwen2
    "model.layers.{bid}.attention.value",  # rwkv7
    "model.layers.{bid}.attention.v_proj", # rwkv7
),

MODEL_TENSOR.TIME_MIX_RECEPTANCE: (
    "rwkv.blocks.{bid}.attention.receptance",  # rwkv6
    "model.layers.{bid}.self_attn.q_proj",     # rwkv6qwen2
    "model.layers.{bid}.attention.receptance", # rwkv7
    "model.layers.{bid}.attention.r_proj",     # rwkv7
),

MODEL_TENSOR.TIME_MIX_GATE: (
    "rwkv.blocks.{bid}.attention.gate",        # rwkv6
    "model.layers.{bid}.self_attn.gate",       # rwkv6qwen2
),

MODEL_TENSOR.TIME_MIX_LN: (
    "rwkv.blocks.{bid}.attention.ln_x", # rwkv6
    "model.layers.{bid}.attention.ln_x" # rwkv7
),

MODEL_TENSOR.TIME_MIX_OUTPUT: (
    "rwkv.blocks.{bid}.attention.output",  # rwkv6
    "model.layers.{bid}.self_attn.o_proj", # rwkv6qwen2
    "model.layers.{bid}.attention.output", # rwkv7
    "model.layers.{bid}.attention.o_proj", # rwkv7
),

MODEL_TENSOR.CHANNEL_MIX_LERP_K: (
    "rwkv.blocks.{bid}.feed_forward.time_maa_k", # rwkv6
    "model.layers.{bid}.feed_forward.x_k",       # rwkv7
),

MODEL_TENSOR.CHANNEL_MIX_LERP_R: (
    "rwkv.blocks.{bid}.feed_forward.time_maa_r", # rwkv6
),

MODEL_TENSOR.CHANNEL_MIX_KEY: (
    "rwkv.blocks.{bid}.feed_forward.key",  # rwkv6
```

```
        "model.layers.{bid}.feed_forward.key", # rwkv7
    ),

    MODEL_TENSOR.CHANNEL_MIX_RECEPTANCE: (
        "rwkv.blocks.{bid}.feed_forward.receptance", # rwkv6
    ),

    MODEL_TENSOR.CHANNEL_MIX_VALUE: (
        "rwkv.blocks.{bid}.feed_forward.value",  # rwkv6
        "model.layers.{bid}.feed_forward.value", # rwkv7
    ),

    MODEL_TENSOR.ATTN_Q_A: (
        "model.layers.{bid}.self_attn.q_a_proj", # deepseek2
    ),

    MODEL_TENSOR.ATTN_Q_B: (
        "model.layers.{bid}.self_attn.q_b_proj", # deepseek2
    ),

    MODEL_TENSOR.ATTN_KV_A_MQA: (
        "model.layers.{bid}.self_attn.kv_a_proj_with_mqa", # deepseek2
    ),

    MODEL_TENSOR.ATTN_KV_B: (
        "model.layers.{bid}.self_attn.kv_b_proj", # deepseek2
    ),

    MODEL_TENSOR.ATTN_Q_A_NORM: (
        "model.layers.{bid}.self_attn.q_a_layernorm", # deepseek2
    ),

    MODEL_TENSOR.ATTN_KV_A_NORM: (
        "model.layers.{bid}.self_attn.kv_a_layernorm", # deepseek2
    ),

    MODEL_TENSOR.ATTN_SUB_NORM: (
        "model.layers.{bid}.self_attn.inner_attn_ln",  # bitnet
    ),

    MODEL_TENSOR.FFN_SUB_NORM: (
        "model.layers.{bid}.mlp.ffn_layernorm",  # bitnet
    ),

    MODEL_TENSOR.DEC_ATTN_NORM: (
        "decoder.block.{bid}.layer.0.layer_norm", # t5
    ),

    MODEL_TENSOR.DEC_ATTN_Q: (
        "decoder.block.{bid}.layer.0.SelfAttention.q", # t5
    ),

    MODEL_TENSOR.DEC_ATTN_K: (
        "decoder.block.{bid}.layer.0.SelfAttention.k", # t5
    ),

    MODEL_TENSOR.DEC_ATTN_V: (
        "decoder.block.{bid}.layer.0.SelfAttention.v", # t5
    ),

    MODEL_TENSOR.DEC_ATTN_OUT: (
        "decoder.block.{bid}.layer.0.SelfAttention.o", # t5
    ),

    MODEL_TENSOR.DEC_ATTN_REL_B: (
        "decoder.block.{bid}.layer.0.SelfAttention.relative_attention_bias", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_NORM: (
        "decoder.block.{bid}.layer.1.layer_norm", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_Q: (
        "decoder.block.{bid}.layer.1.EncDecAttention.q", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_K: (
        "decoder.block.{bid}.layer.1.EncDecAttention.k", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_V: (
        "decoder.block.{bid}.layer.1.EncDecAttention.v", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_OUT: (
        "decoder.block.{bid}.layer.1.EncDecAttention.o", # t5
    ),

    MODEL_TENSOR.DEC_CROSS_ATTN_REL_B: (
        "decoder.block.{bid}.layer.1.EncDecAttention.relative_attention_bias", # t5
    ),

    MODEL_TENSOR.DEC_FFN_NORM: (
        "decoder.block.{bid}.layer.2.layer_norm", # t5
    ),

    MODEL_TENSOR.DEC_FFN_GATE: (
        "decoder.block.{bid}.layer.2.DenseReluDense.wi_0", # flan-t5
    ),

    MODEL_TENSOR.DEC_FFN_UP: (
        "decoder.block.{bid}.layer.2.DenseReluDense.wi",   # t5
        "decoder.block.{bid}.layer.2.DenseReluDense.wi_1", # flan-t5
    ),

    MODEL_TENSOR.DEC_FFN_DOWN: (
        "decoder.block.{bid}.layer.2.DenseReluDense.wo", # t5
    ),
    MODEL_TENSOR.DEC_OUTPUT_NORM: (
        "decoder.final_layer_norm", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_NORM: (
        "encoder.block.{bid}.layer.0.layer_norm", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_Q: (
        "encoder.block.{bid}.layer.0.SelfAttention.q", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_K: (
        "encoder.block.{bid}.layer.0.SelfAttention.k", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_V: (
        "encoder.block.{bid}.layer.0.SelfAttention.v", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_OUT: (
        "encoder.block.{bid}.layer.0.SelfAttention.o", # t5
    ),

    MODEL_TENSOR.ENC_ATTN_REL_B: (
        "encoder.block.{bid}.layer.0.SelfAttention.relative_attention_bias", # t5
    ),

    MODEL_TENSOR.ENC_FFN_NORM: (
        "encoder.block.{bid}.layer.1.layer_norm", # t5
    ),

    MODEL_TENSOR.ENC_FFN_GATE: (
        "encoder.block.{bid}.layer.1.DenseReluDense.wi_0", # flan-t5
    ),

    MODEL_TENSOR.ENC_FFN_UP: (
        "encoder.block.{bid}.layer.1.DenseReluDense.wi",   # t5
        "encoder.block.{bid}.layer.1.DenseReluDense.wi_1", # flan-t5
    ),

    MODEL_TENSOR.ENC_FFN_DOWN: (
        "encoder.block.{bid}.layer.1.DenseReluDense.wo", # t5
    ),

    ############################################################################
    # TODO: these do not belong to block_mappings_cfg - move them to mappings_cfg
    MODEL_TENSOR.ENC_OUTPUT_NORM: (
        "encoder.final_layer_norm", # t5
    ),

    MODEL_TENSOR.CLS: (
        "classifier",       # jina
```

```
        "classifier.dense", # roberta
    ),

    MODEL_TENSOR.CLS_OUT: (
        "classifier.out_proj", # roberta
    ),
    #############################################################################

    MODEL_TENSOR.CONVNEXT_DW: (
        "backbone.convnext.{bid}.dwconv", # wavtokenizer
    ),

    MODEL_TENSOR.CONVNEXT_NORM: (
        "backbone.convnext.{bid}.norm", # wavtokenizer
    ),

    MODEL_TENSOR.CONVNEXT_PW1: (
        "backbone.convnext.{bid}.pwconv1", # wavtokenizer
    ),

    MODEL_TENSOR.CONVNEXT_PW2: (
        "backbone.convnext.{bid}.pwconv2", # wavtokenizer
    ),

    MODEL_TENSOR.CONVNEXT_GAMMA: (
        "backbone.convnext.{bid}.gamma", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_CONV1: (
        "backbone.posnet.{bid}.conv1", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_CONV2: (
        "backbone.posnet.{bid}.conv2", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_NORM: (
        "backbone.posnet.{bid}.norm", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_NORM1: (
        "backbone.posnet.{bid}.norm1", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_NORM2: (
        "backbone.posnet.{bid}.norm2", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_ATTN_NORM: (
        "backbone.posnet.{bid}.norm", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_ATTN_Q: (
        "backbone.posnet.{bid}.q", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_ATTN_K: (
        "backbone.posnet.{bid}.k", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_ATTN_V: (
        "backbone.posnet.{bid}.v", # wavtokenizer
    ),

    MODEL_TENSOR.POSNET_ATTN_OUT: (
        "backbone.posnet.{bid}.proj_out", # wavtokenizer
    ),
}

# architecture-specific block mappings
arch_block_mappings_cfg: dict[MODEL_ARCH, dict[MODEL_TENSOR, tuple[str, ...]]] = {
    MODEL_ARCH.ARCTIC: {
        MODEL_TENSOR.FFN_NORM: (
            "model.layers.{bid}.residual_layernorm",
        ),
        MODEL_TENSOR.FFN_NORM_EXP: (
            "model.layers.{bid}.post_attention_layernorm",
        ),
    },
}

mapping: dict[str, tuple[MODEL_TENSOR, str]]

def __init__(self, arch: MODEL_ARCH, n_blocks: int):
    self.mapping = {}
    for tensor, keys in self.mappings_cfg.items():
        if tensor not in MODEL_TENSORS[arch]:
            continue
        tensor_name = TENSOR_NAMES[tensor]
        self.mapping[tensor_name] = (tensor, tensor_name)
        for key in keys:
            self.mapping[key] = (tensor, tensor_name)
    if arch in self.arch_block_mappings_cfg:
        self.block_mappings_cfg.update(self.arch_block_mappings_cfg[arch])
    for bid in range(n_blocks):
        for tensor, keys in self.block_mappings_cfg.items():
            if tensor not in MODEL_TENSORS[arch]:
                continue

            tensor_name = TENSOR_NAMES[tensor].format(bid = bid)
            self.mapping[tensor_name] = (tensor, tensor_name)
            for key in keys:
                key = key.format(bid = bid)
                self.mapping[key] = (tensor, tensor_name)

def get_type_and_name(self, key: str, try_suffixes: Sequence[str] = ()) -> tuple[MODEL_TENSOR, str] | None:
    result = self.mapping.get(key)
    if result is not None:
        return result
    for suffix in try_suffixes:
        if key.endswith(suffix):
            result = self.mapping.get(key[:-len(suffix)])
            if result is not None:
                return result[0], result[1] + suffix
    return None

def get_name(self, key: str, try_suffixes: Sequence[str] = ()) -> str | None:
    result = self.get_type_and_name(key, try_suffixes = try_suffixes)
    if result is None:
        return None
    return result[1]

def get_type(self, key: str, try_suffixes: Sequence[str] = ()) -> MODEL_TENSOR | None:
    result = self.get_type_and_name(key, try_suffixes = try_suffixes)
    if result is None:
        return None
    return result[0]

def __getitem__(self, key: str) -> str:
    try:
        return self.mapping[key][1]
    except KeyError:
        raise KeyError(key)

def __contains__(self, key: str) -> bool:
    return key in self.mapping

def __repr__(self) -> str:
    return repr(self.mapping)


def get_tensor_name_map(arch: MODEL_ARCH, n_blocks: int) -> TensorNameMap:
return TensorNameMap(arch, n_blocks)

==== test-tokenizer-0.py ====
import time
import argparse

from transformers import AutoTokenizer

parser = argparse.ArgumentParser()
parser.add_argument("dir_tokenizer", help="directory containing 'tokenizer.model' file")
parser.add_argument("--fname-tok",   help="path to a text file to tokenize", required=True)
args = parser.parse_args()

dir_tokenizer = args.dir_tokenizer
fname_tok = args.fname_tok

tokenizer = AutoTokenizer.from_pretrained(dir_tokenizer)

print('tokenizing file: ', fname_tok) # noqa: NP100
fname_out = fname_tok + '.tok'
```

```
with open(fname_tok, 'r', encoding='utf-8') as f:
    lines = f.readlines()
    s = ''.join(lines)
    t_start = time.time()
    res = tokenizer.encode(s, add_special_tokens=False)
    t_end = time.time()
    print('\nmain : tokenized in', "{:.3f}".format(1000.0 * (t_end - t_start)), 'ms (py)') # noqa: NP100
    with open(fname_out, 'w', encoding='utf-8') as f:
        for x in res:
            # LLaMA v3 for some reason strips the space for these tokens (and others)
            # if x == 662:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # elif x == 1174:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # elif x == 2564:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # elif x == 758:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # elif x == 949:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # elif x == 5354:
            #     f.write(str(x) + ' \' ' + tokenizer.decode(x) + '\'\n')
            # else:
            #     f.write(str(x) + ' \'' + tokenizer.decode(x) + '\'\n')
            # f.write(str(x) + ' \'' + tokenizer.decode(x).strip() + '\'\n')
            f.write(str(x) + '\n')
    print('len(res): ', len(res)) # noqa: NP100
    print('len(lines): ', len(lines)) # noqa: NP100
print('results written to: ', fname_out) # noqa: NP100

==== test-tokenizer-random.py ====
# Test libllama tokenizer == AutoTokenizer.
# Brute force random words/text generation.
#
# Sample usage:
#
#   python3 tests/test-tokenizer-random.py ./models/ggml-vocab-llama-bpe.gguf ./models/tokenizers/llama-bpe
#

from __future__ import annotations

import time
import logging
import argparse
import subprocess
import random
import unicodedata

from pathlib import Path
from typing import Any, Iterator, cast
from typing_extensions import Buffer

import cffi
from transformers import AutoTokenizer, PreTrainedTokenizer
```

```
logger = logging.getLogger("test-tokenizer-random")


class LibLlama:

    DEFAULT_PATH_LLAMA_H = "./include/llama.h"
    DEFAULT_PATH_INCLUDES = ["./ggml/include/", "./include/"]
    DEFAULT_PATH_LIBLLAMA = "./build/src/libllama.so"  # CMakeLists.txt: BUILD_SHARED_LIBS ON

     def __init__(self, path_llama_h: str | None = None, path_includes: list[str] = [], path_libllama: str | None = None):
        path_llama_h = path_llama_h or self.DEFAULT_PATH_LLAMA_H
        path_includes = path_includes or self.DEFAULT_PATH_INCLUDES
        path_libllama = path_libllama or self.DEFAULT_PATH_LIBLLAMA
        (self.ffi, self.lib) = self._load_libllama_cffi(path_llama_h, path_includes, path_libllama)
        self.lib.llama_backend_init()

       def _load_libllama_cffi(self, path_llama_h: str, path_includes: list[str], path_libllama: str) -> tuple[cffi.FFI, Any]:
        cmd = ["gcc", "-O0", "-E", "-P", "-D__restrict=", "-D__attribute__(x)=", "-D__asm__(x)="]
        cmd += ["-I" + path for path in path_includes] + [path_llama_h]
        res = subprocess.run(cmd, stdout=subprocess.PIPE)
        assert (res.returncode == 0)
        source = res.stdout.decode()
        ffi = cffi.FFI()
        if True:  # workarounds for pycparser
            source = "typedef struct { } __builtin_va_list;" + "\n" + source
            source = source.replace("sizeof (int)",    str(ffi.sizeof("int")))
            source = source.replace("sizeof (void *)", str(ffi.sizeof("void*")))
            source = source.replace("sizeof (size_t)", str(ffi.sizeof("size_t")))
            source = source.replace("sizeof(int32_t)", str(ffi.sizeof("int32_t")))
        ffi.cdef(source, override=True)
        lib = ffi.dlopen(path_libllama)
        return (ffi, lib)

    def model_default_params(self, **kwargs):
        mparams = self.lib.llama_model_default_params()
        for k, v in kwargs.items():
            setattr(mparams, k, v)
        return mparams

    def context_default_params(self, **kwargs):
        cparams = self.lib.llama_context_default_params()
        for k, v in kwargs.items():
            setattr(cparams, k, v)
        return cparams


class LibLlamaModel:

    def __init__(self, libllama: LibLlama, path_model: str, mparams={}, cparams={}):
        self.lib: Any = libllama.lib
        self.ffi = libllama.ffi
        if isinstance(mparams, dict):
            mparams = libllama.model_default_params(**mparams)
        self.model = self.lib.llama_model_load_from_file(path_model.encode(), mparams)
        if not self.model:
            raise RuntimeError("error: failed to load model '%s'" % path_model)
        if isinstance(cparams, dict):
            cparams = libllama.context_default_params(**cparams)
        self.ctx = self.lib.llama_new_context_with_model(self.model, cparams)
        if not self.ctx:
            raise RuntimeError("error: failed to create context for model '%s'" % path_model)
        n_tokens_max = self.lib.llama_n_ctx(self.ctx)
        self.token_ids = self.ffi.new("llama_token[]", n_tokens_max)
        self.text_buff = self.ffi.new("uint8_t[]", 1024)

    def free(self):
        if self.ctx:
            self.lib.llama_free(self.ctx)
        if self.model:
            self.lib.llama_model_free(self.model)
        self.ctx = None
        self.model = None
        self.lib = None

    def tokenize(self, text: str, add_special: bool = False, parse_special: bool = False) -> list[int]:
        encoded_text: bytes = text.encode("utf-8")
                num = self.lib.llama_tokenize(self.model, encoded_text, len(encoded_text), self.token_ids,
len(self.token_ids), add_special, parse_special)
        while num < 0 and len(self.token_ids) < (16 << 20):
            self.token_ids = self.ffi.new("llama_token[]", -2 * num)
                  num = self.lib.llama_tokenize(self.model, encoded_text, len(encoded_text), self.token_ids,
len(self.token_ids), add_special, parse_special)
        return list(self.token_ids[0:num])

    def detokenize(self, ids: list[int], remove_special: bool = False, unparse_special: bool = False) -> str:
        if len(self.token_ids) < len(ids):
            self.token_ids = self.ffi.new("llama_token[]", 2 * len(ids))
        for i, id in enumerate(ids):
            self.token_ids[i] = id
                  num = self.lib.llama_detokenize(self.model, self.token_ids, len(ids), self.text_buff,
len(self.text_buff), remove_special, unparse_special)
        while num < 0 and len(self.text_buff) < (16 << 20):
            self.text_buff = self.ffi.new("uint8_t[]", -2 * num)
                     num = self.lib.llama_detokenize(self.model, self.token_ids, len(ids), self.text_buff,
len(self.text_buff), remove_special, unparse_special)
         return str(cast(Buffer, self.ffi.buffer(self.text_buff, num)), encoding="utf-8", errors="replace")  #
replace errors with '\uFFFD'


class Tokenizer:

    def encode(self, text: str) -> list[int]:
        raise NotImplementedError
    def decode(self, ids: list[int]) -> str:
        raise NotImplementedError


class TokenizerGroundtruth (Tokenizer):

    def __init__(self, dir_tokenizer: str):
        self.model: PreTrainedTokenizer = AutoTokenizer.from_pretrained(dir_tokenizer)
        # guess BOS and EOS
        ids = self.encode("a")
        assert 1 <= len(ids) <= 3
        add_bos_token = len(ids) > 1 and self.model.bos_token_id == ids[0]
        add_eos_token = len(ids) > 1 and self.model.eos_token_id == ids[-1]
        self.add_bos_token = getattr(self.model, "add_bos_token", add_bos_token)
        self.add_eos_token = getattr(self.model, "add_eos_token", add_eos_token)
        # build vocab
        tokens = list(self.model.get_vocab().values())
        self.vocab = self.model.batch_decode(tokens, skip_special_tokens=True)
        self.vocab = list(sorted(self.vocab))
        # tokens and lists
        self.special_tokens = list(self.model.all_special_tokens)
                   self.added_tokens       =  self.model.batch_decode(self.model.added_tokens_encoder.values(),
skip_special_tokens=False)
        self.bos_token = self.model.bos_token
        self.eos_token = self.model.eos_token

    def encode(self, text: str) -> list[int]:
        return self.model.encode(text, add_special_tokens=True)

    def decode(self, ids: list[int]) -> str:
        return self.model.decode(ids, skip_special_tokens=False)


class TokenizerLlamaCpp (Tokenizer):

    libllama: LibLlama | None = None

    def __init__(self, vocab_file: str):
        if not self.libllama:
            self.libllama = LibLlama()
                    self.model   =   LibLlamaModel(self.libllama,   vocab_file,   mparams=dict(vocab_only=True),
cparams=dict(n_ctx=4096))

    def encode(self, text: str) -> list[int]:
        return self.model.tokenize(text, add_special=True, parse_special=True)

    def decode(self, ids: list[int]) -> str:
        return self.model.detokenize(ids, remove_special=False, unparse_special=True)


def generator_custom_text() -> Iterator[str]:
    """General tests"""
    yield from [
        "",
        " ",
        "  ",
        "   ",
        "\t",
        "\n",
        "\n\n",
        "\n\n\n",
        "\t\n",
        "Hello world",
        " Hello world",
        "Hello World",
        " Hello World",
        " Hello World!",
        "Hello, world!",
        " Hello, world!",
        " this is ?.cpp",
        "w048 7tuijk dsdfhu",
        "???? ?? ?????????",
        "???????????????????",
        "LAUNCH (normal) ???? (multiple emojis concatenated) [OK] (only emoji that has its own token)",
        "Hello",
        " Hello",
        "  Hello",
        "   Hello",
        "    Hello",
        "    Hello\n    Hello",
        " (",
        "\n =",
        "' era",
        "Hello, y'all! How are you ? ????apple??1314151??",
        "3",
        "33",
        "333",
        "3333",
        "33333",
        "333333",
        "3333333",
        "33333333",
        "333333333",
    ]


def generator_custom_text_edge_cases() -> Iterator[str]:
    """Edge cases found while debugging"""
    yield from [
        '\x1f-a',     # unicode_ranges_control, {0x00001C, 0x00001F}
        '?-a',        # unicode_ranges_digit, 0x00BC
        '?-a',        # unicode_ranges_digit, 0x00BD
        '?-a',        # unicode_ranges_digit, 0x00BE
        'a ?b',      # unicode_ranges_digit, 0x3007
        '?-a',       # unicode_ranges_digit, {0x00002150, 0x0000218F} // Number Forms
        '\uFEFF//',   # unicode_ranges_control, 0xFEFF (BOM)
        'C?a Vi?t',   # llama-3, ignore_merges = true
        '<s>a',       # Phi-3 fail
        '<unk><|endoftext|><s>',  # Phi-3 fail
        'a\na',            # bert fail
        '"`',              # falcon
        ' \u2e4e',         # falcon
        '\n\x0b  ',        # falcon
        'a\xa0\xa0\x00b',  # jina-v2-es
        'one <mask>',      # jina-v2-es  <mask> lstrip=true
        'a </s> b',        # rstrip phi-3
        'a <mask> b',      # lstrip jina-v2
        '\xa0aC',          # deepseek
        '\u2029 \uA3E4',   # deepseek-llm
        "a ?",
        'a?',               # mpt
        '\U000ac517',      # utf-8 encode error, falcon
        '\U000522f4',      # utf-8 encode error, starcoder
        "<s><s><unk><s>a<s>b<s>c<unk>d<unk></s>",
        "<s> <s> <unk><s>a<s>b<s>c<unk>d<unk></s>",
    ]


def generator_vocab_words(tokenizer: TokenizerGroundtruth) -> Iterator[str]:
    """Brute force check all vocab words"""
    yield from tokenizer.vocab


def generator_ascii_lr_strip() -> Iterator[str]:
    WHITESPACES = ["", " ", "  "]
    CHARACTERS = list(chr(i) for i in range(1, 0x80)) + [""]
    for char1 in CHARACTERS:
        for char2 in CHARACTERS:
            for lstrip in WHITESPACES:
                for rstrip in WHITESPACES:
                    yield lstrip + char1 + char2 + rstrip
                    yield lstrip + char1 + rstrip + char2
                    yield char1 + lstrip + char2 + rstrip


def generator_apostrophe() -> Iterator[str]:
    WHITESPACES = ["", " ", "  "]
    CHARACTERS = list(chr(i) for i in range(1, 0x80)) + [""]
    for char1 in CHARACTERS:
        for char2 in CHARACTERS:
            for lstrip in WHITESPACES:
                for rstrip in WHITESPACES:
                    yield char1 + lstrip + "'" + rstrip + char2
                    yield char1 + char2 + lstrip + "'" + rstrip + "z"
                    yield "a" + lstrip + "'" + rstrip + char1 + char2


def generator_added_lr_strip(tokenizer: TokenizerGroundtruth) -> Iterator[str]:
    WHITESPACES = ["", " ", "  ", "\n", "\r\n", "\n\n", "\t", "\t\t"]
    all_tokens = list(sorted(set(tokenizer.special_tokens + tokenizer.added_tokens)))
    for token in all_tokens:
        for lstrip in WHITESPACES:
            for rstrip in WHITESPACES:
                yield lstrip + token + rstrip
                yield "a" + lstrip + token + rstrip
                yield lstrip + token + rstrip + "z"
                yield "a" + lstrip + token + rstrip + "z"


def generator_random_added_tokens(tokenizer: TokenizerGroundtruth, iterations=100) -> Iterator[str]:
    separations = [" ", "\n", "\t", "-", "!", "one", "1", "<s>", "</s>"]
    all_tokens  = list(sorted(set(tokenizer.special_tokens + tokenizer.added_tokens + separations)))
    rand = random.Random()
    for m in range(iterations):
        rand.seed(m)
        words = rand.choices(all_tokens, k=500)
        if words and words[0] == tokenizer.bos_token:  # skip spam warning of double BOS
            while len(words) > 1 and words[1] == tokenizer.bos_token:  # leave one starting BOS
                words.pop(0)
            if tokenizer.add_bos_token:  # drop all starting BOS
                words.pop(0)
        if words and words[-1] == tokenizer.eos_token:  # skip spam warning of double EOS
            while len(words) > 1 and words[-2] == tokenizer.eos_token:  # leave one trailing EOS
                words.pop(-1)
            if tokenizer.add_bos_token:  # drop all trailing EOS
                words.pop(-1)
        yield "".join(words)


def generator_random_chars(iterations=100) -> Iterator[str]:
    """Brute force random text with simple characters"""

    NUM_WORDS = 400
    WHITESPACES = list(" " * 20 + "\n" * 5 + "\r\n" * 5 + "\t" * 5)
    CHARS = list(sorted(set("""
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
        abcdefghijklmnopqrstuvwxyz
        ????????????????????
        ????????????????????
        .-,*/-+?!"?$%&/()=??[]{}<>\\|@#~??~;:_
    """)))

    rand = random.Random()
    for m in range(iterations):
        rand.seed(m)
        text = []
        for _ in range(NUM_WORDS):
            k = rand.randint(1, 7)
            word = rand.choices(CHARS, k=k)
            word.append(rand.choice(WHITESPACES))
            text.append("".join(word))
        yield "".join(text)


def generator_unicodes() -> Iterator[str]:
    """Iterate unicode characters"""
    MAX_CODEPOINTS = 0x30000  # 0x110000

    def _valid(cpt):
        if cpt >= 0x30000:  # unassigned and supplement?ary
            return False
        # if cpt == 0x2029:  # deepseek-llm
        #    return False
        if unicodedata.category(chr(cpt)) in ("Cn", "Cs", "Co"):  # undefined, surrogates, private
            return False
        return True

    characters = [chr(cpt) for cpt in range(0, MAX_CODEPOINTS) if _valid(cpt)]

    yield from characters


def generator_random_unicodes(iterations=100) -> Iterator[str]:
    """Brute force random text with unicode characters"""

    NUM_WORDS = 200
    WHITESPACES = list(" " * 20 + "\n" * 5 + "\r\n" * 5 + "\t" * 5)

    characters = list(generator_unicodes())

    rand = random.Random()
    for m in range(iterations):
        rand.seed(m)
        text = []
        for _ in range(NUM_WORDS):
            k = rand.randint(1, 7)
            word = rand.choices(characters, k=k)
            word.append(rand.choice(WHITESPACES))
            text.append("".join(word))
        yield "".join(text)


def generator_random_vocab_chars(tokenizer: TokenizerGroundtruth, iterations=100) -> Iterator[str]:
    """Brute force random text with vocab characters"""

    vocab_chars = set()
    for word in tokenizer.vocab:
        vocab_chars.update(word)
    vocab_chars = list(sorted(vocab_chars))

    rand = random.Random()
    for m in range(iterations):
        rand.seed(m)
        text = rand.choices(vocab_chars, k=1024)
        yield "".join(text)


def generator_random_vocab_words(tokenizer: TokenizerGroundtruth, iterations=100) -> Iterator[str]:
    """Brute force random text from vocab words"""
    vocab = [w.strip() for w in tokenizer.vocab]
    yield from vocab

    rand = random.Random()
    for m in range(iterations):
        rand.seed(m)
        text = []
        num_words = rand.randint(300, 400)
        for i in range(num_words):
            k = rand.randint(1, 3)
            words = rand.choices(vocab, k=k)
            sep = rand.choice("     \n\r\t")
            text.append("".join(words) + sep)
        yield "".join(text)


def compare_tokenizers(tokenizer1: TokenizerGroundtruth, tokenizer2: TokenizerLlamaCpp, generator: Iterator[str]):

    def find_first_mismatch(ids1: list[int] | str, ids2: list[int] | str):
        for i, (a, b) in enumerate(zip(ids1, ids2)):
            if a != b:
                return i
        if len(ids1) == len(ids2):
            return -1
        return min(len(ids1), len(ids2))

    def check_detokenizer(text: str, text1: str, text2: str) -> bool:
        if text1 == text2:  # equal to TokenizerGroundtruth?
            return True
        # equal to source text?
        if tokenizer1.add_bos_token:  # remove BOS
            if text2.startswith(tokenizer1.bos_token):
                text2 = text2[len(tokenizer1.bos_token):]
        if tokenizer1.add_eos_token:  # remove EOS
            if text2.endswith(tokenizer1.eos_token):
                text2 = text2[:-len(tokenizer1.eos_token)]
        return text == text2

    t_encode1 = 0
    t_encode2 = 0
    t_decode1 = 0
    t_decode2 = 0
    t_start = time.perf_counter()
    encode_errors = 0
    decode_errors = 0
    MAX_ERRORS = 10

    logger.info("%s: %s" % (generator.__qualname__, "ini"))
    for text in generator:
        # print(repr(text), text.encode())
        # print(repr(text), hex(ord(text[0])), text.encode())
        t0 = time.perf_counter()
        ids1 = tokenizer1.encode(text)
        t1 = time.perf_counter()
        ids2 = tokenizer2.encode(text)
        t2 = time.perf_counter()
        text1 = tokenizer1.decode(ids1)
        t3 = time.perf_counter()
        text2 = tokenizer2.decode(ids1)
        t4 = time.perf_counter()
        t_encode1 += t1 - t0
        t_encode2 += t2 - t1
        t_decode1 += t3 - t2
        t_decode2 += t4 - t3
        if encode_errors < MAX_ERRORS and ids1 != ids2:
            i = find_first_mismatch(ids1, ids2)
            ids1 = list(ids1)[max(0, i - 2) : i + 5 + 1]
            ids2 = list(ids2)[max(0, i - 2) : i + 5 + 1]
            logger.error(" Expected: " + str(ids1))
            logger.error("   Result: " + str(ids2))
            encode_errors += 1
            logger.error(f" {encode_errors=}")
        if decode_errors < MAX_ERRORS and not check_detokenizer(text, text1, text2):
            i = find_first_mismatch(text1, text2)
            text1 = list(text1[max(0, i - 2) : i + 5 + 1])
            text2 = list(text2[max(0, i - 2) : i + 5 + 1])
            logger.error(" Expected: " + " ".join(hex(ord(x)) for x in text1))
            logger.error("   Result: " + " ".join(hex(ord(x)) for x in text2))
            decode_errors += 1
            logger.error(f" {decode_errors=}")
        if encode_errors >= MAX_ERRORS and decode_errors >= MAX_ERRORS:
            logger.error(f" EXIT: {encode_errors=} {decode_errors=}")
            # raise Exception()
            break

    t_total = time.perf_counter() - t_start
        logger.info(f"{generator.__qualname__}:  end,    {t_encode1=:.3f}  {t_encode2=:.3f}    {t_decode1=:.3f}
{t_decode2=:.3f}  {t_total=:.3f}")



def main(argv: list[str] | None = None):
    parser = argparse.ArgumentParser()
    parser.add_argument("vocab_file", type=str, help="path to vocab 'gguf' file")
    parser.add_argument("dir_tokenizer", type=str, help="directory containing 'tokenizer.model' file")
    parser.add_argument("--verbose", action="store_true", help="increase output verbosity")
    args = parser.parse_args(argv)

    logging.basicConfig(level = logging.DEBUG if args.verbose else logging.INFO)
    logger.info(f"VOCABFILE: '{args.vocab_file}'")

    tokenizer1 = TokenizerGroundtruth(args.dir_tokenizer)
    tokenizer2 = TokenizerLlamaCpp(args.vocab_file)

    # compare_tokenizers(tokenizer1, tokenizer2, generator_custom_text())
    # compare_tokenizers(tokenizer1, tokenizer2, generator_custom_text_edge_cases())
    compare_tokenizers(tokenizer1, tokenizer2, generator_ascii_lr_strip())
    compare_tokenizers(tokenizer1, tokenizer2, generator_apostrophe())
    compare_tokenizers(tokenizer1, tokenizer2, generator_unicodes())
    compare_tokenizers(tokenizer1, tokenizer2, generator_vocab_words(tokenizer1))
    compare_tokenizers(tokenizer1, tokenizer2, generator_added_lr_strip(tokenizer1))
    # compare_tokenizers(tokenizer1, tokenizer2, generator_random_added_tokens(tokenizer1, 10_000))
    # compare_tokenizers(tokenizer1, tokenizer2, generator_random_chars(10_000))
    # compare_tokenizers(tokenizer1, tokenizer2, generator_random_unicodes(10_000))
    # compare_tokenizers(tokenizer1, tokenizer2, generator_random_vocab_chars(tokenizer1, 10_000))
    # compare_tokenizers(tokenizer1, tokenizer2, generator_random_vocab_words(tokenizer1, 5_000))

    tokenizer2.model.free()


if __name__ == "__main__":
    # main()

    if True:
        logging.basicConfig(
            level    = logging.DEBUG,
            format   = "%(asctime)s.%(msecs)03d %(name)s %(levelname)s %(message)s",
            datefmt  = "%Y-%m-%d %H:%M:%S",
            filename = logger.name + ".log",
            filemode = "a"
        )
    logging.basicConfig(
        level    = logging.DEBUG,
        format   = "%(levelname)s %(message)s",
    )

    path_tokenizers   = Path("./models/tokenizers/")
    path_vocab_format = "./models/ggml-vocab-%s.gguf"

    tokenizers = [
        "llama-spm",      # SPM
        "phi-3",          # SPM
        "gemma",          # SPM
        "gemma-2",        # SPM
        "baichuan",       # SPM
        "bert-bge",       # WPM
        "jina-v2-en",     # WPM
        "llama-bpe",      # BPE
        "phi-2",          # BPE
        "deepseek-llm",   # BPE
        "deepseek-coder", # BPE
        "falcon",         # BPE
        "mpt",            # BPE
        "starcoder",      # BPE
        "gpt-2",          # BPE
        "stablelm2",      # BPE
        "refact",         # BPE
        "qwen2",          # BPE
        "olmo",           # BPE
        "jina-v2-es",     # BPE
        "jina-v2-de",     # BPE
        "smaug-bpe",      # BPE
        "poro-chat",      # BPE
        "jina-v2-code",   # BPE
        "viking",         # BPE
        "jais",           # BPE
    ]

    logger.info("=" * 50)
    for tokenizer in tokenizers:
        logger.info("-" * 50)
        logger.info(f"TOKENIZER: '{tokenizer}'")
        vocab_file = Path(path_vocab_format % tokenizer)
        dir_tokenizer = path_tokenizers / tokenizer
        main([str(vocab_file), str(dir_tokenizer), "--verbose"])

==== test_network.py ====
import pytest
import torch

from crm.core import Network


def test_network():
    assert Network(1, [[0]])


def test_topological_sort():
    n = Network(4, [[2, 3], [3], [3], []])
    assert n._topological_sort() == [1, 0, 2, 3]

    n = Network(4, [[2, 3], [3], [], []])
    assert n._topological_sort() == [1, 0, 3, 2]

    n = Network(4, [[1], [2], [3], []])
    assert n._topological_sort() == [0, 1, 2, 3]


def test_forward():
    n = Network(4, [[1], [2], [3], []])
    n.weights = {
        (0, 1): torch.tensor(1.0, requires_grad=True),
        (1, 2): torch.tensor(2.0, requires_grad=True),
        (2, 3): torch.tensor(3.0, requires_grad=True),
    }
    assert n.forward({0: 1, 1: 1, 2: 1, 3: 1}) == torch.tensor(6.0).reshape(1, 1)

    n = Network(6, [[3], [3, 4], [4], [5], [5], []])
    n.weights = {
        (0, 3): torch.tensor(1.0, requires_grad=True),
        (1, 3): torch.tensor(2.0, requires_grad=True),
        (1, 4): torch.tensor(3.0, requires_grad=True),
        (2, 4): torch.tensor(4.0, requires_grad=True),
        (3, 5): torch.tensor(5.0, requires_grad=True),
        (4, 5): torch.tensor(6.0, requires_grad=True),
    }
    assert n.forward({0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1}) == torch.tensor(
        57.0
    ).reshape(1, 1)
    n = Network(5, [[3], [3, 4], [4], [], []])
    n.weights = {
        (0, 3): torch.tensor(1.0, requires_grad=True),
        (1, 3): torch.tensor(2.0, requires_grad=True),
        (1, 4): torch.tensor(3.0, requires_grad=True),
        (2, 4): torch.tensor(4.0, requires_grad=True),
    }

    assert torch.allclose(
        n.forward({0: 1, 1: 1, 2: 1, 3: 1, 4: 1}),
        torch.stack([torch.tensor(3.0), torch.tensor(7.0)]),
    )


def test_fast_forward():
    n = Network(4, [[1], [2], [3], []])
    n.weights = {
        (0, 1): torch.tensor(1.0, requires_grad=True),
        (1, 2): torch.tensor(2.0, requires_grad=True),
        (2, 3): torch.tensor(3.0, requires_grad=True),
    }
    assert n.fast_forward({0: 1, 1: 1, 2: 1, 3: 1}) == torch.tensor(6.0).reshape(1, 1)

    n = Network(6, [[3], [3, 4], [4], [5], [5], []])
    n.weights = {
        (0, 3): torch.tensor(1.0, requires_grad=True),
        (1, 3): torch.tensor(2.0, requires_grad=True),
        (1, 4): torch.tensor(3.0, requires_grad=True),
        (2, 4): torch.tensor(4.0, requires_grad=True),
        (3, 5): torch.tensor(5.0, requires_grad=True),
        (4, 5): torch.tensor(6.0, requires_grad=True),
    }
    assert n.forward({0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1}) == torch.tensor(
        57.0
    ).reshape(1, 1)
    n = Network(5, [[3], [3, 4], [4], [], []])
    n.weights = {
        (0, 3): torch.tensor(1.0, requires_grad=True),
        (1, 3): torch.tensor(2.0, requires_grad=True),
        (1, 4): torch.tensor(3.0, requires_grad=True),
        (2, 4): torch.tensor(4.0, requires_grad=True),
    }

    assert torch.allclose(
        n.forward({0: 1, 1: 1, 2: 1, 3: 1, 4: 1}),
        torch.stack([torch.tensor(3.0), torch.tensor(7.0)]),
    )


def test_no_double_forward():
    n = Network(4, [[1], [2], [3], []])
    n.forward({0: 1, 1: 1, 2: 1, 3: 1})
    with pytest.raises(Exception):
        n.forward({0: 1, 1: 1, 2: 1, 3: 1})


def test_set_neuron_activation():
    n = Network(4, [[1], [2], [3], []])
    n.weights = {
        (0, 1): torch.tensor(1.0, requires_grad=True),
        (1, 2): torch.tensor(2.0, requires_grad=True),
        (2, 3): torch.tensor(3.0, requires_grad=True),
    }

    assert n.forward({0: 1, 1: 1, 2: 1, 3: 1}) == torch.tensor(6.0).reshape(1, 1)
    n.reset()
    n.set_neuron_activation(
        3,
        lambda x: torch.exp(x.float()),
    )
    assert n.forward({0: 1, 1: 1, 2: 1, 3: 1}) == torch.exp(torch.tensor(6.0)).reshape(
        1, 1
    )


def test_assign_layers():
    n = Network(4, [[2], [2, 3], [3], []])
    assert n.neurons[0].layer == 0
    assert n.neurons[1].layer == 0
    assert n.neurons[2].layer == 1
    assert n.neurons[3].layer == 2


def test_lrp():
    n = Network(4, [[2], [2, 3], [3], []])
    n.weights = {
        (0, 2): torch.tensor(1.0, requires_grad=True),
        (1, 2): torch.tensor(3.0, requires_grad=True),
        (1, 3): torch.tensor(2.0, requires_grad=True),
        (2, 3): torch.tensor(2.0, requires_grad=True),
    }
    n.forward({0: -3, 1: 2, 2: 1, 3: 1})
    assert [n.neurons[i].value for i in range(len(n.neurons))] == [
        torch.tensor(-3),
        torch.tensor(2),
        torch.tensor(3.0),
        torch.tensor(10.0),
    ]
    n.lrp(torch.tensor(10.0), 3)
    assert [n.neurons[i].relevance for i in range(len(n.neurons))] == [
        torch.tensor(-6.0),
        torch.tensor(16.0),
        torch.tensor(6.0),
        torch.tensor(10.0),
    ]

    n = Network(7, [[4], [4], [5], [5], [6], [6], []])
    n.weights = {
        (0, 4): torch.tensor(6.0, requires_grad=True),
        (1, 4): torch.tensor(5.0, requires_grad=True),
        (2, 5): torch.tensor(4.0, requires_grad=True),
        (3, 5): torch.tensor(6.0, requires_grad=True),
        (4, 6): torch.tensor(4.0, requires_grad=True),
        (5, 6): torch.tensor(3.0, requires_grad=True),
    }
    n.forward({0: 1, 1: -1, 2: 2, 3: -1, 4: 1, 5: 1, 6: 1})
    print([n.neurons[i].value for i in range(len(n.neurons))])
    n.lrp(torch.tensor(10.0), 6)
    print([n.neurons[i].relevance for i in range(len(n.neurons))])
    # assert False

==== test_neuron.py ====
import torch

from crm.core import Neuron



def test_neuron():
    Neuron(0)
    Neuron(0, lambda x: 10 * x)



def test_successor():
    n = Neuron(0)
    successors = [Neuron(1), Neuron(2), Neuron(3)]
    n.set_successor_neurons(successors)
    assert n.successor_neurons == successors



def test_predecessor():
    n = Neuron(0)
    predecessors = [Neuron(1), Neuron(2), Neuron(3)]
    n.set_predecessor_neurons(predecessors)
    assert n.predecessor_neurons == predecessors



def test_activation():
    n = Neuron(0)
    assert n.activation_fn(torch.tensor(-1)) == torch.tensor(0)
    assert n.activation_fn(torch.tensor(10)) == torch.tensor(10)
    n = Neuron(0, lambda x: 10 * x)
    assert n.activation_fn(torch.tensor(1)) == torch.tensor(10)



# def test_activation_fn_grad():
#     n = Neuron(0)
#     assert n.activation_fn_grad(torch.tensor(-1)) == torch.tensor(0)
#     assert n.activation_fn_grad(torch.tensor(10)) == torch.tensor(1)
#     n = Neuron(0, lambda x: 10 * x)
#     assert n.activation_fn_grad(torch.tensor(12)) == torch.tensor(10)

==== test_utils.py ====

==== token_agent.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: token_agent.py
Purpose: Load YAML beliefs, walk symbolic paths, emit updates to cortex
"""

import os
import yaml
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  # Assumes cortex_bus has send_message function

FRAG_DIR = Path("fragments/core")

class TokenAgent:
    def __init__(self, agent_id="token_agent_01"):
        self.agent_id = agent_id
        self.frag_path = FRAG_DIR
        self.fragment_cache = []

    def load_fragments(self):
        files = list(self.frag_path.glob("*.yaml"))
        random.shuffle(files)
        for f in files:
            with open(f, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag:
                        self.fragment_cache.append((f, frag))
                except yaml.YAMLError as e:
                    print(f"[{self.agent_id}] YAML error in {f.name}: {e}")

    def walk_fragment(self, path, frag):
        # Walk logic example: shallow claim reassertion and mutation flag
        if 'claim' not in frag:
            return
        walk_log = {
            'fragment': path.name,
            'claim': frag['claim'],
            'tags': frag.get('tags', []),
            'confidence': frag.get('confidence', 0.5),
            'walk_time': time.time()
        }
        # Randomly flag for mutation
        if random.random() < 0.2:
            walk_log['flag_mutation'] = True
        send_message({
            'from': self.agent_id,
            'type': 'walk_log',
            'payload': walk_log,
            'timestamp': int(time.time())
        })

    def run(self):
        self.load_fragments()
        for path, frag in self.fragment_cache:
            self.walk_fragment(path, frag)
            time.sleep(0.1)  # Optional pacing

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

==== tool_bench.py ====
#!/usr/bin/env uv run
'''
    Simplistic tool call benchmarks for llama-server and ollama.

     Essentially runs the tests at server/examples/server/tests/unit/test_tool_call.py N times, at different
temperatures and on different backends (current llama-server, baseline llama-server and ollama),
    and plots the results of multiple runs (from same .jsonl file or multiple ones) as a success rate heatmap.

    Simple usage example:

        cmake -B build -DLLAMA_CURL=1 && cmake --build build --config Release -j -t llama-server

        export LLAMA_SERVER_BIN_PATH=$PWD/build/bin/llama-server
        export LLAMA_CACHE=${LLAMA_CACHE:-$HOME/Library/Caches/llama.cpp}

         ./scripts/tool_bench.py run --n 30 --temp -1 --temp 0 --temp 1 --model "Qwen 2.5 1.5B Q4_K_M"
--output qwen1.5b.jsonl  --hf bartowski/Qwen2.5-1.5B-Instruct-GGUF      --ollama qwen2.5:1.5b-instruct-q4_K_M
         ./scripts/tool_bench.py run --n 30 --temp -1 --temp 0 --temp 1 --model "Qwen 2.5 Coder 7B Q4_K_M"
--output qwenc7b.jsonl   --hf bartowski/Qwen2.5-Coder-7B-Instruct-GGUF  --ollama qwen2.5-coder:7b

        ./scripts/tool_bench.py plot *.jsonl                         # Opens window w/ heatmap
        ./scripts/tool_bench.py plot qwen*.jsonl  --output qwen.png  # Saves heatmap to qwen.png

    (please see ./scripts/tool_bench.sh for a more complete example)
'''
# /// script
# requires-python = ">=3.10"
# dependencies = [
#     "pytest",
#     "pandas",
#     "matplotlib",
#     "seaborn",
#     "requests",
#     "wget",
#     "typer",
# ]
# ///
from contextlib import contextmanager
from pathlib import Path
import re
from statistics import mean, median
from typing import Annotated, Dict, List, Optional, Tuple
import atexit
import json
import logging
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import subprocess
import sys
import time
import typer

sys.path.insert(0, Path(__file__).parent.parent.as_posix())
if True:
    from examples.server.tests.utils import ServerProcess
         from  examples.server.tests.unit.test_tool_call  import  TIMEOUT_SERVER_START,  do_test_calc_result,
do_test_hello_world, do_test_weather


@contextmanager
def scoped_server(sp: ServerProcess):
    def stop():
        nonlocal sp
        if sp is not None:
            sp.stop()
            sp = None # type: ignore
    atexit.register(stop)
    yield sp
    stop()


logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

app = typer.Typer()


@app.command()
def plot(files: List[Path], output: Optional[Path] = None, test_regex: Optional[str] = None, server_regex:
Optional[str] = None):
```

```python
lines: List[Dict] = []
for file in files:
    if not file.exists():
        logger.error(f"File not found: {file}")
        continue

    try:
        with file.open() as f:
            raw_data = f.read()
        logger.info(f"Reading {file} ({len(raw_data)} bytes)")

        for line_num, line in enumerate(raw_data.split('\n'), 1):
            line = line.strip()
            if not line:
                continue
            try:
                record = json.loads(line)
                lines.append(record)
            except json.JSONDecodeError as e:
                logger.warning(f"Invalid JSON at {file}:{line_num} - {e}")
    except Exception as e:
        logger.error(f"Error processing {file}: {e}")

if not lines:
    raise Exception("No valid data was loaded")

data_dict: Dict[Tuple, float] = {}
models: List[str] = []
temps = set()
tests = set()
server_names = set()
total_counts = set()
for rec in lines:
    try:
        model = rec["model"]
        temp = rec["temp"]
        server_name = rec["server_name"]
        test = rec["test"]
        success = rec["success_ratio"]
        success_count = rec["success_count"]
        failure_count = rec["failure_count"]
        total_count = success_count + failure_count
        total_counts.add(total_count)

        if test_regex and not re.search(test_regex, test):
            continue

        if server_regex and not re.search(server_regex, server_name):
            continue

        data_dict[(model, temp, server_name, test)] = success

        if model not in models:
            models.append(model)
            temps.add(temp)
            tests.add(test)
            server_names.add(server_name)

        except KeyError as e:
            logger.warning(f"Missing required field in record: {e}")

    if len(total_counts) > 1:
        logger.warning(f"Total counts are not consistent: {total_counts}")

    # Sort the collected values
    temps = list(sorted(temps, key=lambda x: x if x is not None else -1))
    tests = list(sorted(tests))
    server_names = list(sorted(server_names))

    logger.info(f"Processed {len(lines)} lines")
    logger.info(f"Found {len(data_dict)} valid data points")
    logger.info(f"Models: {models}")
    logger.info(f"Temperatures: {temps}")
    logger.info(f"Tests: {tests}")
    logger.info(f"Servers: {server_names}")

    matrix: list[list[float]] = []
    index: list[str] = []

    all_cols = [
        (server_name, test)
        for server_name in server_names
        for test in tests
    ]
    for model in models:
        for temp in temps:
            index.append(f"{model} @ {temp}")
            row_vals = [
                data_dict.get((model, temp, server_name, test), np.nan)
                for server_name, test in all_cols
            ]
            matrix.append(row_vals)

    columns: list[str] = [f"{server_name}\n{test}" for server_name, test in all_cols]

    df = pd.DataFrame(matrix, index=np.array(index), columns=np.array(columns))

    plt.figure(figsize=(12, 6))

    sns.heatmap(
        df, annot=True, cmap="RdYlGn", vmin=0.0, vmax=1.0, cbar=True, fmt=".2f", center=0.5, square=True,
linewidths=0.5,
        cbar_kws={"label": "Success Ratio"},
    )

    plt.title(f"Tool Call Bench (n = {str(min(total_counts)) if len(total_counts) == 1 else
f'{min(total_counts)}-{max(total_counts)}'})\nSuccess Ratios by Server & Test", pad=20)
    plt.xlabel("Server & Test", labelpad=10)
```

```python
    plt.ylabel("Model @ Temperature", labelpad=10)

    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)

    plt.tight_layout()

    if output:
        plt.savefig(output, dpi=300, bbox_inches='tight')
        logger.info(f"Plot saved to {output}")
    else:
        plt.show()


@app.command()
def run(
    output: Annotated[Path, typer.Option(help="Output JSON file")],
    model: Annotated[Optional[str], typer.Option(help="Name of the model to test (server agnostic)")] = None,
    hf: Annotated[Optional[str], typer.Option(help="GGUF huggingface model repo id (+ optional quant) to test w/ llama-server")] = None,
    chat_template: Annotated[Optional[str], typer.Option(help="Chat template override for llama-server")] = None,
    ollama: Annotated[Optional[str], typer.Option(help="Ollama model tag to test")] = None,
    llama_baseline: Annotated[Optional[str], typer.Option(help="llama-server baseline binary path to use as baseline")] = None,
    n: Annotated[int, typer.Option(help="Number of times to run each test")] = 10,
    temp: Annotated[Optional[List[float]], typer.Option(help="Set of temperatures to test")] = None,
    top_p: Annotated[Optional[float], typer.Option(help="top_p")] = None,
    top_k: Annotated[Optional[int], typer.Option(help="top_k")] = None,
    ctk: Annotated[Optional[str], typer.Option(help="ctk")] = None,
    ctv: Annotated[Optional[str], typer.Option(help="ctv")] = None,
    fa: Annotated[Optional[bool], typer.Option(help="fa")] = None,
    seed: Annotated[Optional[int], typer.Option(help="Random seed")] = None,
    port: Annotated[int, typer.Option(help="llama-server port")] = 8084,
    force: Annotated[bool, typer.Option(help="Force overwrite of output file")] = False,
    append: Annotated[bool, typer.Option(help="Append to output file")] = False,

    test_hello_world: Annotated[bool, typer.Option(help="Whether to run the hello world test")] = True,
    test_weather: Annotated[bool, typer.Option(help="Whether to run the weather test")] = True,
    test_calc_result: Annotated[bool, typer.Option(help="Whether to run the calc result test")] = False,
):
    # Check only one of output and append

    n_predict = 512 # High because of DeepSeek R1
    # n_ctx = 8192
    n_ctx = 2048

    assert force or append or not output.exists(), f"Output file already exists: {output}; use --force to overwrite"

    with output.open('a' if append else 'w') as output_file:

        def run(server: ServerProcess, *, server_name: str, model_id: str, temp: Optional[float] = None, output_kwargs={}, request_kwargs={}):
```

```
request_kwargs = {**request_kwargs}
if temp is not None:
    request_kwargs['temperature'] = temp
if top_p is not None:
    request_kwargs['top_p'] = top_p
if top_k is not None:
    request_kwargs['top_k'] = top_k
if seed is not None:
    request_kwargs['seed'] = seed

request_kwargs['cache_prompt'] = False

tests = {}
if test_hello_world:
    tests["hello world"] = lambda server: do_test_hello_world(server, **request_kwargs)
if test_weather:
    tests["weather"] = lambda server: do_test_weather(server, **request_kwargs)
if test_calc_result:
    tests["calc result"] = lambda server: do_test_calc_result(server, None, 512, **request_kwargs)

for test_name, test in tests.items():
    success_count = 0
    failure_count = 0
    failures = []
    success_times = []
    failure_times = []
    logger.info(f"Running {test_name} ({server_name}, {model}): ")
    for i in range(n):
        start_time = time.time()

        def elapsed():
            return time.time() - start_time

        try:
            test(server)
            success_times.append(elapsed())
            success_count += 1
            logger.info('success')
        except Exception as e:
            logger.error(f'failure: {e}')
            failure_count += 1
            failure_times.append(elapsed())
            failures.append(str(e))
            # import traceback
            # traceback.print_exc()
    output_file.write(json.dumps({**output_kwargs, **dict(
        model=model,
        server_name=server_name,
        model_id=model_id,
        test=test_name,
        temp=t,
        top_p=top_p,
        top_k=top_k,
        ctk=ctk,
                ctv=ctv,
                seed=seed,
                success_ratio=float(success_count) / n,
                avg_time=mean(success_times + failure_times),
                median_time=median(success_times + failure_times),
                success_count=success_count,
                success_times=success_times,
                failure_count=failure_count,
                failure_times=failure_times,
                failures=list(set(failures)),
            )}) + '\n')
            output_file.flush()

    for t in [None] if temp is None else [t if t >= 0 else None for t in temp]:
        if hf is not None:

            servers: list[Tuple[str, Optional[str]]] = [('llama-server', None)]
            if llama_baseline is not None:
                servers.append(('llama-server (baseline)', llama_baseline))

            for server_name, server_path in servers:
                server = ServerProcess()
                server.n_ctx = n_ctx
                server.n_slots = 1
                server.jinja = True
                server.ctk = ctk
                server.ctv = ctv
                server.fa = fa
                server.n_predict = n_predict
                server.model_hf_repo = hf
                server.model_hf_file = None
                server.chat_template = chat_template
                server.server_path = server_path
                if port is not None:
                    server.server_port = port
                # server.debug = True

                with scoped_server(server):
                    server.start(timeout_seconds=TIMEOUT_SERVER_START)
                    for ignore_chat_grammar in [False]:
                        run(
                            server,
                            server_name=server_name,
                            model_id=hf,
                            temp=t,
                            output_kwargs=dict(
                                chat_template=chat_template,
                            ),
                            request_kwargs=dict(
                                ignore_chat_grammar=ignore_chat_grammar,
                            ),
                        )

        if ollama is not None:
                server = ServerProcess()
                server.server_port = 11434
                server.server_host = "localhost"
                subprocess.check_call(["ollama", "pull", ollama])

                with scoped_server(server):
                    run(
                        server,
                        server_name="ollama",
                        model_id=ollama,
                        temp=t,
                        output_kwargs=dict(
                            chat_template=None,
                        ),
                        request_kwargs=dict(
                            model=ollama,
                            max_tokens=n_predict,
                            num_ctx = n_ctx,
                        ),
                    )


if __name__ == "__main__":
    app()

==== total_devourer.py ====
"""
LOGICSHREDDER :: total_devourer.py
Purpose: Consume .txt, .json, .yaml, .py from logic_input/, convert to symbolic logic, store in fragments/core
"""

import os, uuid, yaml, json, re, shutil
from pathlib import Path
import time

INPUT_DIR = Path("logic_input")
CONSUMED_DIR = INPUT_DIR / "devoured"
FRAG_DIR = Path("fragments/core")
DISPATCH_DIR = Path("fragments/incoming")

INPUT_DIR.mkdir(exist_ok=True)
CONSUMED_DIR.mkdir(exist_ok=True)
FRAG_DIR.mkdir(parents=True, exist_ok=True)
DISPATCH_DIR.mkdir(parents=True, exist_ok=True)

def is_valid_sentence(line):
    if not line or len(line) < 10: return False
    if line.count(" ") < 2: return False
    if re.match(r'^[\d\W_]+$', line): return False
    return True

def sanitize(line):
    return line.strip().strip("\"',.;:").replace("?", "").replace("?", "")
```

```
def extract_claims_txt(f):
    return [sanitize(l) for l in open(f, 'r', encoding='utf-8') if is_valid_sentence(sanitize(l))]

def extract_claims_json(f):
    try:
        data = json.load(open(f, 'r', encoding='utf-8'))
        if isinstance(data, list):
            return [sanitize(item) for item in data if isinstance(item, str) and is_valid_sentence(item)]
        if isinstance(data, dict):
            return [sanitize(v) for k, v in data.items() if isinstance(v, str) and is_valid_sentence(v)]
    except: pass
    return []

def extract_claims_yaml(f):
    try:
        data = yaml.safe_load(open(f, 'r', encoding='utf-8'))
        if isinstance(data, list):
            return [sanitize(item) for item in data if isinstance(item, str) and is_valid_sentence(item)]
        if isinstance(data, dict):
            return [sanitize(v) for k, v in data.items() if isinstance(v, str) and is_valid_sentence(v)]
    except: pass
    return []

def extract_claims_py(f):
    lines = []
    for line in open(f, 'r', encoding='utf-8'):
        if is_valid_sentence(line) and any(k in line for k in ["def ", "return", "==", "if "]):
            lines.append(sanitize(line))
    return lines

def write_fragment(claim, origin):
    frag = {
        "id": str(uuid.uuid4())[:8],
        "claim": claim,
        "confidence": 0.8,
        "emotion": {},
        "timestamp": int(time.time()),
        "source": origin
    }
    core_path = FRAG_DIR / f"{frag['id']}.yaml"
    dist_path = DISPATCH_DIR / f"{frag['id']}.yaml"
    for p in [core_path, dist_path]:
        with open(p, 'w', encoding='utf-8') as f:
            yaml.safe_dump(frag, f)

def devour():
    files = list(INPUT_DIR.glob("*.*"))
    total = 0
    for f in files:
        claims = []
        ext = f.suffix.lower()
        if ext == ".txt":
            claims = extract_claims_txt(f)
        elif ext == ".json":
```