```
"""
NeuroGraph Cognitive Enhancement Suite — Module 2: Activation Persistence  v1.1
================================================================================
Changes from v1.0
-----------------
- Normalization fallback: if the entire graph is quiet at snapshot time
  (max_v < default_threshold * 0.1), normalize against default_threshold
  instead of max_v to prevent over-inflation of faint signals.
- All magic numbers replaced with CESConfig lookups.
- reset() hook: clears/invalidates the snapshot when the graph is reset
  mid-session so stale node IDs don't leak across boundaries.
- Minor: snapshot_info() now reports whether the snapshot would produce a
  warm or cold start based on current elapsed time + decay math.
"""

from __future__ import annotations

import json
import logging
import time
from pathlib import Path
from typing import Any, Dict, List, Optional, Tuple

from ces_config import CESConfig, make_config

logger = logging.getLogger("neurograph.activation_persistence")

SNAPSHOT_VERSION: str = "1.1"


class ActivationPersistence:
    """
    Save and restore NeuroGraph activation state across sessions.

    Parameters
    ----------
    graph           neuro_foundation.Graph
    snapshot_path   Path to the JSON sidecar file
    cfg             CESConfig (optional)
    """

    def __init__(
        self,
        graph: Any,
```

```
    snapshot_path: "str | Path",
    cfg: Optional[CESConfig] = None,
) -> None:
    self._graph = graph
    self._path = Path(snapshot_path)
    self._cfg = cfg or make_config()

# ------------------------------------------------------------------
# Save
# ------------------------------------------------------------------

def save_session(self) -> bool:
    """Snapshot activation state. Returns True on success."""
    try:
        snapshot = self._build_snapshot()
        self._path.parent.mkdir(parents=True, exist_ok=True)
        self._path.write_text(json.dumps(snapshot, indent=2))
        logger.info(
            "Activation snapshot saved: %d nodes, %d predictions → %s",
            len(snapshot["nodes"]),
            len(snapshot["predictions"]),
            self._path,
        )
        return True
    except Exception as exc:
        logger.warning("Activation snapshot save failed: %s", exc)
        return False

def _build_snapshot(self) -> Dict[str, Any]:
    cfg = self._cfg
    min_v: float = cfg["ap_min_save_voltage"]
    top_n: int = cfg["ap_top_n_nodes"]
    top_p: int = cfg["ap_top_n_predictions"]

    # Collect nodes above noise floor
    node_entries: List[Tuple[float, str, float]] = []
    for nid, node in self._graph.nodes.items():
        if node.voltage >= min_v:
            node_entries.append((node.voltage, nid, node.firing_rate_ema))
    node_entries.sort(reverse=True)
    top_nodes = node_entries[:top_n]

    # v1.1: normalization fallback
    default_threshold = self._graph.config.get("default_threshold", 1.0)
    if top_nodes:
        max_v = top_nodes[0][0]
        # If graph is in a quiet state, don't over-inflate weak signals
```

```
    if max_v < default_threshold * 0.1:
        logger.debug(
            "Snapshot: graph is quiet (max_v=%.4f); "
            "normalising against default_threshold=%.4f",
            max_v, default_threshold,
        )
        max_v = default_threshold
else:
    max_v = default_threshold

node_snapshots = [
    {
        "node_id": nid,
        "norm_voltage": round(v / max_v, 4),
        "firing_rate_ema": round(ema, 6),
    }
    for v, nid, ema in top_nodes
]

# Active predictions
pred_snapshots: List[Dict[str, Any]] = []
active_preds = getattr(self._graph, "_active_predictions", {})
if active_preds:
    sorted_preds = sorted(
        active_preds.values(),
        key=lambda p: p.get("strength", 0),
        reverse=True,
    )
    for pred in sorted_preds[:top_p]:
        pred_snapshots.append({
            "target_node_id": pred.get("target_node_id"),
            "strength": round(pred.get("strength", 0), 4),
            "source_he_id": pred.get("source_he_id"),
        })

# Partially triggered hyperedges
partial_he: List[Dict[str, Any]] = []
for he_id, he in self._graph.hyperedges.items():
    if he.refractory_remaining > 0:
        continue
    active_members = [
        nid for nid in he.member_node_ids
        if nid in self._graph.nodes
        and self._graph.nodes[nid].voltage >= min_v
    ]
    if active_members:
        ratio = len(active_members) / max(len(he.member_node_ids), 1)
```

```
            if ratio >= 0.2:
                partial_he.append({
                    "hyperedge_id": he_id,
                    "active_ratio": round(ratio, 3),
                    "active_members": active_members,
                })

    return {
        "version": SNAPSHOT_VERSION,
        "saved_at": time.time(),
        "graph_timestep": self._graph.timestep,
        "nodes": node_snapshots,
        "predictions": pred_snapshots,
        "partial_hyperedges": partial_he,
    }

# ------------------------------------------------------------------
# Restore
# ------------------------------------------------------------------

def restore_ambient(self) -> Dict[str, Any]:
    """
    Apply Tier 1 ambient restoration.  Always safe to call.
    Returns a status dict (cold_start or warm_start).
    """
    if not self._path.exists():
        logger.info("No activation snapshot found — cold start.")
        return {"status": "cold_start", "reason": "no_snapshot"}

    try:
        snapshot = json.loads(self._path.read_text())
    except (json.JSONDecodeError, OSError) as exc:
        logger.warning("Corrupt activation snapshot — cold start: %s", exc)
        return {"status": "cold_start", "reason": str(exc)}

    # Accept both v1.0 and v1.1 snapshots
    snap_ver = snapshot.get("version", "")
    if snap_ver not in ("1.0", "1.1"):
        logger.info(
            "Snapshot version '%s' unrecognised — cold start.", snap_ver
        )
        return {"status": "cold_start", "reason": "version_mismatch"}

    cfg = self._cfg
    elapsed_hours = (
        time.time() - snapshot.get("saved_at", time.time())
    ) / 3600
```

```
decay = max(
    0.0,
    1.0 - cfg["ap_temporal_decay_per_hour"] * elapsed_hours,
)
effective_ratio = cfg["ap_ambient_restore_ratio"] * decay

if effective_ratio < cfg["ap_min_effective_ratio"]:
    logger.info("Snapshot too old (decay=%.3f) — cold start.", decay)
    return {"status": "cold_start", "reason": "snapshot_expired"}

threshold = self._graph.config.get("default_threshold", 1.0)
restored_nodes = skipped_nodes = restored_preds = restored_he = 0

# Tier 1: ambient node stimulation
for entry in snapshot.get("nodes", []):
    nid = entry.get("node_id")
    if not nid or nid not in self._graph.nodes:
        skipped_nodes += 1
        continue
    nudge = threshold * entry.get("norm_voltage", 0.5) * effective_ratio
    try:
        self._graph.stimulate(nid, nudge)
        restored_nodes += 1
    except Exception as exc:
        logger.debug("Failed to restore node %s: %s", nid, exc)
        skipped_nodes += 1

# Partial prediction restoration
for pred in snapshot.get("predictions", []):
    target = pred.get("target_node_id")
    if not target or target not in self._graph.nodes:
        continue
    nudge = (
        threshold
        * pred.get("strength", 0)
        * cfg["ap_prediction_restore_ratio"]
        * decay
    )
    try:
        self._graph.stimulate(target, nudge)
        restored_preds += 1
    except Exception:
        pass

# Partial hyperedge member restoration
he_completion = self._graph.config.get("he_pattern_completion_strength",
for he_entry in snapshot.get("partial_hyperedges", []):
```

```
            he_id = he_entry.get("hyperedge_id")
            if he_id not in self._graph.hyperedges:
                continue
            active_ratio = he_entry.get("active_ratio", 0)
            for nid in he_entry.get("active_members", []):
                if nid not in self._graph.nodes:
                    continue
                nudge = he_completion * active_ratio * effective_ratio * threshol
                try:
                    self._graph.stimulate(nid, nudge)
                    restored_he += 1
                except Exception:
                    pass

        logger.info(
            "Ambient restore: nodes=%d skip=%d preds=%d he_members=%d "
            "decay=%.2f elapsed_h=%.1f",
            restored_nodes, skipped_nodes, restored_preds, restored_he,
            decay, elapsed_hours,
        )

        return {
            "status": "warm_start",
            "restored_nodes": restored_nodes,
            "skipped_nodes": skipped_nodes,
            "restored_predictions": restored_preds,
            "restored_he_members": restored_he,
            "effective_restore_ratio": round(effective_ratio, 3),
            "elapsed_hours": round(elapsed_hours, 2),
        }

    # ------------------------------------------------------------------
    # Reset hook  (v1.1)
    # ------------------------------------------------------------------

    def reset(self) -> None:
        """
        Invalidate the snapshot when the graph is cleared mid-session.
        Renames the file rather than deleting, so it can be inspected.
        """
        if self._path.exists():
            invalid_path = self._path.with_suffix(".invalid.json")
            try:
                self._path.rename(invalid_path)
                logger.info(
                    "Activation snapshot invalidated (graph reset) → %s",
                    invalid_path,
```

```
            )
        except Exception as exc:
            logger.warning("Could not invalidate snapshot: %s", exc)

# ------------------------------------------------------------------
# Introspection
# ------------------------------------------------------------------

def snapshot_info(self) -> Dict[str, Any]:
    """Metadata about the saved snapshot, including warm/cold prediction."""
    if not self._path.exists():
        return {"exists": False}
    try:
        snapshot = json.loads(self._path.read_text())
        saved_at = snapshot.get("saved_at", 0)
        elapsed_h = (time.time() - saved_at) / 3600
        cfg = self._cfg
        decay = max(0.0, 1.0 - cfg["ap_temporal_decay_per_hour"] * elapsed_h)
        effective = cfg["ap_ambient_restore_ratio"] * decay
        would_warm = effective >= cfg["ap_min_effective_ratio"]
        return {
            "exists": True,
            "version": snapshot.get("version"),
            "saved_at": saved_at,
            "elapsed_hours": round(elapsed_h, 2),
            "node_count": len(snapshot.get("nodes", [])),
            "prediction_count": len(snapshot.get("predictions", [])),
            "partial_he_count": len(snapshot.get("partial_hyperedges", [])),
            "graph_timestep": snapshot.get("graph_timestep"),
            "would_warm_start": would_warm,
            "effective_restore_ratio": round(effective, 3),
        }
    except Exception as exc:
        return {"exists": True, "readable": False, "error": str(exc)}
```