# cl5d.py

"""

ConsciousLeaf 5D (CL5D) -- production-ready Python module.

This module implements a simplified ConsciousLeafModel used in the

Protopine CL5D package. It is intentionally faithful to the earlier

structure and semantics (depth transform, SHP, capacity, valence, gating,

permutations, prediction). The key fractal / symbolic constraints from the

project must NOT be changed; these include the numeric ranges and the

mapping logic in `map\_to\_coords\_protein` and `map\_to\_coords\_compound`.

Author: Generated for Mrinmoy Chakraborty (adapted & hardened)

Version: 1.0.0

"""

from \_\_future\_\_ import annotations

import math

from math import factorial

from typing import List, Dict, Tuple, Any, Optional

import numpy as np

try:

    # SciPy used for gamma (robust for non-integers)

    from scipy.special import gamma as \_gamma

except Exception:  # pragma: no cover - fallback if scipy not installed

    \_gamma = math.gamma  # math.gamma exists in Python 3.8+

\_\_all\_\_ = [

    "ConsciousLeafModel",

    "map\_to\_coords\_protein",

    "map\_to\_coords\_compound",

]

# Semantic note kept from project brief:

# - "0 = ♾️" (Bombard) etc. are conceptual domain constraints.

# - Do NOT alter fractal geometry mapping heuristics without project

#   owner approval. This implementation only tidies and stabilizes code.

\_DEFAULT\_HYPER = {

    "lambda\_T": 0.8,

    "lambda\_D": 0.7,

    "beta": 3.0,

    "eta": 0.9,

    "kappa": 0.45,

    "n\_shp": 12,

    "tau": 0.12,

    "k\_min": 6,

    "k\_max": 12,

    "cn\_min": 0.000123,

    "cn\_top": 0.20,

    "R": 20,

    "xi": 1.2,

    "rho": 0.8,

}

def \_clamp01(x: float) -> float:

    """Clamp value to [0, 1]."""

    return max(0.0, min(1.0, float(x)))

class ConsciousLeafModel:

    """

    ConsciousLeafModel implements the CL5D scoring and selection pipeline.

    Public methods:

      - run(domain\_valence, agents) -> (prediction, selected\_agents)

      - map\_to\_coords\_\* helpers exist at module level (keeps compatibility)

    """

    def \_\_init\_\_(self, hyperparams: Optional[Dict[str, float]] = None) -> None:

        """Initialize model with hyperparameters (default provided)."""

        self.hyper = dict(\_DEFAULT\_HYPER)

        if hyperparams:

            # Merge but keep keys that exist in default - prevents accidental

            # injection of unrelated parameters.

            for k, v in hyperparams.items():

                if k in self.hyper:

                    self.hyper[k] = v

    # ---------- Core building-block functions ----------

    def depth\_transform(self, cn: float) -> float:

        """

        Depth transform maps contact number (or concentration) `cn` to a depth

        scaling in [0, 1]. Values >= cn\_top -> depth 0 (surface); lower cn ->

        deeper mapping. Uses logarithmic scaling to produce fractal-like decay.

        """

        cn\_top = float(self.hyper["cn\_top"])

        cn\_min = float(self.hyper["cn\_min"])

        cn = max(cn\_min, min(1.0, float(cn)))

        if cn >= cn\_top:

            return 0.0

        # Avoid division by zero: cn\_min < cn\_top by design.

        num = math.log(cn\_top / cn)

        den = math.log(cn\_top / cn\_min)

        if den == 0:

            return 0.0

        return float(num / den)

    def entropy(self, vals: List[float]) -> float:

        """

        Shannon-style entropy across clipped values in [0,1].

        Slight epsilon added to avoid log(0).

        """

        eps = 1e-12

        vals\_clamped = [\_clamp01(v) for v in vals]

        s = sum(vals\_clamped) + len(vals\_clamped) \* eps

        if s <= 0:

            return 0.0

        probs = [(v + eps) / s for v in vals\_clamped]

        ent = 0.0

        for p in probs:

            if p > 0:

                ent -= p \* math.log(p)

        return float(ent)

    def shp(self, T: float, D: float) -> float:

        """

        SHP (shape / fractal sum) aggregator.

        Sum over n\_shp terms of 1/(alpha + k \* delta) where

        alpha = 1 + lambda\_T \* (1 - T), delta = lambda\_D \* (1 - D).

        """

        lamT = float(self.hyper["lambda\_T"])

        lamD = float(self.hyper["lambda\_D"])

        n = int(max(1, self.hyper.get("n\_shp", 12)))

        alpha = 1.0 + lamT \* (1.0 - float(T))

        delta = lamD \* (1.0 - float(D))

        total = 0.0

        for k in range(n):

            denom = alpha + k \* delta

            if denom <= 0:

                # Numerically unlikely, but guard.

                continue

            total += 1.0 / denom

        return float(total)

    def capacity(self, coords: Dict[str, float], D: float, H: float) -> float:

        """

        Capacity combines a gamma-term (sensitive to D and local AVG surface)

        and an exponential of entropy. Designed to grow with structural 'order'.

        """

        avg\_surface = (

            float(coords.get("At", 0.0))

            + float(coords.get("Ab", 0.0))

            + float(coords.get("Ex", 0.0))

            + float(coords.get("T", 0.0))

        ) / 4.0

        # Use scipy/math gamma wrapper (\_gamma) which handles non-integer args.

        gamma\_term = \_gamma(1.0 + float(self.hyper["beta"]) \* float(D) \* float(avg\_surface))

        entropy\_term = math.exp(float(self.hyper["eta"]) \* float(H))

        # Defensive cast to float

        return float(gamma\_term \* entropy\_term)

    def apply\_valence(self, V: float, C: float, SHP: float) -> float:

        """

        Apply domain valence (V in [0,1]) to capacity and SHP to produce Ct.

        Ct = V \* C \* (1 + kappa \* SHP)

        """

        return float(float(V) \* float(C) \* (1.0 + float(self.hyper["kappa"]) \* float(SHP)))

    def permutation(self, n: int, m: int) -> float:

        """Return nPm (permutation). If m > n return 0.0 to avoid errors."""

        n\_ = int(n)

        m\_ = int(m)

        if m\_ > n\_ or m\_ <= 0:

            return 0.0

        # Use factorial but guard large numbers

        try:

            return float(factorial(n\_) // factorial(n\_ - m\_))

        except (OverflowError, ValueError):

            # Fallback to product (safer for large numbers)

            prod = 1

            for i in range(n\_, n\_ - m\_, -1):

                prod \*= i

            return float(prod)

    # ---------- Agent selection pipeline ----------

    def gate\_agents(self, agents: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

        """

        Gate agents by a heuristic — keep those that satisfy at least one strong

        signal threshold. If none pass, fall back to top-k by 'Ct' value.

        Returns a list of agent dicts (preserving original fields + computed).

        """

        active = []

        for a in agents:

            At = float(a.get("At", 0.0))

            Ab = float(a.get("Ab", 0.0))

            H = float(a.get("H", 0.0))

            Ex = float(a.get("Ex", 0.0))

            T = float(a.get("T", 0.0))

            D = float(a.get("D", 0.0))

            # thresholding logic is intentionally conservative and preserved

            if (At >= 0.7) or (Ab \* H >= 0.35) or (Ex >= 0.6) or (T >= 0.6) or (D >= 0.5):

                active.append(a)

        if not active:

            # Fallback: sort by Ct (computed) desc and take up to 8

            agents\_sorted = sorted(agents, key=lambda x: float(x.get("Ct", 0.0)), reverse=True)

            active = agents\_sorted[:8]

        return active

    def calculate\_optimal\_potential(self, active\_agents: List[Dict[str, Any]]) -> Tuple[float, int]:

        """

        For m=1..len(active) compute phi\_m = (sum top m Ct) / permutation(R, m)

        Return (phi\_star, m\_star). If no agents returns (0.0, 0).

        """

        if not active\_agents:

            return 0.0, 0

        phis: List[float] = []

        R = int(max(1, self.hyper.get("R", 20)))

        for m in range(1, len(active\_agents) + 1):

            top\_m = sum(float(a.get("Ct", 0.0)) for a in active\_agents[:m])

            perm = self.permutation(R, m)

            phi\_m = (top\_m / perm) if perm > 0 else 0.0

            phis.append(phi\_m)

        idx = int(np.argmax(phis)) if phis else 0

        return float(phis[idx]), int(idx + 1)

    def generate\_prediction(self, phi\_star: float, m\_star: int) -> float:

        """

        Generate final prediction combining phi\_star and permutation-driven denominator.

        pred = gamma(1 + xi \* phi\_star) / (1 + rho \* permutation(R, m\_star))

        """

        perm = self.permutation(int(self.hyper["R"]), int(m\_star))

        num = \_gamma(1.0 + float(self.hyper["xi"]) \* float(phi\_star))

        den = 1.0 + float(self.hyper["rho"]) \* float(perm)

        if den == 0:

            return 0.0

        return float(num / den)

    # ---------- High-level run pipeline ----------

    def run(self, domain\_valence: float, agents: List[Dict[str, Any]]) -> Tuple[float, List[Dict[str, Any]]]:

        """

        Process list-of-agents (each agent a dict of raw features) and return:

          - prediction (float)

          - selected agents (list of dicts; each dict includes computed keys:

            D, H, SHP, C, Ct)

        Domain\_valence clipped to [0,1].

        """

        Vd = \_clamp01(domain\_valence)

        processed: List[Dict[str, Any]] = []

        for a in agents:

            # Ensure we don't mutate original dict

            coords = {

                "At": float(a.get("At", 0.0)),

                "Ab": float(a.get("Ab", 0.0)),

                "Ex": float(a.get("Ex", 0.0)),

                "T": float(a.get("T", 0.0)),

                "Cn": float(a.get("Cn", self.hyper["cn\_min"])),

            }

            D = self.depth\_transform(coords["Cn"])

            H = self.entropy([coords["At"], coords["Ab"], coords["Ex"], coords["T"]])

            SHP = self.shp(coords["T"], D)

            C = self.capacity(coords, D, H)

            Ct = self.apply\_valence(Vd, C, SHP)

            proc = dict(a)  # copy original fields

            proc.update({"D": D, "H": H, "SHP": SHP, "C": C, "Ct": Ct})

            processed.append(proc)

        active = self.gate\_agents(processed)

        phi\_star, m\_star = self.calculate\_optimal\_potential(active)

        pred = self.generate\_prediction(phi\_star, m\_star)

        # Return top-m\_star active agents as the chosen set (preserving order)

        return float(pred), active[:m\_star]

# ---------- Mapping helpers (kept compatible) ----------

def map\_to\_coords\_protein(row: Dict[str, Any]) -> Dict[str, float]:

    """

    Map a protein record (dictionary) to CL5D coordinate features:

      - At, Ab, Ex, T, Cn

    WARNING: This mapping encodes domain heuristics and is part of the

    project's fractal/valence specification. Do not change lightly.

    """

    aff = float(row.get("Affinity\_kcal\_per\_mol", 0.0))

    # affinity is negative for binding; normalize by an 8 kcal/mol scale

    At = \_clamp01((-aff) / 8.0)

    diseases = str(row.get("Associated\_Diseases", "") or "")

    n = len([d.strip() for d in diseases.split(",") if d.strip()])

    Ab = min(1.0, float(n) / 5.0)

    pdbid = str(row.get("PDB\_ID", "") or "")

    # If PDB\_ID includes 'AlphaFold' we downweight experimental evidence

    Ex = 0.6 if ("AlphaFold" in pdbid) else 1.0

    T = \_clamp01(abs(aff) / 8.0)

    # Cn uses a minimum and reduces with At/Ab (preserves fractal mapping)

    Cn = max(float(\_DEFAULT\_HYPER["cn\_min"]), min(1.0, 1.0 - (At \* 0.5 + Ab \* 0.5)))

    return {"At": At, "Ab": Ab, "Ex": Ex, "T": T, "Cn": Cn}

def map\_to\_coords\_compound(row: Dict[str, Any]) -> Dict[str, float]:

    """

    Map a compound record (with 'score' and 'smiles') to CL5D coords.

    Mapping mirrors prior heuristics and keeps ranges stable.

    """

    score = float(row.get("score", 0.0))

    At = \_clamp01(score)

    smi = str(row.get("smiles", "") or "")

    Ab = min(1.0, float(len(smi)) / 80.0)

    # crude chemical existence heuristic: presence of '1' or aromatic letters

    Ex = 1.0 if ("1" in smi or "c" in smi or "C" in smi) else 0.5

    T = At

    # Cn defined with combination weights similar to protein mapping

    Cn = max(float(\_DEFAULT\_HYPER["cn\_min"]), min(1.0, 1.0 - (At \* 0.4 + Ab \* 0.3 + Ex \* 0.3)))

    return {"At": At, "Ab": Ab, "Ex": Ex, "T": T, "Cn": Cn}

# ---------- Minimal demonstration when invoked as script ----------

if \_\_name\_\_ == "\_\_main\_\_":  # pragma: no cover - demo

    import json

    import os

    from pprint import pprint

    print("CL5D module demo running...")

    # Try to locate sample JSON files if present in working directory

    sample\_proteins = [

        {"Protein": "DRD2", "PDB\_ID": "6CM4", "Affinity\_kcal\_per\_mol": -5.516, "Associated\_Diseases": "Dystonia 11, MDD"},

        {"Protein": "PTGS2", "PDB\_ID": "3HS5", "Affinity\_kcal\_per\_mol": -6.843, "Associated\_Diseases": "Bursitis"},

    ]

    sample\_compounds = [

        {"compound\_id": "4531024", "score": 0.744, "smiles": "COC1=C(C=CC=C1)C(=O)N1CC2=C(CC1C)C=C1OCOC1=C2"},

        {"compound\_id": "4553762", "score": 0.740, "smiles": "CC1CC2=C(CN1C(=O)C1=CC3=C(OCO3)C=C1)C=C1OCOC1=C2"},

    ]

    model = ConsciousLeafModel()

    print("Mapping proteins -> coords and running model (domain\_valence=0.85)")

    prot\_agents = [dict(map\_to\_coords\_protein(p), \*\*{"name": p.get("Protein")}) for p in sample\_proteins]

    pred\_p, sel\_p = model.run(0.85, prot\_agents)

    print("Protein prediction:", pred\_p)

    print("Selected protein agents (top):")

    pprint(sel\_p)

    print("\nMapping compounds -> coords and running model (domain\_valence=0.80)")

    cmp\_agents = [dict(map\_to\_coords\_compound(c), \*\*{"name": c.get("compound\_id")}) for c in sample\_compounds]

    pred\_c, sel\_c = model.run(0.80, cmp\_agents)

    print("Compound prediction:", pred\_c)

    print("Selected compound agents (top):")

    pprint(sel\_c)

    print("\nDemo finished. If you want this model used from a notebook, import:\n"

          "from cl5d import ConsciousLeafModel, map\_to\_coords\_protein, map\_to\_coords\_compound\n")