```
    return f"{minutes}m ago"

def get_agent_usage():
    stats = []
    for file in AGENT_STATS.glob("*.stat"):
        try:
            with open(file, 'r', encoding='utf-8') as f:
                lines = f.readlines()
                if not lines:
                    continue
                last = lines[-1].strip().split(",")
                stats.append({
                    "name": file.stem,
                    "cpu": float(last[1]),
                    "mem": float(last[2]),
                    "read": float(last[3]),
                    "write": float(last[4])
                })
        except:
            continue
    return stats

def draw_static(stdscr):
    stdscr.clear()
    stdscr.border()
    stdscr.addstr(1, 2, "LOGICSHREDDER: REAL-TIME DASH", curses.A_BOLD)
    stdscr.addstr(3, 4, "AGENTS STATUS", curses.A_UNDERLINE)
    stdscr.addstr(3, 35, "RESOURCE USAGE", curses.A_UNDERLINE)
    stdscr.addstr(10, 4, "? HOT BELIEFS (TOP 5)", curses.A_UNDERLINE)
    stdscr.addstr(18, 4, "[Q] Quit  |  [L] Lock Brain  |  [U] Unlock Brain", curses.A_DIM)
def draw_dynamic(stdscr):
    lock_status = "LOCKED" if get_lock_status() else "UNLOCKED"
    snap_time = get_last_snapshot_time()
    agents = get_agent_usage()
    beliefs = get_hot_beliefs()

    for i, agent in enumerate(agents[:6]):
        stdscr.addstr(4 + i, 4, f"{agent['name']:<16} [OK]")
        stdscr.addstr(4 + i, 35, f"CPU: {agent['cpu']}% | MEM: {agent['mem']} MB")

    stdscr.addstr(4, 65, f"I/O:")
    stdscr.addstr(5, 65, f"Read: {sum(a['read'] for a in agents):.1f} MB")
    stdscr.addstr(6, 65, f"Write: {sum(a['write'] for a in agents):.1f} MB")
    stdscr.addstr(7, 65, f"Last snapshot: {snap_time}")
    stdscr.addstr(8, 65, f"Lock status: {lock_status}")

    for i, (fid, data) in enumerate(beliefs):
        claim = fid.replace("frag_", "")[:16]
        score = data.get("heat_score", 0.0)
        stdscr.addstr(11 + i, 6, f"- {claim:<18} (heat: {score:.2f})")

def toggle_lock(lock=True):
    if lock:
        Path("core/neuro.lock").write_text(str(int(time.time())))
```

```
    else:
        Path("core/neuro.lock").unlink(missing_ok=True)

def main(stdscr):
    curses.curs_set(0)
    stdscr.nodelay(True)
    while True:
        draw_static(stdscr)
        draw_dynamic(stdscr)
        stdscr.refresh()
        for _ in range(REFRESH_INTERVAL * 10):
            key = stdscr.getch()
            if key == ord("q"):
                return
            elif key == ord("l"):
                toggle_lock(True)
            elif key == ord("u"):
                toggle_lock(False)
            time.sleep(0.1)

if __name__ == "__main__":
    curses.wrapper(main)

==== logic_ram_scheduler.py ====
# logic_ram_scheduler.py
import os
import yaml
import psutil
import time
import threading
import subprocess
from pathlib import Path
from shutil import copyfile

BASE = Path(__file__).parent
CONFIG_PATH = BASE / "system_config.yaml"
ADAPTIVE_INSTALLER = BASE / "adaptive_installer.py"
FRAG_ROOT = BASE / "fragments" / "core"

def ensure_config_exists():
    if not CONFIG_PATH.exists():
        print("INFO system_config.yaml not found. Running adaptive_installer.py...")
        result = subprocess.run(["python", str(ADAPTIVE_INSTALLER)], capture_output=True, text=True)
        if result.returncode != 0:
            print("ERROR Failed to run adaptive_installer.py:")
            print(result.stderr)
            exit(1)
        print("[OK] system_config.yaml generated.")

def load_config():
    with open(CONFIG_PATH, "r") as f:
        return yaml.safe_load(f)

def get_ram_shards(config):
```

```
    return config.get("logic_ram", {})

def list_fragments(source):
    return list(Path(source).glob("*.yaml"))

def schedule_fragments_to_cache(fragments, shard_paths, per_shard=20):
    shards = list(shard_paths.values())
    if not shards:
        print("ERROR No logic shards found in config.")
        return

    total = len(fragments)
    assigned = 0

    for i, frag in enumerate(fragments):
        target_shard = Path(shards[i % len(shards)])
        dest = target_shard / frag.name
        try:
            copyfile(frag, dest)
            assigned += 1
        except Exception as e:
            print(f"[scheduler] Failed to assign {frag.name}: {e}")

    print(f"[OK] Assigned {assigned}/{total} fragments to {len(shards)} shard(s).")

def preload_scheduler():
    ensure_config_exists()
    config = load_config()
    shards = get_ram_shards(config)

    if not shards:
        print("WARNING No logic RAM shards defined.")
        return

    print("? Scanning logic fragments...")
    fragments = list_fragments(FRAG_ROOT)
    if not fragments:
        print("WARNING No fragments found in core/")
        return

    schedule_fragments_to_cache(fragments, shards)

def monitor_and_reload(interval=60):
    while True:
        preload_scheduler()
        time.sleep(interval)

if __name__ == "__main__":
    thread = threading.Thread(target=monitor_and_reload, daemon=True)
    thread.start()
    print("INFO logic_ram_scheduler running in background. CTRL+C to kill.")
    while True:
        time.sleep(9999)
```

```
==== logic_scraper_dispatch.py ====
"""
LOGICSHREDDER :: logic_scraper_dispatch.py
Purpose: Auto-detects file types in /llm_output/, routes to correct scraper, feeds fragments to core
"""

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

SRC_DIR = Path("llm_output")
CONSUMED_DIR = SRC_DIR / "devoured"
FRAG_DIR = Path("fragments/core")

SRC_DIR.mkdir(exist_ok=True)
CONSUMED_DIR.mkdir(exist_ok=True)
FRAG_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_txt(path):
    return [sanitize(l) for l in open(path, 'r', encoding='utf-8') if is_valid_sentence(sanitize(l))]

def extract_json(path):
    try:
        data = json.load(open(path, '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_yaml(path):
    try:
        data = yaml.safe_load(open(path, '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_py(path):
    lines = []
    for line in open(path, '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, source):
    frag = {
        "id": str(uuid.uuid4())[:8],
        "claim": claim,
        "confidence": 0.85,
        "emotion": {},
        "timestamp": int(time.time()),
        "source": source
    }
    path = FRAG_DIR / f"{frag['id']}.yaml"
    with open(path, 'w', encoding='utf-8') as f:
        yaml.safe_dump(frag, f)

def dispatch():
    files = list(SRC_DIR.glob("*.*"))
    total = 0
    for f in files:
        claims = []
        ext = f.suffix.lower()
        if ext == ".txt":
            claims = extract_txt(f)
        elif ext == ".json":
            claims = extract_json(f)
        elif ext == ".yaml":
            claims = extract_yaml(f)
        elif ext == ".py":
            claims = extract_py(f)
        elif ext in [".gguf", ".bin", ".safetensors"]:
            print(f"[dispatcher] WARNING Skipped binary: {f.name}")
            continue

        if claims:
            for c in claims:
                write_fragment(c, f.name)
            print(f"[dispatcher] [OK] Routed {len(claims)} from {f.name}")
            total += len(claims)
            shutil.move(f, CONSUMED_DIR / f.name)
        else:
            print(f"[dispatcher] WARNING No usable logic in {f.name}")

    print(f"[dispatcher] ? Total symbolic fragments created: {total}")

if __name__ == "__main__":
    dispatch()

==== lstm_glove.py ====
"""Naive LSTM model."""
import keras.layers as L
import keras.backend as K
from keras.models import Model
import numpy as np
```

```
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS
import os

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, training=True, **kwargs):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  var_flat = L.Lambda(lambda x: K.reshape(x, K.stack([-1, K.prod(K.shape(x)[1:])])), name='var_flat')
  flat_ctx = var_flat(context)

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

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

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

  EMBEDDING_DIM = 100

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

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

  # Read query
```

```
  _, *states = L.LSTM(dim, return_state=True, name='query_lstm')(embedded_q)
  # Read context
  out, *states = L.LSTM(dim, return_state=True, name='ctx_lstm')(embedded_ctx, initial_state=states)

  # Prediction
  out = L.concatenate([out]+states, name='final_states')
  out = L.Dense(1, activation='sigmoid', name='out')(out)

  model = Model([context, query], out)
  if training:
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  return model

==== mac_glove.py ====
"""Iterative memory attention model."""
import numpy as np
import keras.backend as K
import keras.layers as L
from keras.models import Model
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS
import os

from .zerogru import ZeroGRU

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

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

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

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

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

EMBEDDING_DIM = 100

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

# Onehot embeddeding of symbols
# onehot_weights = np.eye(char_size)
# onehot_weights[0, 0] = 0 # Clear zero index
# onehot = L.Embedding(char_size, char_size,
#                      trainable=False,
#                      weights=[onehot_weights],
#                      name='onehot')
embedding_layer = L.Embedding(len(word_index) + 1,
                              EMBEDDING_DIM,
                              weights=[embedding_matrix],
                              trainable=False)
embedded_ctx = embedding_layer(flat_ctx) # (?, rules, preds*chars*char_size)
embedded_q = embedding_layer(query) # (?, chars, char_size)

# Embed predicates
embed_pred = ZeroGRU(dim, go_backwards=True, return_sequences=True, return_state=True, name='embed_pred')
embedded_predqs, embedded_predq = embed_pred(embedded_q) # (?, chars, dim)
embed_pred.return_sequences = False
embed_pred.return_state = False
# Embed every rule
embedded_rules = L.TimeDistributed(embed_pred, name='rule_embed')(embedded_ctx)
# (?, rules, dim)

# Reused layers over iterations
concatm1 = L.Concatenate(name='concatm1')
repeat_toqlen = L.RepeatVector(K.shape(embedded_q)[1], name='repeat_toqlen')
mult_cqi = L.Multiply(name='mult_cqi')
dense_cqi = L.Dense(dim, name='dense_cqi')
dense_cais = L.Dense(1, name='dense_cais')

squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
softmax1 = L.Softmax(axis=1, name='softmax1')
dot11 = L.Dot((1, 1), name='dot11')

repeat_toctx = L.RepeatVector(K.shape(context)[1], name='repeat_toctx')
memory_dense = L.Dense(dim, name='memory_dense')
kb_dense = L.Dense(dim, name='kb_dense')
mult_info = L.Multiply(name='mult_info')
info_dense = L.Dense(dim, name='info_dense')
mult_att_dense = L.Multiply(name='mult_att_dense')
read_att_dense = L.Dense(1, name='read_att_dense')

mem_info_dense = L.Dense(dim, name='mem_info_dense')
```

```
stack1 = L.Lambda(lambda xs: K.stack(xs, 1), output_shape=(None, dim), name='stack1')
mult_self_att = L.Multiply(name='mult_self_att')
self_att_dense = L.Dense(1, name='self_att_dense')
misa_dense = L.Dense(dim, use_bias=False, name='misa_dense')
mi_info_dense = L.Dense(dim, name='mi_info_dense')
add_mip = L.Lambda(lambda xy: xy[0]+xy[1], name='add_mip')
control_gate = L.Dense(1, activation='sigmoid', name='control_gate')
gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

# Init control and memory
zeros_like = L.Lambda(K.zeros_like, name='zeros_like')
memory = embedded_predq # (?, dim)
control = zeros_like(memory) # (?, dim)
pmemories, pcontrols = [memory], [control]

# Reasoning iterations
outs = list()
for i in range(iterations):
  # Control Unit
  qi = L.Dense(dim, name='qi'+str(i))(embedded_predq) # (?, dim)
  cqi = dense_cqi(concatm1([control, qi])) # (?, dim)
  cais = dense_cais(mult_cqi([repeat_toqlen(cqi), embedded_predqs])) # (?, qlen, 1)
  cais = squeeze2(cais) # (?, qlen)
  cais = softmax1(cais) # (?, qlen)
  outs.append(cais)
  new_control = dot11([cais, embedded_predqs]) # (?, dim)

  # Read Unit
  info = mult_info([repeat_toctx(memory_dense(memory)), kb_dense(embedded_rules)]) # (?, rules, dim)
  infop = info_dense(concatm1([info, embedded_rules])) # (?, rules, dim)
  rai = read_att_dense(mult_att_dense([repeat_toctx(new_control), infop])) # (?, rules, 1)
  rai = squeeze2(rai) # (?, rules)
  rai = softmax1(rai) # (?, rules)
  outs.append(rai)
  read = dot11([rai, embedded_rules]) # (?, dim)

  # Write Unit
  mi_info = mem_info_dense(concatm1([read, memory])) # (?, dim)
  past_ctrls = stack1(pcontrols) # (?, i+1, dim)
  sai = self_att_dense(mult_self_att([L.RepeatVector(i+1)(new_control), past_ctrls])) # (?, i+1, 1)
  sai = squeeze2(sai) # (?, i+1)
  sai = softmax1(sai) # (?, i+1)
  outs.append(sai)
  past_mems = stack1(pmemories) # (?, i+1, dim)
  misa = L.dot([sai, past_mems], (1, 1), name='misa_'+str(i)) # (?, dim)
  mip = add_mip([misa_dense(misa), mi_info_dense(mi_info)]) # (?, dim)
  cip = control_gate(new_control) # (?, 1)
  outs.append(cip)
  new_memory = gate2([mip, memory, cip]) # (?, dim)

  # Update state
  pcontrols.append(new_control)
  pmemories.append(new_memory)
  memory, control = new_memory, new_control
```

```
  # Output Unit
  out = L.Dense(1, activation='sigmoid', name='out')(concatm1([embedded_predq, memory]))
  if training:
    model = Model([context, query], out)
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== main.py ====
import argparse
import sys

import torch
import torch.nn.functional as F
from sklearn.model_selection import train_test_split

from crm.core import Network
from crm.utils import (  # get_explanations,; train_distributed,
    get_best_config,
    get_max_explanations,
    get_metrics,
    get_predictions,
    load_object,
    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(
        "-s",
        "--saved-model",
        type=str,
        help="location of saved model",
        required=False,
        default=None,
    )
    parser.add_argument(
        "-n", "--num-epochs", type=int, help="number of epochs", required=True
    )
    parser.add_argument(
        "-p", "--predict", help="get predictions for a test set", action="store_true"
    )
    parser.add_argument(
```

```
        "-e", "--explain", help="get explanations for predictions", action="store_true"
    )
    parser.add_argument(
        "-t", "--tune", help="tune the hyper parameters", 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)
    torch.set_num_threads(16)
    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)

    # Load data
    file_name = args.file
    print("***Loading data***")
    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
    )

    # Create CRM structure and train with input data
    print("***Creating CRM structure***")
    n = Network(len(adj_list), adj_list)
    n.to(device)

    if args.saved_model:
```

```
    print("***Loading Saved Model***")
    n = load_object(args.saved_model)

criterion = F.cross_entropy
optimizer = torch.optim.Adam(n.parameters(), lr=0.001)
if args.tune:
    print("***Get Best Config***")
    best = get_best_config(
        n, X_train, y_train, args.num_epochs, optimizer, criterion
    )
    print(best)

print("***Training CRM***")
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=24, stratify=y_train
)

# train_distributed(
#     n,
#     X_train,
#     y_train,
#     args.num_epochs,
#     optimizer,
#     criterion,
#     X_val,
#     y_val,
#     num_workers=16,
# )

train_losses, train_accs, val_losses, val_accs = train(
    n,
    X_train,
    y_train,
    args.num_epochs,
    torch.optim.Adam(n.parameters(), lr=best["lr"] if args.tune else 0.001),
    criterion,
    X_val=X_val,
    y_val=y_val,
    save_here=args.output + "_model",
    verbose=args.verbose,
)

# Train metrics
if not args.saved_model:
    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("-----------------------------------")
```

```
    # Predict for the test instances
    if args.predict:
        print("***Predicting the class labels for the test set***")
        for X_test, y_test in test_dataset:
            get_predictions(n, X_test, y_test)

    # Explain the test instances
    if args.explain:
        print("***Generating explanations for the test set***")
        for X_test, y_test in test_dataset:
            # get_explanations(
            #    n,
            #    X_test,
            #    y_test,
            #    true_explanations=true_explanations,
            #    k=1,
            #    verbose=args.verbose
            # )

            # added by T: get max explanations
            get_max_explanations(
                n,
                X_test,
                y_test,
                true_explanations=true_explanations,
                k=1,
                verbose=args.verbose,
            )
            print("-------------------------------------")


if __name__ == "__main__":
    """
    import cProfile
    import pstats

    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats("cumtime")
    stats.print_stats()
    """
    main()

==== memory_archiver.py ====
import yaml, time, shutil
from pathlib import Path

INDEX = Path("meta/memory_index.yaml")
CORE = Path("fragments/core")
ARCHIVE = Path("fragments/archive")
AGE_LIMIT = 86400 * 3  # 3 days
```

```
def load_index():
    with open(INDEX, 'r') as f:
        return yaml.safe_load(f)

def archive_logic():
    now = int(time.time())
    index = load_index()
    changes = 0

    for fid, meta in index.items():
        last_seen = meta.get('last_seen', now)
        if now - last_seen > AGE_LIMIT and meta.get('status') == 'active':
            fpath = CORE / f"{fid}.yaml"
            if fpath.exists():
                shutil.move(str(fpath), ARCHIVE / f"{fid}.yaml")
                meta['status'] = 'archived'
                changes += 1

    with open(INDEX, 'w') as f:
        yaml.dump(index, f)
    print(f"[memory_archiver] Archived: {changes}")

if __name__ == "__main__":
    archive_logic()

==== memory_tracker.py ====
"""
LOGICSHREDDER :: memory_tracker.py
Purpose: Track which logic fragments are accessed, reused, or aged
"""

import yaml
import time
from pathlib import Path

FRAG_DIR = Path("fragments/core")
MEMORY_INDEX = Path("logs/memory_index.yaml")
MEMORY_INDEX.parent.mkdir(parents=True, exist_ok=True)

# Load or create memory index
def load_memory():
    if MEMORY_INDEX.exists():
        with open(MEMORY_INDEX, 'r', encoding='utf-8') as f:
            return yaml.safe_load(f) or {}
    return {}

def save_memory(index):
    with open(MEMORY_INDEX, 'w', encoding='utf-8') as f:
        yaml.safe_dump(index, f)

def touch_fragment(frag_id):
    memory = load_memory()
    now = int(time.time())
```

```
    if frag_id not in memory:
        memory[frag_id] = {
            "recall_count": 1,
            "last_used": now,
            "first_seen": now,
            "frozen": False,
            "archive_candidate": False
        }
    else:
        memory[frag_id]["recall_count"] += 1
        memory[frag_id]["last_used"] = now

    save_memory(memory)

def log_bulk_fragments(fragment_list):
    memory = load_memory()
    now = int(time.time())
    updated = 0

    for frag in fragment_list:
        frag_id = frag.get("id")
        if not frag_id:
            continue

        if frag_id not in memory:
            memory[frag_id] = {
                "recall_count": 1,
                "last_used": now,
                "first_seen": now,
                "frozen": False,
                "archive_candidate": False
            }
        else:
            memory[frag_id]["recall_count"] += 1
            memory[frag_id]["last_used"] = now

        updated += 1

    save_memory(memory)
    return updated

def forget_old(threshold_days=60):
    memory = load_memory()
    now = int(time.time())
    cutoff = now - (threshold_days * 86400)
    purged = 0

    for frag_id, meta in memory.items():
        if not meta.get("frozen") and meta.get("last_used", 0) < cutoff:
            meta["archive_candidate"] = True
            purged += 1

    save_memory(memory)
    return purged
```

```
if __name__ == "__main__":
    print("INFO Tracking current memory usage...")
    updated = forget_old()
    print(f"? Marked {updated} fragments as archive candidates.")

==== memory_visualizer.py ====
import yaml
from pathlib import Path
from datetime import datetime

INDEX = Path("meta/memory_index.yaml")
OUT = Path("meta/memory_visual_report.txt")

def load_index():
    with open(INDEX, 'r') as f:
        return yaml.safe_load(f)

def make_heatbar(conf):
    filled = int(conf * 20)
    return "?" * filled + "-" * (20 - filled)

def dump_visual():
    data = load_index()
    lines = [f"INFO MEMORY VISUALIZER REPORT ? {datetime.now().isoformat()}"]
    for fid, meta in sorted(data.items()):
        conf = meta.get('confidence', 0.5)
        last = meta.get('last_seen', '?')
           lines.append(f"{fid}: {make_heatbar(conf)} | Confidence: {conf:.2f} | Last Seen: {last} | Status:
{meta.get('status', 'unknown')}")
    with open(OUT, 'w') as f:
        f.write("\n".join(lines))
    print(f"[memory_visualizer] Output -> {OUT}")

if __name__ == "__main__":
    dump_visual()

==== mesh_rebuilder.py ====
# mesh_rebuilder.py
# CONFIG Auto-crawls current directory tree to rebuild logic mesh configuration
import os
import yaml
from pathlib import Path

BASE = Path(__file__).parent.resolve()
CONFIG_PATH = BASE / "system_config.yaml"
MOUNT_MAP_PATH = BASE / "mount_map.yaml"
BRAINMAP_PATH = BASE / "brainmap.yaml"

logic_roles = {
    "core": ["token_agent.py", "mutation_engine.py", "dreamwalker.py"],
    "cold": ["cold_logic_mover.py", "archive/", "cold_storage/"],
    "incoming": ["guffifier_v2.py", "belief_ingestor.py"],
    "emotion": ["emotion_daemon.py", "emotion_bank.nosql"],
```

```
    "subcon": ["subcon_agent.py", "dream_state_loop.py"],
    "fusion": ["fusion_engine.py", "validator.py"],
    "meta": ["meta_agent.py", "feedback_daemon.py"]
}

def find_role(path):
    for role, keywords in logic_roles.items():
        for keyword in keywords:
            if keyword in str(path):
                return role
    return "unassigned"

def crawl_and_map():
    logic_mounts = {}
    brainmap = {}
    for root, dirs, files in os.walk(BASE):
        for f in files:
            path = Path(root) / f
            role = find_role(path)
            if role not in logic_mounts:
                logic_mounts[role] = []
            logic_mounts[role].append(str(path.relative_to(BASE)))
            brainmap[str(path.relative_to(BASE))] = role
    return logic_mounts, brainmap

def write_yaml(data, out_path):
    with open(out_path, "w", encoding="utf-8") as f:
        yaml.safe_dump(data, f, sort_keys=False)

def rebuild_config():
    mounts, brainmap = crawl_and_map()

    system_config = {
        "base_path": str(BASE),
        "logic_zones": list(mounts.keys()),
        "fragment_format": "yaml",
        "storage_mode": "hybrid"
    }

    print(f"? Scanned base: {BASE}")
    print(f"INFO Zones found: {list(mounts.keys())}")
    print(f"? Writing: {CONFIG_PATH}, {MOUNT_MAP_PATH}, {BRAINMAP_PATH}")

    write_yaml(system_config, CONFIG_PATH)
    write_yaml(mounts, MOUNT_MAP_PATH)
    write_yaml(brainmap, BRAINMAP_PATH)

    print("[OK] Mesh rebuild complete.")

if __name__ == "__main__":
    rebuild_config()

==== meta_agent.py ====
from core.config_loader import get
```

```
meta_agent.py"""
LOGICSHREDDER :: meta_agent.py
Purpose: Monitor belief activity, curiosity score, mutation depth, and logic heatmap
"""

import yaml
import time
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 collections import defaultdict
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")
LOG_PATH = Path("logs/meta_agent.log")
ACTIVITY_TRACKER = Path("logs/walk_activity.log")
MUTATION_LOG = Path("logs/mutation_log.txt")

CURIOUS_THRESHOLD = 30  # Seconds since last walk
HOT_THRESHOLD = 5       # High walk count = recent hotness

def load_walk_activity():
    activity = {}
    if ACTIVITY_TRACKER.exists():
        with open(ACTIVITY_TRACKER, 'r', encoding='utf-8') as file:
            for line in file:
                try:
                    timestamp, frag_id = line.strip().split(',')
                    activity[frag_id] = int(timestamp)
                except:
                    continue
    return activity

def load_mutation_counts():
    mutations = defaultdict(int)
    if MUTATION_LOG.exists():
        with open(MUTATION_LOG, 'r', encoding='utf-8') as file:
            for line in file:
                if "Mutation" in line and "from" in line:
                    parts = line.split()
                    new_id = parts[2]
                    parent_id = parts[-1]
                    mutations[parent_id] += 1
    return mutations

def evaluate_fragment(frag_id, last_walk_time, mutation_count):
    now = int(time.time())
    seconds_since_walk = now - last_walk_time
    curiosity_score = min(1.0, seconds_since_walk / 60.0)  # max out at 1.0
    mutation_penalty = min(0.5, mutation_count * 0.05)
    score = curiosity_score - mutation_penalty
    return max(0.0, round(score, 3))
```

```
def log_meta(frag_id, score):
    with open(LOG_PATH, 'a', encoding='utf-8') as log:
        log.write(f"[{int(time.time())}] {frag_id}: curiosity={score}\n")

def analyze_fragments():
    activity = load_walk_activity()
    mutations = load_mutation_counts()
    ranked = []

    for path in FRAG_DIR.glob("*.yaml"):
        try:
            frag = yaml.safe_load(path.read_text(encoding='utf-8'))
            frag_id = frag.get('id', path.stem)
            last_walk = activity.get(frag_id, 0)
            mut_count = mutations.get(frag_id, 0)
            score = evaluate_fragment(frag_id, last_walk, mut_count)
            log_meta(frag_id, score)
            if score >= 0.7:
                ranked.append((score, frag_id))
        except Exception as e:
            print(f"[meta_agent] Error analyzing {path.name}: {e}")

    # Emit top curious fragments
    top = sorted(ranked, reverse=True)[:5]
    for score, fid in top:
        send_message({
            'from': 'meta_agent',
            'type': 'curiosity_alert',
            'payload': {'frag_id': fid, 'curiosity': score},
            'timestamp': int(time.time())
        })
        print(f"[meta_agent] CURIOUS: {fid} -> {score}")

if __name__ == "__main__":
    while True:
        analyze_fragments()
        time.sleep(30)  # Every 30s, reassess curiosity
# [CONFIG_PATCHED]

==== metadata.py ====
from __future__ import annotations

import re
import json
import yaml
import logging
from pathlib import Path
from typing import Any, Literal, Optional
from dataclasses import dataclass

from .constants import Keys

import gguf
```

```
logger = logging.getLogger("metadata")


@dataclass
class Metadata:
    # Authorship Metadata to be written to GGUF KV Store
    name: Optional[str] = None
    author: Optional[str] = None
    version: Optional[str] = None
    organization: Optional[str] = None
    finetune: Optional[str] = None
    basename: Optional[str] = None
    description: Optional[str] = None
    quantized_by: Optional[str] = None
    size_label: Optional[str] = None
    url: Optional[str] = None
    doi: Optional[str] = None
    uuid: Optional[str] = None
    repo_url: Optional[str] = None
    source_url: Optional[str] = None
    source_doi: Optional[str] = None
    source_uuid: Optional[str] = None
    source_repo_url: Optional[str] = None
    license: Optional[str] = None
    license_name: Optional[str] = None
    license_link: Optional[str] = None
    base_models: Optional[list[dict]] = None
    tags: Optional[list[str]] = None
    languages: Optional[list[str]] = None
    datasets: Optional[list[dict]] = None

    @staticmethod
     def load(metadata_override_path: Optional[Path] = None, model_path: Optional[Path] = None, model_name:
Optional[str] = None, total_params: int = 0) -> Metadata:
        # This grabs as many contextual authorship metadata as possible from the model repository
        # making any conversion as required to match the gguf kv store metadata format
        # as well as giving users the ability to override any authorship metadata that may be incorrect

        # Create a new Metadata instance
        metadata = Metadata()

        model_card = Metadata.load_model_card(model_path)
        hf_params = Metadata.load_hf_parameters(model_path)
        # TODO: load adapter_config.json when possible, it usually contains the base model of the LoRA adapter

        # heuristics
        metadata = Metadata.apply_metadata_heuristic(metadata, model_card, hf_params, model_path, total_params)

        # Metadata Override File Provided
        # This is based on LLM_KV_NAMES mapping in llama.cpp
        metadata_override = Metadata.load_metadata_override(metadata_override_path)

        metadata.name            = metadata_override.get(Keys.General.NAME,            metadata.name)
```

```
        metadata.author          = metadata_override.get(Keys.General.AUTHOR,          metadata.author)
        metadata.version         = metadata_override.get(Keys.General.VERSION,         metadata.version)
        metadata.organization    = metadata_override.get(Keys.General.ORGANIZATION,    metadata.organization)

        metadata.finetune        = metadata_override.get(Keys.General.FINETUNE,        metadata.finetune)
        metadata.basename        = metadata_override.get(Keys.General.BASENAME,        metadata.basename)

        metadata.description     = metadata_override.get(Keys.General.DESCRIPTION,     metadata.description)
        metadata.quantized_by    = metadata_override.get(Keys.General.QUANTIZED_BY,    metadata.quantized_by)

        metadata.size_label      = metadata_override.get(Keys.General.SIZE_LABEL,      metadata.size_label)
        metadata.license_name    = metadata_override.get(Keys.General.LICENSE_NAME,    metadata.license_name)
        metadata.license_link    = metadata_override.get(Keys.General.LICENSE_LINK,    metadata.license_link)

        metadata.url             = metadata_override.get(Keys.General.URL,             metadata.url)
        metadata.doi             = metadata_override.get(Keys.General.DOI,             metadata.doi)
        metadata.uuid            = metadata_override.get(Keys.General.UUID,            metadata.uuid)
        metadata.repo_url        = metadata_override.get(Keys.General.REPO_URL,        metadata.repo_url)

        metadata.source_url      = metadata_override.get(Keys.General.SOURCE_URL,      metadata.source_url)
        metadata.source_doi      = metadata_override.get(Keys.General.SOURCE_DOI,      metadata.source_doi)
        metadata.source_uuid     = metadata_override.get(Keys.General.SOURCE_UUID,     metadata.source_uuid)
                        metadata.source_repo_url    =    metadata_override.get(Keys.General.SOURCE_REPO_URL,
metadata.source_repo_url)

        # Base Models is received here as an array of models
        metadata.base_models     = metadata_override.get("general.base_models",        metadata.base_models)

        # Datasets is received here as an array of datasets
        metadata.datasets        = metadata_override.get("general.datasets",           metadata.datasets)

        metadata.tags            = metadata_override.get(Keys.General.TAGS,            metadata.tags)
        metadata.languages       = metadata_override.get(Keys.General.LANGUAGES,       metadata.languages)

        # Direct Metadata Override (via direct cli argument)
        if model_name is not None:
            metadata.name = model_name

        return metadata

    @staticmethod
    def load_metadata_override(metadata_override_path: Optional[Path] = None) -> dict[str, Any]:
        if metadata_override_path is None or not metadata_override_path.is_file():
            return {}

        with open(metadata_override_path, "r", encoding="utf-8") as f:
            return json.load(f)

    @staticmethod
    def load_model_card(model_path: Optional[Path] = None) -> dict[str, Any]:
        if model_path is None or not model_path.is_dir():
            return {}

        model_card_path = model_path / "README.md"
```

```
        if not model_card_path.is_file():
            return {}

        # The model card metadata is assumed to always be in YAML (frontmatter)
                                                                                                    #            ref:
https://github.com/huggingface/transformers/blob/a5c642fe7a1f25d3bdcd76991443ba6ff7ee34b2/src/transformers/mode
lcard.py#L468-L473
        yaml_content: str = ""
        with open(model_card_path, "r", encoding="utf-8") as f:
            content = f.read()
            lines = content.splitlines()
            lines_yaml = []
            if len(lines) == 0:
                # Empty file
                return {}
            if len(lines) > 0 and lines[0] != "---":
                # No frontmatter
                return {}
            for line in lines[1:]:
                if line == "---":
                    break # End of frontmatter
                else:
                    lines_yaml.append(line)
            yaml_content = "\n".join(lines_yaml) + "\n"

        # Quick hack to fix the Norway problem
        # https://hitchdev.com/strictyaml/why/implicit-typing-removed/
        yaml_content = yaml_content.replace("- no\n", "- \"no\"\n")

        if yaml_content:
            data = yaml.safe_load(yaml_content)
            if isinstance(data, dict):
                return data
            else:
                    logger.error(f"while reading YAML model card frontmatter, data is {type(data)} instead of
dict")
                return {}
        else:
            return {}

    @staticmethod
    def load_hf_parameters(model_path: Optional[Path] = None) -> dict[str, Any]:
        if model_path is None or not model_path.is_dir():
            return {}

        config_path = model_path / "config.json"

        if not config_path.is_file():
            return {}

        with open(config_path, "r", encoding="utf-8") as f:
            return json.load(f)
```

```
    @staticmethod
    def id_to_title(string):
        # Convert capitalization into title form unless acronym or version number
        return ' '.join([w.title() if w.islower() and not re.match(r'^(v\d+(?:\.\d+)*|\d.*)$', w) else w for w
in string.strip().replace('-', ' ').split()])

    @staticmethod
    def get_model_id_components(model_id: Optional[str] = None, total_params: int = 0) -> tuple[str | None, str
| None, str | None, str | None, str | None, str | None]:
        # Huggingface often store model id as '<org>/<model name>'
        # so let's parse it and apply some heuristics if possible for model name components

        if model_id is None:
            # model ID missing
            return None, None, None, None, None, None

        if ' ' in model_id:
            # model ID is actually a normal human sentence
            # which means its most likely a normal model name only
            # not part of the hugging face naming standard, but whatever
            return model_id, None, None, None, None, None

        if '/' in model_id:
            # model ID (huggingface style)
            org_component, model_full_name_component = model_id.split('/', 1)
        else:
            # model ID but missing org components
            org_component, model_full_name_component = None, model_id

        # Check if we erroneously matched against './' or '../' etc...
        if org_component is not None and len(org_component) > 0 and org_component[0] == '.':
            org_component = None

        name_parts: list[str] = model_full_name_component.split('-')

        # Remove empty parts
        for i in reversed(range(len(name_parts))):
            if len(name_parts[i]) == 0:
                del name_parts[i]

        name_types: list[
            set[Literal["basename", "size_label", "finetune", "version", "type"]]
        ] = [set() for _ in name_parts]

        # Annotate the name
        for i, part in enumerate(name_parts):
            # Version
            if re.fullmatch(r'(v|iter)?\d+([.]\d+)*', part, re.IGNORECASE):
                name_types[i].add("version")
            # Quant type (should not be there for base models, but still annotated)
            elif re.fullmatch(r'i?q\d(_\w)*|b?fp?(16|32)', part, re.IGNORECASE):
                name_types[i].add("type")
                name_parts[i] = part.upper()
            # Model size
```

```
                elif i > 0 and re.fullmatch(r'(([A]|\d+[x])?\d+([._]\d+)?[KMBT][\d]?|small|mini|medium|large|x?xl)', part, re.IGNORECASE):
                    part = part.replace("_", ".")
                    # Handle weird bloom-7b1 notation
                    if part[-1].isdecimal():
                        part = part[:-2] + "." + part[-1] + part[-2]
                    # Normalize the size suffixes
                    if len(part) > 1 and part[-2].isdecimal():
                        if part[-1] in "kmbt":
                            part = part[:-1] + part[-1].upper()
                    if total_params != 0:
                        try:
                            label_params = float(part[:-1]) * pow(1000, " KMBT".find(part[-1]))
                            # Only use it as a size label if it's close or bigger than the model size
                            # Note that LoRA adapters don't necessarily include all layers,
                            # so this is why bigger label sizes are accepted.
                            # Do not use the size label when it's smaller than 1/8 of the model size
                            if (total_params < 0 and label_params < abs(total_params) // 8) or (
                                # Check both directions when the current model isn't a LoRA adapter
                                total_params > 0 and abs(label_params - total_params) > 7 * total_params // 8
                            ):
                                # Likely a context length
                                name_types[i].add("finetune")
                                # Lowercase the size when it's a context length
                                part = part[:-1] + part[-1].lower()
                        except ValueError:
                            # Failed to convert the size label to float, use it anyway
                            pass
                    if len(name_types[i]) == 0:
                        name_types[i].add("size_label")
                    name_parts[i] = part
                # Some easy to recognize finetune names
                elif i > 0 and re.fullmatch(r'chat|instruct|vision|lora', part, re.IGNORECASE):
                    if total_params < 0 and part.lower() == "lora":
                        # ignore redundant "lora" in the finetune part when the output is a lora adapter
                        name_types[i].add("type")
                    else:
                        name_types[i].add("finetune")

            # Ignore word-based size labels when there is at least a number-based one present
            # TODO: should word-based size labels always be removed instead?
            if any(c.isdecimal() for n, t in zip(name_parts, name_types) if "size_label" in t for c in n):
                for n, t in zip(name_parts, name_types):
                    if "size_label" in t:
                        if all(c.isalpha() for c in n):
                            t.remove("size_label")

            at_start = True
            # Find the basename through the annotated name
            for part, t in zip(name_parts, name_types):
                if at_start and ((len(t) == 0 and part[0].isalpha()) or "version" in t):
                    t.add("basename")
                else:
                    if at_start:
```

```
                    at_start = False
                if len(t) == 0:
                    t.add("finetune")

        # Remove the basename annotation from trailing version
        for part, t in zip(reversed(name_parts), reversed(name_types)):
            if "basename" in t and len(t) > 1:
                t.remove("basename")
            else:
                break

        basename = "-".join(n for n, t in zip(name_parts, name_types) if "basename" in t) or None
        # Deduplicate size labels using order-preserving 'dict' ('set' seems to sort the keys)
           size_label = "-".join(dict.fromkeys(s for s, t in zip(name_parts, name_types) if "size_label" in
t).keys()) or None
        finetune = "-".join(f for f, t in zip(name_parts, name_types) if "finetune" in t) or None
        # TODO: should the basename version always be excluded?
        # NOTE: multiple finetune versions are joined together
         version = "-".join(v for v, t, in zip(name_parts, name_types) if "version" in t and "basename" not in
t) or None

        if size_label is None and finetune is None and version is None:
            # Too ambiguous, output nothing
            basename = None

        return model_full_name_component, org_component, basename, finetune, version, size_label

    @staticmethod
         def apply_metadata_heuristic(metadata: Metadata, model_card: Optional[dict] = None, hf_params:
Optional[dict] = None, model_path: Optional[Path] = None, total_params: int = 0) -> Metadata:
        # Reference Model Card Metadata: https://github.com/huggingface/hub-docs/blob/main/modelcard.md?plain=1

        # Model Card Heuristics
        ########################
        if model_card is not None:

            def use_model_card_metadata(metadata_key: str, model_card_key: str):
                if model_card_key in model_card and getattr(metadata, metadata_key, None) is None:
                    setattr(metadata, metadata_key, model_card.get(model_card_key))

            def use_array_model_card_metadata(metadata_key: str, model_card_key: str):
                # Note: Will append rather than replace if already exist
                tags_value = model_card.get(model_card_key, None)
                if tags_value is None:
                    return

                current_value = getattr(metadata, metadata_key, None)
                if current_value is None:
                    current_value = []

                if isinstance(tags_value, str):
                    current_value.append(tags_value)
                elif isinstance(tags_value, list):
                    current_value.extend(tags_value)
```

```
                setattr(metadata, metadata_key, current_value)

            # LLAMA.cpp's direct internal convention
            # (Definitely not part of hugging face formal/informal standard)
            #########################################
            use_model_card_metadata("name", "name")
            use_model_card_metadata("author", "author")
            use_model_card_metadata("version", "version")
            use_model_card_metadata("organization", "organization")
            use_model_card_metadata("description", "description")
            use_model_card_metadata("finetune", "finetune")
            use_model_card_metadata("basename", "basename")
            use_model_card_metadata("size_label", "size_label")
            use_model_card_metadata("source_url", "url")
            use_model_card_metadata("source_doi", "doi")
            use_model_card_metadata("source_uuid", "uuid")
            use_model_card_metadata("source_repo_url", "repo_url")

            # LLAMA.cpp's huggingface style convention
              # (Definitely not part of hugging face formal/informal standard... but with model_ appended to
match their style)
            ###########################################
            use_model_card_metadata("name", "model_name")
            use_model_card_metadata("author", "model_author")
            use_model_card_metadata("version", "model_version")
            use_model_card_metadata("organization", "model_organization")
            use_model_card_metadata("description", "model_description")
            use_model_card_metadata("finetune", "model_finetune")
            use_model_card_metadata("basename", "model_basename")
            use_model_card_metadata("size_label", "model_size_label")
            use_model_card_metadata("source_url", "model_url")
            use_model_card_metadata("source_doi", "model_doi")
            use_model_card_metadata("source_uuid", "model_uuid")
            use_model_card_metadata("source_repo_url", "model_repo_url")

            # Hugging Face Direct Convention
            ################################

            # Not part of huggingface model card standard but notice some model creator using it
            # such as TheBloke in 'TheBloke/Mistral-7B-Instruct-v0.2-GGUF'
            use_model_card_metadata("name", "model_name")
            use_model_card_metadata("author", "model_creator")
            use_model_card_metadata("basename", "model_type")

            if "base_model" in model_card or "base_models" in model_card or "base_model_sources" in model_card:
                # This represents the parent models that this is based on
                # Example: stabilityai/stable-diffusion-xl-base-1.0. Can also be a list (for merges)
                                                                        #     Example     of     merges:
https://huggingface.co/EmbeddedLLM/Mistral-7B-Merge-14-v0.1/blob/main/README.md
                metadata_base_models = []
                                base_model_value  =  model_card.get("base_model",  model_card.get("base_models",
model_card.get("base_model_sources", None)))
```

```
                if base_model_value is not None:
                    if isinstance(base_model_value, str):
                        metadata_base_models.append(base_model_value)
                    elif isinstance(base_model_value, list):
                        metadata_base_models.extend(base_model_value)

                if metadata.base_models is None:
                    metadata.base_models = []

                for model_id in metadata_base_models:
                    # NOTE: model size of base model is assumed to be similar to the size of the current model
                    base_model = {}
                    if isinstance(model_id, str):
                                    if  model_id.startswith("http://")  or  model_id.startswith("https://")  or
model_id.startswith("ssh://"):
                            base_model["repo_url"] = model_id

                            # Check if Hugging Face ID is present in URL
                            if "huggingface.co" in model_id:
                                match = re.match(r"https?://huggingface.co/([^/]+/[^/]+)$", model_id)
                                if match:
                                    model_id_component = match.group(1)
                                        model_full_name_component, org_component, basename, finetune, version,
size_label = Metadata.get_model_id_components(model_id_component, total_params)

                                    # Populate model dictionary with extracted components
                                    if model_full_name_component is not None:
                                        base_model["name"] = Metadata.id_to_title(model_full_name_component)
                                    if org_component is not None:
                                        base_model["organization"] = Metadata.id_to_title(org_component)
                                    if version is not None:
                                        base_model["version"] = version

                        else:
                            # Likely a Hugging Face ID
                            model_full_name_component, org_component, basename, finetune, version, size_label =
Metadata.get_model_id_components(model_id, total_params)

                            # Populate model dictionary with extracted components
                            if model_full_name_component is not None:
                                base_model["name"] = Metadata.id_to_title(model_full_name_component)
                            if org_component is not None:
                                base_model["organization"] = Metadata.id_to_title(org_component)
                            if version is not None:
                                base_model["version"] = version
                            if org_component is not None and model_full_name_component is not None:
                                                                                    base_model["repo_url"]   =
f"https://huggingface.co/{org_component}/{model_full_name_component}"

                    elif isinstance(model_id, dict):
                        base_model = model_id

                    else:
                        logger.error(f"base model entry '{str(model_id)}' not in a known format")
```

```
                    metadata.base_models.append(base_model)

            if "datasets" in model_card or "dataset" in model_card or "dataset_sources" in model_card:
                # This represents the datasets that this was trained from
                metadata_datasets = []
                                        dataset_value  =  model_card.get("datasets",  model_card.get("dataset",
model_card.get("dataset_sources", None)))

                if dataset_value is not None:
                    if isinstance(dataset_value, str):
                        metadata_datasets.append(dataset_value)
                    elif isinstance(dataset_value, list):
                        metadata_datasets.extend(dataset_value)

                if metadata.datasets is None:
                    metadata.datasets = []

                for dataset_id in metadata_datasets:
                    # NOTE: model size of base model is assumed to be similar to the size of the current model
                    dataset = {}
                    if isinstance(dataset_id, str):
                        if dataset_id.startswith(("http://", "https://", "ssh://")):
                            dataset["repo_url"] = dataset_id

                            # Check if Hugging Face ID is present in URL
                            if "huggingface.co" in dataset_id:
                                match = re.match(r"https?://huggingface.co/([^/]+/[^/]+)$", dataset_id)
                                if match:
                                    dataset_id_component = match.group(1)
                                           dataset_name_component, org_component, basename, finetune, version,
size_label = Metadata.get_model_id_components(dataset_id_component, total_params)

                                    # Populate dataset dictionary with extracted components
                                    if dataset_name_component is not None:
                                        dataset["name"] = Metadata.id_to_title(dataset_name_component)
                                    if org_component is not None:
                                        dataset["organization"] = Metadata.id_to_title(org_component)
                                    if version is not None:
                                        dataset["version"] = version

                        else:
                            # Likely a Hugging Face ID
                               dataset_name_component, org_component, basename, finetune, version, size_label =
Metadata.get_model_id_components(dataset_id, total_params)

                            # Populate dataset dictionary with extracted components
                            if dataset_name_component is not None:
                                dataset["name"] = Metadata.id_to_title(dataset_name_component)
                            if org_component is not None:
                                dataset["organization"] = Metadata.id_to_title(org_component)
                            if version is not None:
                                dataset["version"] = version
                            if org_component is not None and dataset_name_component is not None:
```

```
                                                                    dataset["repo_url"]  =
f"https://huggingface.co/{org_component}/{dataset_name_component}"

                    elif isinstance(dataset_id, dict):
                        dataset = dataset_id

                    else:
                        logger.error(f"dataset entry '{str(dataset_id)}' not in a known format")

                    metadata.datasets.append(dataset)

            use_model_card_metadata("license", "license")
            use_model_card_metadata("license_name", "license_name")
            use_model_card_metadata("license_link", "license_link")

            use_array_model_card_metadata("tags", "tags")
            use_array_model_card_metadata("tags", "pipeline_tag")

            use_array_model_card_metadata("languages", "languages")
            use_array_model_card_metadata("languages", "language")

        # Hugging Face Parameter Heuristics
        #####################################

        if hf_params is not None:

            hf_name_or_path = hf_params.get("_name_or_path")
            if hf_name_or_path is not None and hf_name_or_path.count('/') <= 1:
                # Use _name_or_path only if its actually a model name and not some computer path
                # e.g. 'meta-llama/Llama-2-7b-hf'
                model_id = hf_name_or_path
                        model_full_name_component, org_component, basename, finetune, version, size_label =
Metadata.get_model_id_components(model_id, total_params)
                if metadata.name is None and model_full_name_component is not None:
                    metadata.name = Metadata.id_to_title(model_full_name_component)
                if metadata.organization is None and org_component is not None:
                    metadata.organization = Metadata.id_to_title(org_component)
                if metadata.basename is None and basename is not None:
                    metadata.basename = basename
                if metadata.finetune is None and finetune is not None:
                    metadata.finetune = finetune
                if metadata.version is None and version is not None:
                    metadata.version = version
                if metadata.size_label is None and size_label is not None:
                    metadata.size_label = size_label

        # Directory Folder Name Fallback Heuristics
        ############################################
        if model_path is not None:
            model_id = model_path.name
                       model_full_name_component, org_component, basename, finetune, version, size_label =
Metadata.get_model_id_components(model_id, total_params)
            if metadata.name is None and model_full_name_component is not None:
                metadata.name = Metadata.id_to_title(model_full_name_component)
```

```
        if metadata.organization is None and org_component is not None:
            metadata.organization = Metadata.id_to_title(org_component)
        if metadata.basename is None and basename is not None:
            metadata.basename = basename
        if metadata.finetune is None and finetune is not None:
            metadata.finetune = finetune
        if metadata.version is None and version is not None:
            metadata.version = version
        if metadata.size_label is None and size_label is not None:
            metadata.size_label = size_label

    return metadata

def set_gguf_meta_model(self, gguf_writer: gguf.GGUFWriter):
    assert self.name is not None
    gguf_writer.add_name(self.name)

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

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

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

    if self.size_label is not None:
        gguf_writer.add_size_label(self.size_label)

    if self.license is not None:
        if isinstance(self.license, list):
            gguf_writer.add_license(",".join(self.license))
        else:
            gguf_writer.add_license(self.license)
    if self.license_name is not None:
        gguf_writer.add_license_name(self.license_name)
    if self.license_link is not None:
        gguf_writer.add_license_link(self.license_link)

    if self.url is not None:
        gguf_writer.add_url(self.url)
    if self.doi is not None:
        gguf_writer.add_doi(self.doi)
    if self.uuid is not None:
        gguf_writer.add_uuid(self.uuid)
    if self.repo_url is not None:
```

```
    gguf_writer.add_repo_url(self.repo_url)

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

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

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

```
        if self.tags is not None:
            gguf_writer.add_tags(self.tags)
        if self.languages is not None:
            gguf_writer.add_languages(self.languages)

==== minicpmv-convert-image-encoder-to-gguf.py ====
# coding=utf-8
# Copyright 2024 Google AI and The HuggingFace Team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" PyTorch Siglip model. """
# Copied from  HuggingFaceM4/siglip-so400m-14-980-flash-attn2-navit and add tgt_sizes


import os
import math
import warnings

import numpy as np
import torch
import torch.nn.functional as F
import torch.utils.checkpoint
from torch import nn
from torch.nn.init import _calculate_fan_in_and_fan_out

from transformers.activations import ACT2FN
from transformers.modeling_utils import PreTrainedModel
from transformers.configuration_utils import PretrainedConfig
from transformers.utils import (
    logging,
)
from transformers.utils import logging

logger = logging.get_logger(__name__)

class SiglipVisionConfig(PretrainedConfig):
    r"""
     This is the configuration class to store the configuration of a [`SiglipVisionModel`]. It is used to instantiate a
    Siglip vision encoder according to the specified arguments, defining the model architecture. Instantiating a
     configuration with the defaults will yield a similar configuration to that of the vision encoder of the Siglip
```

```
    [google/siglip-base-patch16-224](https://huggingface.co/google/siglip-base-patch16-224) architecture.
    Configuration objects inherit from [`PretrainedConfig`] and can be used to control the model outputs. Read the
    documentation from [`PretrainedConfig`] for more information.
    Args:
        hidden_size (`int`, *optional*, defaults to 768):
            Dimensionality of the encoder layers and the pooler layer.
        intermediate_size (`int`, *optional*, defaults to 3072):
            Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.
        num_hidden_layers (`int`, *optional*, defaults to 12):
            Number of hidden layers in the Transformer encoder.
        num_attention_heads (`int`, *optional*, defaults to 12):
            Number of attention heads for each attention layer in the Transformer encoder.
        num_channels (`int`, *optional*, defaults to 3):
            Number of channels in the input images.
        image_size (`int`, *optional*, defaults to 224):
            The size (resolution) of each image.
        patch_size (`int`, *optional*, defaults to 16):
            The size (resolution) of each patch.
        hidden_act (`str` or `function`, *optional*, defaults to `"gelu_pytorch_tanh"`):
            The non-linear activation function (function or string) in the encoder and pooler. If string, `"gelu"`,
            `"relu"`, `"selu"` and `"gelu_new"` ``"quick_gelu"` are supported.
        layer_norm_eps (`float`, *optional*, defaults to 1e-06):
            The epsilon used by the layer normalization layers.
        attention_dropout (`float`, *optional*, defaults to 0.0):
            The dropout ratio for the attention probabilities.
    Example:
    ```python
    >>> from transformers import SiglipVisionConfig, SiglipVisionModel
    >>> # Initializing a SiglipVisionConfig with google/siglip-base-patch16-224 style configuration
    >>> configuration = SiglipVisionConfig()
    >>> # Initializing a SiglipVisionModel (with random weights) from the google/siglip-base-patch16-224 style configuration
    >>> model = SiglipVisionModel(configuration)
    >>> # Accessing the model configuration
    >>> configuration = model.config
    ```"""

    model_type = "siglip_vision_model"

    def __init__(
        self,
        hidden_size=768,
        intermediate_size=3072,
        num_hidden_layers=12,
        num_attention_heads=12,
        num_channels=3,
        image_size=224,
        patch_size=16,
        hidden_act="gelu_pytorch_tanh",
        layer_norm_eps=1e-6,
        attention_dropout=0.0,
        **kwargs,
```

```
    ):
        super().__init__(**kwargs)

        self.hidden_size = hidden_size
        self.intermediate_size = intermediate_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.num_channels = num_channels
        self.patch_size = patch_size
        self.image_size = image_size
        self.attention_dropout = attention_dropout
        self.layer_norm_eps = layer_norm_eps
        self.hidden_act = hidden_act

_CHECKPOINT_FOR_DOC = "google/siglip-base-patch16-224"

SIGLIP_PRETRAINED_MODEL_ARCHIVE_LIST = [
    "google/siglip-base-patch16-224",
    # See all SigLIP models at https://huggingface.co/models?filter=siglip
]

# Copied from transformers.models.llama.modeling_llama._get_unpad_data
def _get_unpad_data(attention_mask):
    seqlens_in_batch = attention_mask.sum(dim=-1, dtype=torch.int32)
    indices = torch.nonzero(attention_mask.flatten(), as_tuple=False).flatten()
    max_seqlen_in_batch = seqlens_in_batch.max().item()
    cu_seqlens = F.pad(torch.cumsum(seqlens_in_batch, dim=0, dtype=torch.int32), (1, 0))
    return (
        indices,
        cu_seqlens,
        max_seqlen_in_batch,
    )


def _trunc_normal_(tensor, mean, std, a, b):
    # Cut & paste from PyTorch official master until it's in a few official releases - RW
    # Method based on https://people.sc.fsu.edu/~jburkardt/presentations/truncated_normal.pdf
    def norm_cdf(x):
        # Computes standard normal cumulative distribution function
        return (1.0 + math.erf(x / math.sqrt(2.0))) / 2.0

    if (mean < a - 2 * std) or (mean > b + 2 * std):
        warnings.warn(
            "mean is more than 2 std from [a, b] in nn.init.trunc_normal_. "
            "The distribution of values may be incorrect.",
            stacklevel=2,
        )

    # Values are generated by using a truncated uniform distribution and
    # then using the inverse CDF for the normal distribution.
    # Get upper and lower cdf values
    l = norm_cdf((a - mean) / std)
    u = norm_cdf((b - mean) / std)
```

```
    # Uniformly fill tensor with values from [l, u], then translate to
    # [2l-1, 2u-1].
    tensor.uniform_(2 * l - 1, 2 * u - 1)

    # Use inverse cdf transform for normal distribution to get truncated
    # standard normal
    if tensor.dtype in [torch.float16, torch.bfloat16]:
        # The `erfinv_` op is not (yet?) defined in float16+cpu, bfloat16+gpu
        og_dtype = tensor.dtype
        tensor = tensor.to(torch.float32)
        tensor.erfinv_()
        tensor = tensor.to(og_dtype)
    else:
        tensor.erfinv_()

    # Transform to proper mean, std
    tensor.mul_(std * math.sqrt(2.0))
    tensor.add_(mean)

    # Clamp to ensure it's in the proper range
    if tensor.dtype == torch.float16:
        # The `clamp_` op is not (yet?) defined in float16+cpu
        tensor = tensor.to(torch.float32)
        tensor.clamp_(min=a, max=b)
        tensor = tensor.to(torch.float16)
    else:
        tensor.clamp_(min=a, max=b)


def trunc_normal_tf_(
    tensor: torch.Tensor, mean: float = 0.0, std: float = 1.0, a: float = -2.0, b: float = 2.0
):
    """Fills the input Tensor with values drawn from a truncated
    normal distribution. The values are effectively drawn from the
    normal distribution :math:`\\mathcal{N}(\text{mean}, \text{std}^2)`
    with values outside :math:`[a, b]` redrawn until they are within
    the bounds. The method used for generating the random values works
    best when :math:`a \\leq \text{mean} \\leq b`.
    NOTE: this 'tf' variant behaves closer to Tensorflow / JAX impl where the
    bounds [a, b] are applied when sampling the normal distribution with mean=0, std=1.0
    and the result is subsquently scaled and shifted by the mean and std args.
    Args:
        tensor: an n-dimensional `torch.Tensor`
        mean: the mean of the normal distribution
        std: the standard deviation of the normal distribution
        a: the minimum cutoff value
        b: the maximum cutoff value
    """
    with torch.no_grad():
        _trunc_normal_(tensor, 0, 1.0, a, b)
        tensor.mul_(std).add_(mean)


def variance_scaling_(tensor, scale=1.0, mode="fan_in", distribution="normal"):
```

```
    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    denom = fan_in
    if mode == "fan_in":
        denom = fan_in
    elif mode == "fan_out":
        denom = fan_out
    elif mode == "fan_avg":
        denom = (fan_in + fan_out) / 2

    variance = scale / denom

    if distribution == "truncated_normal":
        # constant is stddev of standard normal truncated to (-2, 2)
        trunc_normal_tf_(tensor, std=math.sqrt(variance) / 0.87962566103423978)
    elif distribution == "normal":
        with torch.no_grad():
            tensor.normal_(std=math.sqrt(variance))
    elif distribution == "uniform":
        bound = math.sqrt(3 * variance)
        with torch.no_grad():
            tensor.uniform_(-bound, bound)
    else:
        raise ValueError(f"invalid distribution {distribution}")


def lecun_normal_(tensor):
    variance_scaling_(tensor, mode="fan_in", distribution="truncated_normal")


def default_flax_embed_init(tensor):
    variance_scaling_(tensor, mode="fan_in", distribution="normal")

class SiglipVisionEmbeddings(nn.Module):
    def __init__(self, config: SiglipVisionConfig):
        super().__init__()
        self.config = config
        self.embed_dim = config.hidden_size
        self.image_size = config.image_size
        self.patch_size = config.patch_size

        self.patch_embedding = nn.Conv2d(
            in_channels=config.num_channels,
            out_channels=self.embed_dim,
            kernel_size=self.patch_size,
            stride=self.patch_size,
            padding="valid",
        )

        self.num_patches_per_side = self.image_size // self.patch_size
        self.num_patches = self.num_patches_per_side**2
        self.num_positions = self.num_patches
        self.position_embedding = nn.Embedding(self.num_positions, self.embed_dim)

class SiglipAttention(nn.Module):
```

```
    """Multi-headed attention from 'Attention Is All You Need' paper"""

    # Copied from transformers.models.clip.modeling_clip.CLIPAttention.__init__
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.embed_dim = config.hidden_size
        self.num_heads = config.num_attention_heads
        self.head_dim = self.embed_dim // self.num_heads
        if self.head_dim * self.num_heads != self.embed_dim:
            raise ValueError(
                f"embed_dim must be divisible by num_heads (got `embed_dim`: {self.embed_dim} and `num_heads`:"
                f" {self.num_heads})."
            )
        self.scale = self.head_dim**-0.5
        self.dropout = config.attention_dropout

        self.k_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.v_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.q_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.out_proj = nn.Linear(self.embed_dim, self.embed_dim)

# Copied from transformers.models.clip.modeling_clip.CLIPMLP with CLIP->Siglip
class SiglipMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.activation_fn = ACT2FN[config.hidden_act]
        self.fc1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.fc2 = nn.Linear(config.intermediate_size, config.hidden_size)


# Copied from transformers.models.clip.modeling_clip.CLIPEncoderLayer with CLIP->Siglip
class SiglipEncoderLayer(nn.Module):
    def __init__(self, config: SiglipVisionConfig):
        super().__init__()
        self.embed_dim = config.hidden_size
        self._use_flash_attention_2 = config._attn_implementation == "flash_attention_2"
        self.self_attn = (
            SiglipAttention(config)
        )
        self.layer_norm1 = nn.LayerNorm(self.embed_dim, eps=config.layer_norm_eps)
        self.mlp = SiglipMLP(config)
        self.layer_norm2 = nn.LayerNorm(self.embed_dim, eps=config.layer_norm_eps)

class SiglipPreTrainedModel(PreTrainedModel):
    """
     An abstract class to handle weights initialization and a simple interface for downloading and loading
pretrained
    models.
    """

    config_class = SiglipVisionConfig
    base_model_prefix = "siglip"
```

```
    supports_gradient_checkpointing = True

    def _init_weights(self, module):
        """Initialize the weights"""

        if isinstance(module, SiglipVisionEmbeddings):
            width = self.config.hidden_size
            nn.init.normal_(module.position_embedding.weight, std=1 / np.sqrt(width))
        elif isinstance(module, nn.Embedding):
            default_flax_embed_init(module.weight)
        elif isinstance(module, SiglipAttention):
            nn.init.normal_(module.q_proj.weight)
            nn.init.normal_(module.k_proj.weight)
            nn.init.normal_(module.v_proj.weight)
            nn.init.normal_(module.out_proj.weight)
            nn.init.zeros_(module.q_proj.bias)
            nn.init.zeros_(module.k_proj.bias)
            nn.init.zeros_(module.v_proj.bias)
            nn.init.zeros_(module.out_proj.bias)
        elif isinstance(module, SiglipMLP):
            nn.init.normal_(module.fc1.weight)
            nn.init.normal_(module.fc2.weight)
            nn.init.normal_(module.fc1.bias, std=1e-6)
            nn.init.normal_(module.fc2.bias, std=1e-6)
        elif isinstance(module, (nn.Linear, nn.Conv2d)):
            lecun_normal_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)


SIGLIP_START_DOCSTRING = r"""
     This model inherits from [`PreTrainedModel`]. Check the superclass documentation for the generic methods
the
    library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning
heads
    etc.)
      This model is also a PyTorch [torch.nn.Module](https://pytorch.org/docs/stable/nn.html#torch.nn.Module)
subclass.
    Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general
usage
    and behavior.
    Parameters:
        config ([`SiglipVisionConfig`]): Model configuration class with all the parameters of the model.
            Initializing with a config file does not load the weights associated with the model, only the
            configuration. Check out the [`~PreTrainedModel.from_pretrained`] method to load the model weights.
"""


SIGLIP_VISION_INPUTS_DOCSTRING = r"""
    Args:
        pixel_values (`torch.FloatTensor` of shape `(batch_size, num_channels, height, width)`):
```

```
                Pixel values. Padding will be ignored by default should you provide it. Pixel values can be obtained using
            [`AutoImageProcessor`]. See [`CLIPImageProcessor.__call__`] for details.
        output_attentions (`bool`, *optional*):
              Whether or not to return the attentions tensors of all attention layers. See `attentions` under returned
            tensors for more detail.
        output_hidden_states (`bool`, *optional*):
               Whether or not to return the hidden states of all layers. See `hidden_states` under returned tensors for
            more detail.
        return_dict (`bool`, *optional*):
            Whether or not to return a [`~utils.ModelOutput`] instead of a plain tuple.
"""


# Copied from transformers.models.clip.modeling_clip.CLIPEncoder with CLIP->Siglip
class SiglipEncoder(nn.Module):
    """
    Transformer encoder consisting of `config.num_hidden_layers` self attention layers. Each layer is a
    [`SiglipEncoderLayer`].
    Args:
        config: SiglipConfig
    """

    def __init__(self, config: SiglipVisionConfig):
        super().__init__()
        self.config = config
        self.layers = nn.ModuleList([SiglipEncoderLayer(config) for _ in range(config.num_hidden_layers)])
        self.gradient_checkpointing = False

class SiglipVisionTransformer(SiglipPreTrainedModel):
    config_class = SiglipVisionConfig
    main_input_name = "pixel_values"
    _supports_flash_attn_2 = True

    def __init__(self, config: SiglipVisionConfig):
        super().__init__(config)
        self.config = config
        embed_dim = config.hidden_size

        self.embeddings = SiglipVisionEmbeddings(config)
        self.encoder = SiglipEncoder(config)
        self.post_layernorm = nn.LayerNorm(embed_dim, eps=config.layer_norm_eps)
        self._use_flash_attention_2 = config._attn_implementation == "flash_attention_2"

        # Initialize weights and apply final processing
        self.post_init()

    def get_input_embeddings(self) -> nn.Module:
        return self.embeddings.patch_embedding

import argparse
import json
```

```
import re

import numpy as np
from gguf import *
from transformers.models.idefics2.modeling_idefics2 import Idefics2VisionTransformer, Idefics2VisionConfig

TEXT = "clip.text"
VISION = "clip.vision"


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


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

    if has_minicpmv and name in ["visual_projection.weight"]:
        return True

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

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

    return False


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

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


def bytes_to_unicode():
    """
    Returns list of utf-8 byte and a corresponding list of unicode strings.
    The reversible bpe codes work on unicode strings.
    This means you need a large # of unicode characters in your vocab if you want to avoid UNKs.
```

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


ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model-dir", help="Path to model directory cloned from HF Hub", required=True)
ap.add_argument("--use-f32", action="store_true", default=False, help="Use f32 instead of f16")
ap.add_argument("--text-only", action="store_true", required=False,
                help="Save a text-only model. It can't be used to encode images")
ap.add_argument("--vision-only", action="store_true", required=False,
                help="Save a vision-only model. It can't be used to encode texts")
ap.add_argument("--clip-model-is-vision", action="store_true", required=False,
                help="The clip model is a pure vision model (ShareGPT4V vision extract for example)")
ap.add_argument("--clip-model-is-openclip", action="store_true", required=False,
                help="The clip model is from openclip (for ViT-SO400M type))")
ap.add_argument("--minicpmv-projector", help="Path to minicpmv.projector file. If specified, save an image encoder for MiniCPM-V models.")
ap.add_argument("--projector-type", help="Type of projector. Possible values: mlp, ldp, ldpv2", choices=["mlp", "ldp", "ldpv2"], default="mlp")
ap.add_argument("-o", "--output-dir", help="Directory to save GGUF files. Default is the original model directory", default=None)
# Example --image_mean 0.48145466 0.4578275 0.40821073 --image_std 0.26862954 0.26130258 0.27577711
# Example --image_mean 0.5 0.5 0.5 --image_std 0.5 0.5 0.5
default_image_mean = [0.48145466, 0.4578275, 0.40821073]
default_image_std = [0.26862954, 0.26130258, 0.27577711]
ap.add_argument('--image-mean', type=float, nargs='+', help='Mean of the images for normalization (overrides processor) ', default=None)
ap.add_argument('--image-std', type=float, nargs='+', help='Standard deviation of the images for normalization (overrides processor)', default=None)
ap.add_argument('--minicpmv_version', type=int, help='minicpmv_version: MiniCPM-V-2 use 1; MiniCPM-V-2.5 use 2; MiniCPM-V-2.6 use 3; MiniCPM-o-2.6 use 4', default=2)

# with proper
args = ap.parse_args()


if args.text_only and args.vision_only:
```

```
    print("--text-only and --image-only arguments cannot be specified at the same time.")
    exit(1)

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

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

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

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

ftype = 1
if args.use_f32:
    ftype = 0

# if args.clip_model_is_vision or args.clip_model_is_openclip:
#     model = CLIPVisionModel.from_pretrained(dir_model)
#     processor = None
# else:
#     model = CLIPModel.from_pretrained(dir_model)
#     processor = CLIPProcessor.from_pretrained(dir_model)

minicpmv_version = args.minicpmv_version
emb_dim = 4096
block_count = 26
if minicpmv_version == 1:
    emb_dim = 2304
    block_count = 26
elif minicpmv_version == 2:
    emb_dim = 4096
    block_count = 27
elif minicpmv_version == 3:
    emb_dim = 3584
    block_count = 27
elif minicpmv_version == 4:
    emb_dim = 3584
    block_count = 27

default_vision_config = {
        "hidden_size": 1152,
```

```
        "image_size": 980,
        "intermediate_size": 4304,
        "model_type": "idefics2",
        "num_attention_heads": 16,
        "num_hidden_layers": 27,
        "patch_size": 14,
    }

vision_config = Idefics2VisionConfig(**default_vision_config)
model = Idefics2VisionTransformer(vision_config)
if minicpmv_version == 3:
    vision_config = SiglipVisionConfig(**default_vision_config)
    model = SiglipVisionTransformer(vision_config)
elif minicpmv_version == 4:
    vision_config = SiglipVisionConfig(**default_vision_config)
    model = SiglipVisionTransformer(vision_config)

processor = None
# if model.attn_pool is not None:
#     model.attn_pool = torch.nn.Identity()

# model.blocks = model.blocks[:-1]
model.load_state_dict(torch.load(os.path.join(dir_model, "minicpmv.clip")))

fname_middle = None
has_text_encoder = True
has_vision_encoder = True
has_minicpmv_projector = False

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

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

fout.add_bool("clip.has_text_encoder", has_text_encoder)
fout.add_bool("clip.has_vision_encoder", has_vision_encoder)
fout.add_bool("clip.has_minicpmv_projector", has_minicpmv_projector)
fout.add_file_type(ftype)
if args.text_only:
    fout.add_description("text-only CLIP model")
```

```
elif args.vision_only and not has_minicpmv_projector:
    fout.add_description("vision-only CLIP model")
elif has_minicpmv_projector:
    fout.add_description("image encoder for MiniCPM-V")
    # add projector type
    fout.add_string("clip.projector_type", "resampler")
    fout.add_int32("clip.minicpmv_version", minicpmv_version)
else:
    fout.add_description("two-tower CLIP model")

if has_vision_encoder:
    # vision_model hparams
    fout.add_uint32("clip.vision.image_size", 448)
    fout.add_uint32("clip.vision.patch_size", 14)
    fout.add_uint32(add_key_str(KEY_EMBEDDING_LENGTH, VISION), 1152)
    fout.add_uint32(add_key_str(KEY_FEED_FORWARD_LENGTH, VISION), 4304)
    fout.add_uint32("clip.vision.projection_dim", 0)
    fout.add_uint32(add_key_str(KEY_ATTENTION_HEAD_COUNT, VISION), 16)
    fout.add_float32(add_key_str(KEY_ATTENTION_LAYERNORM_EPS, VISION), 1e-6)
    fout.add_uint32(add_key_str(KEY_BLOCK_COUNT, VISION), block_count)

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

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

def get_1d_sincos_pos_embed_from_grid(embed_dim, pos):
    """
    embed_dim: output dimension for each position
    pos: a list of positions to be encoded: size (M,)
    out: (M, D)
    """
    assert embed_dim % 2 == 0
    omega = np.arange(embed_dim // 2, dtype=np.float32)
    omega /= embed_dim / 2.
    omega = 1. / 10000 ** omega  # (D/2,)

    pos = pos.reshape(-1)  # (M,)
    out = np.einsum('m,d->md', pos, omega)  # (M, D/2), outer product

    emb_sin = np.sin(out)  # (M, D/2)
    emb_cos = np.cos(out)  # (M, D/2)

    emb = np.concatenate([emb_sin, emb_cos], axis=1)  # (M, D)
    return emb
```

```
def get_2d_sincos_pos_embed_from_grid(embed_dim, grid):
    assert embed_dim % 2 == 0

    # use half of dimensions to encode grid_h
    emb_h = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[0])  # (H*W, D/2)
    emb_w = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[1])  # (H*W, D/2)

    emb = np.concatenate([emb_h, emb_w], axis=1)  # (H*W, D)
    return emb


# https://github.com/facebookresearch/mae/blob/efb2a8062c206524e35e47d04501ed4f544c0ae8/util/pos_embed.py#L20
def get_2d_sincos_pos_embed(embed_dim, grid_size, cls_token=False):
    """
    grid_size: int of the grid height and width
    return:
    pos_embed: [grid_size*grid_size, embed_dim] or [1+grid_size*grid_size, embed_dim] (w/ or w/o cls_token)
    """
    if isinstance(grid_size, int):
        grid_h_size, grid_w_size = grid_size, grid_size
    else:
        grid_h_size, grid_w_size = grid_size[0], grid_size[1]

    grid_h = np.arange(grid_h_size, dtype=np.float32)
    grid_w = np.arange(grid_w_size, dtype=np.float32)
    grid = np.meshgrid(grid_w, grid_h)  # here w goes first
    grid = np.stack(grid, axis=0)

    grid = grid.reshape([2, 1, grid_h_size, grid_w_size])
    pos_embed = get_2d_sincos_pos_embed_from_grid(embed_dim, grid)
    if cls_token:
        pos_embed = np.concatenate([np.zeros([1, embed_dim]), pos_embed], axis=0)
    return pos_embed

def _replace_name_resampler(s, v):
    if re.match("resampler.pos_embed", s):
        return {
            s: v,
                 re.sub("pos_embed", "pos_embed_k", s): torch.from_numpy(get_2d_sincos_pos_embed(emb_dim, (70,
70))),
        }
    if re.match("resampler.proj", s):
        return {
            re.sub("proj", "pos_embed_k", s): torch.from_numpy(get_2d_sincos_pos_embed(emb_dim, (70, 70))),
            re.sub("proj", "proj.weight", s): v.transpose(-1, -2).contiguous(),
        }
    if re.match("resampler.attn.in_proj_.*", s):
        return {
            re.sub("attn.in_proj_", "attn.q.", s): v.chunk(3, dim=0)[0],
            re.sub("attn.in_proj_", "attn.k.", s): v.chunk(3, dim=0)[1],
            re.sub("attn.in_proj_", "attn.v.", s): v.chunk(3, dim=0)[2],
        }
    return {s: v}
```

```
if has_minicpmv_projector:
    projector = torch.load(args.minicpmv_projector)
    new_state_dict = {}
    for k, v in projector.items():
        kvs = _replace_name_resampler(k, v)
        for nk, nv in kvs.items():
            new_state_dict[nk] = nv
    projector = new_state_dict
    ftype_cur = 0
    for name, data in projector.items():
        name = get_tensor_name(name)
        data = data.squeeze().numpy()

        n_dims = len(data.shape)
        if ftype == 1:
            if name[-7:] == ".weight" and n_dims == 2:
                print("  Converting to float16")
                data = data.astype(np.float16)
                ftype_cur = 1
            else:
                print("  Converting to float32")
                data = data.astype(np.float32)
                ftype_cur = 0
        else:
            if data.dtype != np.float32:
                print("  Converting to float32")
                data = data.astype(np.float32)
                ftype_cur = 0

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

    print("Projector tensors added\n")

def _replace_name(s, v):
    s = "vision_model." + s
    if re.match("vision_model.embeddings.position_embedding", s):
        v = v.unsqueeze(0)
        return {s: v}

    return {s: v}

state_dict = model.state_dict()
new_state_dict = {}
for k, v in state_dict.items():
    kvs = _replace_name(k, v)
    for nk, nv in kvs.items():
        new_state_dict[nk] = nv
state_dict = new_state_dict
for name, data in state_dict.items():
    if should_skip_tensor(name, has_text_encoder, has_vision_encoder, has_minicpmv_projector):
        # we don't need this
        print(f"skipping parameter: {name}")
```

```
        continue

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

    n_dims = len(data.shape)

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

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


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

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

==== minicpmv-surgery.py ====
import argparse
import os
import torch
from transformers import AutoModel, AutoTokenizer

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

# find the model part that includes the the multimodal projector weights
model      =      AutoModel.from_pretrained(args.model,      trust_remote_code=True,      local_files_only=True,
torch_dtype=torch.bfloat16)
checkpoint = model.state_dict()
```

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

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

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

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

config = model.llm.config
config.auto_map = {
    "AutoConfig": "configuration_minicpm.MiniCPMConfig",
    "AutoModel": "modeling_minicpm.MiniCPMModel",
    "AutoModelForCausalLM": "modeling_minicpm.MiniCPMForCausalLM",
    "AutoModelForSeq2SeqLM": "modeling_minicpm.MiniCPMForCausalLM",
    "AutoModelForSequenceClassification": "modeling_minicpm.MiniCPMForSequenceClassification"
}
model.llm.save_pretrained(f"{args.model}/model")
tok = AutoTokenizer.from_pretrained(args.model, trust_remote_code=True)
tok.save_pretrained(f"{args.model}/model")

print("Done!")
print(f"Now you can convert {args.model} to a regular LLaMA GGUF file.")
print(f"Also, use {args.model}/minicpmv.projector to prepare a minicpmv-encoder.gguf file.")

==== mount_binder.py ====
# mount_binder.py
# ? Rebinds orphan .py modules into correct logic zones in mount_map.yaml and brainmap.yaml

import yaml
from pathlib import Path

BASE = Path(__file__).parent
MAP_PATH = BASE / "mount_map.yaml"
BRAIN_PATH = BASE / "brainmap.yaml"
PY_FILES = list(BASE.glob("*.py"))

# Heuristic keyword map
zone_keywords = {
    "core": ["run_logicshredder", "cortex_bus", "dreamwalker"],
    "incoming": ["guffifier", "belief_ingestor"],
    "fusion": ["fusion", "mutation", "validator", "abstraction"],
    "emotion": ["emotion", "fan", "feedback", "heatmap"],
    "meta": ["meta_agent", "boot_wrapper", "auto_configurator"],
    "utils": ["config_loader", "patch", "builder", "optimizer"],
    "cold": ["cold", "archive", "teleporter"],
```

```
    "subcon": ["subcon", "dream_state", "sleep", "inner"],
    "quant": ["quant", "prompt", "devourer"],
    "distributed": ["remote", "net", "swarm", "dispatch"]
}

def guess_zone(name):
    for zone, keywords in zone_keywords.items():
        for word in keywords:
            if word.lower() in name.lower():
                return zone
    return "unassigned"

def update_map(path: Path, content: dict):
    with open(path, 'w', encoding='utf-8') as f:
        yaml.safe_dump(content, f, sort_keys=False)

def main():
    print("? Scanning for unbound logic modules...")
    mount_map = {}
    brain_map = {}

    for py in PY_FILES:
        rel_path = str(py.relative_to(BASE))
        zone = guess_zone(py.name)

        if zone not in mount_map:
            mount_map[zone] = []
        mount_map[zone].append(rel_path)
        brain_map[rel_path] = zone

        print(f"[bind] {rel_path} -> {zone}")

    update_map(MAP_PATH, mount_map)
    update_map(BRAIN_PATH, brain_map)
    print(f"[OK] Updated {MAP_PATH.name} + {BRAIN_PATH.name}")

if __name__ == "__main__":
    main()

==== mutation_engine.py ====
"""
LOGICSHREDDER :: mutation_engine.py
Purpose: Apply confidence decay, mutate symbolic beliefs, log ancestry and emit changes
"""

import os
import yaml
import time
import uuid
import random
import redis
from pathlib import Path
from core.cortex_bus import send_message
from utils import agent_profiler
```

```
import threading

# Start background profiler
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()

r = redis.Redis(decode_responses=True)

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

class MutationEngine:
    def __init__(self, agent_id="mutation_engine_01"):
        self.agent_id = agent_id

    def decay_confidence(self, frag):
        current = frag.get('confidence', 0.5)
        decay = 0.01 + random.uniform(0.005, 0.02)
        return max(0.0, current - decay)

    def mutate_claim(self, claim):
        if random.random() < 0.5:
            return f"It is possible that {claim.lower()}"
        else:
            return f"Not {claim.strip()}"

    def mutate_fragment(self, path, frag):
        new_claim = self.mutate_claim(frag['claim'])
        mutated = {
            'id': str(uuid.uuid4())[:8],
            'origin': str(path),
            'claim': new_claim,
            'parent_id': frag.get('id', None),
            'confidence': self.decay_confidence(frag),
            'emotion': frag.get('emotion', {}),
            'timestamp': int(time.time())
        }
        return mutated

    def save_mutation(self, new_frag):
        new_path = FRAG_DIR / f"{new_frag['id']}.yaml"
        with open(new_path, 'w', encoding='utf-8') as out:
            yaml.safe_dump(new_frag, out)

        with open(MUTATION_LOG, 'a', encoding='utf-8') as log:
                                    log.write(f"[{new_frag['timestamp']}]   Mutation:   {new_frag['id']}   from
{new_frag.get('parent_id')}\n")

        send_message({
            'from': self.agent_id,
            'type': 'mutation_event',
            'payload': new_frag,
            'timestamp': new_frag['timestamp']
```

```
        })

        # ? SYMBO-MODE: Notify Redis
        r.publish("decay_event", new_frag['claim'])

    def run(self):
        files = list(FRAG_DIR.glob("*.yaml"))
        for path in files:
            with open(path, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag and 'claim' in frag:
                        new_frag = self.mutate_fragment(path, frag)
                        self.save_mutation(new_frag)
                        time.sleep(0.1)
                except Exception as e:
                    print(f"[{self.agent_id}] Failed to mutate {path.name}: {e}")

if __name__ == "__main__":
    MutationEngine().run()

==== network.py ====
from itertools import repeat
from typing import Callable

import torch
import torch.multiprocessing as mp
from torch.multiprocessing import Pool

from crm.core import Neuron

# from torch.multiprocessing.pool import ThreadPool



class Network:
    def __init__(self, num_neurons, adj_list, custom_activations=None):
        self.num_neurons = num_neurons
        self.adj_list = adj_list
        self.neurons = [
            Neuron(i)
            if custom_activations is None
            else Neuron(i, custom_activations[i][0], custom_activations[i][1])
            for i in range(num_neurons)
        ]
        self.num_layers = 1
        self.weights = self._set_weights()
        self.topo_order = self._topological_sort()
        self._setup_neurons()
        self._set_output_neurons()
        self._assign_layers()
        self.has_forwarded = False
        self.is_fresh = True

    def _forward_layer(self, n_id, f_mapper, queue):
```