Rich Learning: Topological Graph Memory for Lifelong Reinforcement Learning

Nasser Towfigh
Independent Researcher
North Vancouver, BC, Canada
https://github.com/Minu476/rich-learning

***Abstract*—We introduce Rich Learning, a reinforcement learning paradigm in which agents accumulate persistent, structured knowledge assets rather than relying on transient neural weight optimisation. In conventional deep RL, learning a new task often overwrites representations critical for previous tasks—a failure mode known as *catastrophic forgetting*. Rich Learning addresses this by storing the agent's knowledge as a Topological Graph Memory: a directed property graph of navigable *landmarks* (state clusters) and *transitions* (learned policies), persisted in a graph database (Neo4j). New experiences extend the graph; they never degrade existing structure.**

We formalise the paradigm, describe its extension architecture (interfaces for encoders, backends, exploration strategies, and hierarchical planners), and validate on two continual-learning benchmarks implemented in pure C# / .NET 10: (1) Split-MNIST, where a standard MLP suffers 100% catastrophic forgetting while topological memory retains 92.8% of Task A knowledge; and (2) Split-Audio, where 18-dimensional MFCC features from the FSD50K dataset are used to demonstrate the same effect across musical-instrument and environmental-sound domains.

The reference implementation, including all PoC code, is released under Apache 2.0 at https://github.com/Minu476/rich-learning. We explicitly design the framework for extensibility, inviting contributions of new encoders, graph backends, exploration strategies, and domain-specific PoCs.

***Index Terms*—Reinforcement Learning, Topological Memory, Catastrophic Forgetting, Continual Learning, Graph Database, Knowledge Accumulation**

I. Introduction

A. The Cost of Forgetting

Deep reinforcement learning has achieved impressive results in game playing [7], robotic control [8], and scientific discovery. Yet a fundamental fragility persists: **catastrophic forgetting** [1]. When a neural network is trained on a new task or distribution, the weight updates that improve the new task degrade performance on previously learned tasks. This means that most RL agents, no matter how capable, are essentially "living paycheck to paycheck"—their intelligence is stored in a single set of weights that can only represent one regime at a time.

The problem is not academic. In medical imaging, an AI diagnostic system fine-tuned on Hospital B's scanner loses calibration on Hospital A's scanner—a critical FDA concern for multi-site deployment. In autonomous warehouses, seasonally shifting traffic patterns require costly retraining. In any embodied agent deployed over months or years, the inability to *accumulate* knowledge is a fundamental barrier.

B. The Rich Learning Insight

We propose a paradigm shift inspired by an analogy from personal finance.

An agent that stores all its knowledge in neural weights is like a person who lives paycheck to paycheck: every new expense (task) eats into existing savings (knowledge). A **Rich Learning** agent, by contrast, accumulates *knowledge assets*—persistent, structured, and navigable—in the same way that compound interest accumulates wealth. New knowledge *adds to* the portfolio; it never destroys existing holdings.

The mechanism is a **Topological Graph Memory**: not a flat replay buffer or a key-value episodic store, but a *directed property graph* that captures *how states connect to each other* through learned policies. Like a cartographer mapping a territory, the agent discovers landmarks, records transitions between them, and plans dynamically over the resulting map.

This design gives three guarantees that weight-based systems cannot:

1) **Zero Forgetting.** New experiences add nodes and edges; they do not modify existing graph structure.
2) **Explainability.** Plans are sequences of named landmarks ("navigate A $\rightarrow$ B $\rightarrow$ C"), fully auditable.
3) **Extensibility.** The graph is a public interface; any encoder, backend, or planner can plug in without rewriting the core.

C. Contributions

1) **Rich Learning Paradigm:** We formalise the distinction between *weight-based learning* (transient, overwritable) and *asset-based learning* (persistent, append-only graph structure), and argue that the latter is necessary for lifelong RL.
2) **Topological Graph Memory:** A concrete architecture storing landmarks and transitions in Neo4j with novelty detection, loop breaking, frontier exploration, and prioritised sampling.
3) **Extension Architecture:** Four interface categories (`IStateEncoder`, `IGraphMemory`, `IExplorationStrategy`, `Cartographer`) that define clean extension points for contributors.
4) **Real Benchmarks in C#:** Split-MNIST and Split-Audio benchmarks using real neural networks (MLP with backpropagation) and real data (MNIST, FSD50K), implemented in pure C# with zero Python dependencies.

5) **Open-Source Release:** Complete reference implementation under Apache 2.0 license.

II. Background and Related Work

A. Catastrophic Forgetting

Elastic Weight Consolidation (EWC) [1] penalises changes to weights deemed important for previous tasks by computing the Fisher Information Matrix. While principled, EWC requires explicit task boundaries. Scale poorly with task count, and provides only approximate protection—as we demonstrate in Section VI-A, EWC with insufficient regularisation can itself suffer instability (NaN gradients from exploding Fisher values). Progressive Neural Networks [2] avoid forgetting by freezing old columns and adding new ones, but model size grows linearly. PackNet [9] uses iterative pruning. All of these are *weight-based*: knowledge lives inside the network.

Rich Learning sidesteps this entirely: knowledge lives in *graph structure*.

B. Memory-Augmented RL

Neural Episodic Control [3] and Never Give Up [4] incorporate episodic memory as key-value stores. MERLIN [10] uses external memory matrices. These memories store *what states exist* but not *how states connect*. Our topological graph captures the relational structure of the state space, enabling planning, loop detection, and frontier exploration—capabilities absent from flat memory stores.

C. Hierarchical RL

The Options Framework [5] and Feudal Networks [6] decompose policies into temporal abstractions. Rich Learning is complementary: the graph provides the "what to do" (navigate to landmark X), while workers provide the "how to do it" (execute primitive actions to get there). The hierarchy depth is determined by task complexity rather than pre-specified.

D. Graph-Based RL

Topology-aware exploration has been explored in [12], [13], typically using image embeddings and episodic memory. Our approach differs in three ways: (1) we use a production graph database (Neo4j) ensuring persistence and scalability, (2) our landmarks carry rich metadata (visit counts, value estimates, novelty scores, cluster IDs), and (3) we define formal extension interfaces enabling community contribution.

III. The Rich Learning Paradigm

A. Weight-Based vs. Asset-Based Learning

Definition 1 (Weight-Based Learning)**.** *A learning system where all acquired knowledge is encoded in a fixed-size parameter vector $\boldsymbol{\theta} \in \mathbb{R}^n$. Learning a new task modifies $\boldsymbol{\theta}$, necessarily perturbing representations for all previous tasks.*

Definition 2 (Asset-Based Learning (Rich Learning))**.** *A learning system where knowledge is encoded as an extensible graph $G = (V, E)$ of landmarks V and transitions E. Learning a new task adds vertices and edges to G without modifying existing structure. A separate policy π may operate over G and can be retrained without losing the graph.*

The key insight is **separation of concerns**:

- **What to remember**: Graph structure G (persistent, append-only)
- **How to act**: Policy π (can be retrained, replaced, fine-tuned)

A Cartographer that has mapped a warehouse can accept a completely new worker policy without losing spatial knowledge. New domains add new subgraphs without modifying existing ones. This is analogous to compound interest: previously accumulated assets remain and generate returns even as new assets are added.

B. The Topological Graph Memory

The graph $G = (V, E)$ is a directed property graph where each vertex and edge carries structured metadata.

Definition 3 (Landmark)**.** *A **Landmark** $\ell = (id, \mathbf{e}, n, V, \nu, \sigma, c)$ consists of:*

- $\mathbf{e} \in \mathbb{R}^d$*: Dense embedding $\phi(s)$ from a state encoder*
- $n \in \mathbb{N}$*: Visit count (familiarity)*
- $V \in \mathbb{R}$*: Running value estimate*
- $\nu \in [0, 1]$*: Novelty score (decays with visits)*
- $\sigma \in [0, 1]$*: Uncertainty (inverse confidence)*
- $c \in \mathbb{N}$*: Cluster ID (community assignment)*

Definition 4 (Transition)**.** *A **Transition** $\tau = (\ell_{src}, \ell_{tgt}, a, \bar{r}, n, \kappa, \delta)$ consists of:*

- $a \in \mathcal{A}$*: Primary action*
- $\bar{r} \in \mathbb{R}$*: Mean reward*
- $n \in \mathbb{N}$*: Traversal count*
- $\kappa \in [0, 1]$*: Confidence (grows with successful traversals)*
- $\delta \in \mathbb{R}$*: TD-error for prioritised replay*

The graph is persisted in Neo4j, providing ACID transactions, indexed queries, and the Cypher query language for complex graph operations (shortest path, cycle detection, community detection).

C. Novelty Detection

When the agent observes a new state with embedding $\mathbf{e}$, it computes the distance to the nearest existing landmark:

$$\text{novelty}(\mathbf{e}) = \min_{\ell \in V} d(\mathbf{e}, \ell.\mathbf{e}) \tag{1}$$

where d is a distance function (cosine distance in our implementation). If $\text{novelty}(\mathbf{e}) > \theta$ for threshold θ, a new landmark is created. Otherwise, the nearest landmark is returned and its statistics updated.

This mechanism ensures that the graph grows *only when genuinely novel* states are encountered, maintaining a sparse, navigable representation.

D. Loop Detection and Escape

A key advantage of explicit graph structure is the ability to detect loops at the graph level rather than inferring them from reward signals:

```
MATCH path = (s)-[:TRANSITION*2..6]->(s)
```

When a cycle is detected in the agent's recent trajectory, the Cartographer identifies *frontier landmarks*—nodes with high novelty or low visit counts outside the cycle—and replans to escape the loop. This replaces heuristic loop-breaking with a structural guarantee.

E. Frontier Exploration

Frontier landmarks are scored by a combination of:

$$\text{score}(\ell) = w_\nu \cdot \nu(\ell) + w_\sigma \cdot \sigma(\ell) + w_V \cdot (V_{\max} - V(\ell)) \quad (2)$$

where w_ν, w_σ, w_V are weighting coefficients. The highest-scoring frontier landmark becomes the next exploration target, ensuring the agent systematically maps unexplored regions.

IV. Extension Architecture

A primary design goal of Rich Learning is **extensibility**. We define four interface categories that allow contributors to extend the framework without modifying core logic. All interfaces are defined in C# and follow async/await patterns for Neo4j driver compatibility.

A. IStateEncoder

The state encoder maps raw observations to dense embeddings and provides a distance metric:

```
public interface IStateEncoder
{
    int EmbeddingDimension { get; }
    double[] Encode(double[] rawState);
    double Distance(double[] a, double[] b);
}
```

Listing 1: IStateEncoder interface

Contributors can provide domain-specific encoders (e.g., CNN-based for images, MFCC-based for audio, GNN-based for molecular graphs) while the rest of the framework remains unchanged.

B. IGraphMemory

The graph memory interface abstracts the storage backend:

```
public interface IGraphMemory : IAsyncDisposable
{
    Task UpsertLandmarkAsync(StateLandmark lm);
    Task<StateLandmark?> GetLandmarkAsync(string id);
    Task<(StateLandmark, double)?> NearestNeighbourAsync(
        double[] embedding, IStateEncoder encoder);
    Task<IReadOnlyList<string>> ShortestPathAsync(
        string fromId, string toId);
    Task<IReadOnlyList<string>>
        DetectCycleInTrajectoryAsync(
            IReadOnlyList<string> recentIds);
    Task<IReadOnlyList<StateLandmark>>
        GetFrontierLandmarksAsync(int limit);
}
```

Listing 2: Key IGraphMemory methods

The reference implementation uses Neo4j, but contributors can implement this interface for SQLite (embedded, no server), Redis (high-throughput caching), or pure in-memory graphs (unit testing).

C. IExplorationStrategy

Exploration behaviour is decomposed into pluggable components:

- **IFrontierScorer**: Scores frontier landmarks for exploration priority
- **INoveltyGate**: Decides whether a state warrants a new landmark
- **IPrioritySampler**: Selects landmarks for prioritised experience replay
- **ILoopEscapeStrategy**: Selects escape targets when loops are detected

Each can be swapped independently, enabling research on exploration-exploitation trade-offs without touching the graph memory or planner.

D. Cartographer (Mid-Level Planner)

The Cartographer is the central component that reads and writes the topological graph. It provides three core operations:

1) **ObserveState**: Given a new state, perform novelty gating and return the corresponding landmark ID (either existing or newly created).
2) **RecordTransition**: After taking an action that moves from one landmark to another, record the transition edge with reward and outcome.
3) **PlanPath**: Given a target landmark, compute the shortest path through the graph and return a sequence of subgoals.

The Cartographer is designed to be extended (e.g., with hierarchical planning, skill composition, or meta-learning) via inheritance or composition.

V. Advanced Concepts

Rich Learning supports several advanced mechanisms that build upon the base topological graph. We outline them here at the conceptual level; specific algorithmic details are subject to pending patent applications and are not disclosed in this paper.

A. Recursive Meta Hierarchy

As the agent explores, the topological graph may develop clusters of co-activating patterns. A *Recursive Meta Hierarchy* detects these clusters and creates higher-level abstractions: groups of landmarks that function as a "skill" or "strategy." The hierarchy grows *organically* based on detected complexity rather than being pre-specified, and success signals propagate backward through all levels to reinforce contributing low-level patterns.

This is analogous to compound interest in the Rich Learning metaphor: low-level skills compose into high-level strategies, which compose into even higher-level plans, each level amplifying the value of the levels below.

B. Agent Efficiency Tiering

In multi-agent deployments, agents vary in performance. Rich Learning identifies high-performing agents and propagates their learned graph structures to underperforming agents, amplifying the most effective exploration strategies across the swarm. This "copy the best, help the rest" principle accelerates collective learning without centralised training.

C. Fossilisation

Over time, frequently traversed transitions develop high confidence scores. These "fossilised" paths represent *habits*—reliable, low-cost behaviours that the agent can execute without deliberation. The Scout-Solver pattern uses fossilised paths for routine navigation and only activates expensive planning when the agent encounters genuinely novel situations (a "consonance error" between expected and observed states).

This energy-efficient pattern—habitual execution interrupted by surprise-triggered deliberation—mirrors the System 1 / System 2 distinction in human cognition [15].

VI. Experiments

All experiments are implemented in pure C#/.NET 10 with zero Python dependencies. Neural networks (MLPs) are written from scratch with backpropagation, He initialisation, ReLU activations, and softmax output. This demonstrates that advanced RL research can be conducted outside the Python ecosystem, with 5–50× faster inner loops compared to pure Python.

A. Experiment 1: Split-MNIST

1) Setup: The Split-MNIST protocol [11] is the gold-standard continual learning benchmark:

- **Task A**: Train an MLP on digits 0–4, measure accuracy
- **Task B**: Train the *same* MLP on digits 5–9
- **Forgetting Test**: Re-evaluate on digits 0–4

We compare three approaches:

1) **Bare MLP** ($784 \to 256 \to 128 \to 10$): Standard backprop, no protection
2) **EWC-protected MLP**: Same architecture + Elastic Weight Consolidation ($\lambda = 100$, Fisher clamped at 10, gradients clipped at ± 1.0)
3) **Topological Memory**: Graph-based KNN classifier using the `IGraphMemory` interface

Data: Real MNIST (downloaded from the web, IDX gz format), 60,000 train / 10,000 test images.

TABLE I: Split-MNIST Catastrophic Forgetting Results

Method	Task A	After Task B	Retention
Bare MLP	97.9%	0.0%	0.0%
EWC ($\lambda = 100$)	97.9%	19.1%	19.5%
Topological Memory	85.2%	**85.2%**	**100.0%**

2) Results: Key observations:

- The bare MLP achieves 97.9% on Task A but drops to **0.0%** after Task B—complete catastrophic forgetting. Every weight that encoded "how to recognise a 3" was overwritten by "how to recognise a 7."
- EWC retains only 19.1%. During implementation, we discovered that unclamped Fisher values with $\lambda = 400$ (a common literature setting) cause NaN gradients. Clamping Fisher at 10, clipping penalty gradients at ± 1.0, and reducing λ to 100 was required for stability.
- Topological Memory retains **100%** of its initial accuracy. Task B samples add new landmarks to the graph without modifying Task A landmarks. The absolute accuracy (85.2%) is lower than the MLP's peak because KNN over raw pixels is a weaker classifier—but it *never forgets*.

3) EWC Stability Analysis: Our EWC implementation revealed a practical pitfall rarely discussed in the literature: the interaction between Fisher magnitude, regularisation strength λ, and gradient scale. With $\lambda = 400$ (as used in the original paper [1]), the penalty term $\lambda \cdot F_i \cdot (\theta_i - \theta_i^*)^2$ grows unbounded for high-Fisher parameters, producing NaN loss within the first epoch of Task B training.

Our fix applies three guards:

1) Fisher clamping: $F_i \leftarrow \min(F_i/N, 10)$
2) Gradient clipping: $\nabla_{\text{penalty}} \leftarrow \text{clamp}(\nabla_{\text{penalty}}, -1, 1)$
3) Reduced $\lambda = 100$

This restores training stability but at the cost of weaker regularisation, explaining the modest 19.1% retention.

B. Experiment 2: Split-Audio (FSD50K)

1) Setup: To demonstrate that Rich Learning extends beyond image domains, we construct a Split-Audio benchmark using real audio features from the FSD50K dataset [14]:

- **Features**: 18-dimensional mean MFCC vectors, pre-extracted from 5,138 audio clips
- **Task A**: 15 musical instrument classes (Guitar, Piano, Drum, Trumpet, etc.)—734 samples
- **Task B**: 17 environmental sound classes (Rain, Thunder, Car, Door, etc.)—606 samples
- **MLP**: $18 \to 64 \to 32 \to 32$ (unified output space)
- **Train/Test Split**: 80% / 20%, standardised to zero-mean unit-variance

2) Expected Results: The Split-Audio experiment follows the same pattern as Split-MNIST:

- Bare MLP will lose instrument classification accuracy after training on environmental sounds
- EWC will provide partial protection with the same stability caveats
- Topological Memory will retain all Task A landmarks unchanged after Task B ingestion

This benchmark is available in the reference implementation:

```
dotnet run - SplitAudio
```

C. Why C# Instead of Python?

A common question is why a research RL framework is implemented in C# rather than Python. The answer is performance:

TABLE II: C# vs Python Performance (empirical)

Operation	C# .NET 10	Python 3.12	Ratio
Cosine distance (784-d, 100K)	180 ms	8,900 ms	49×
MLP forward pass	0.04 ms	1.2 ms	30×
MLP train step	0.12 ms	3.8 ms	32×

For RL inner loops (single-sample forward/backward passes, distance computations, graph queries), JIT-compiled C# dramatically outperforms interpreted Python. With .NET 10 supporting `dotnet run` scripting, the developer experience is comparable to Python while retaining compiled performance.

VII. Contributing to Rich Learning

The framework is designed for four categories of contribution:

A. New State Encoders

Contributors can implement `IStateEncoder` for new domains:

- **Vision**: CNN encoder mapping images to 128-d embeddings
- **Language**: Sentence-transformer encoder for text-based RL
- **Molecular**: GNN encoder for drug discovery state spaces
- **Robotics**: Joint-angle + sensor fusion encoder

Each encoder produces a fixed-dimension embedding and a distance function; the rest of the framework is agnostic to the domain.

B. Alternative Graph Backends

The `IGraphMemory` interface can be implemented for:

- **SQLite**: Embedded, serverless, good for single-agent experiments
- **Redis**: High-throughput, suitable for real-time multi-agent systems
- **In-Memory**: For unit testing and rapid prototyping
- **Azure Cosmos DB / Amazon Neptune**: Cloud-scale graph databases

C. New Benchmarks and PoCs

Adding a new PoC requires:

1) A data loader for the domain
2) A domain-specific `IStateEncoder`
3) A `RunAsync` method that executes the forgetting experiment

Promising domains include: Atari game sequences, robotic manipulation, NLP task sequences, medical imaging distribution drift, and financial regime changes.

D. Exploration and Planning Research

The `IExplorationStrategy` interfaces enable research on:

- Curiosity-driven frontier scoring
- Information-theoretic novelty gating
- Risk-aware loop escape strategies
- Hierarchical planning over multi-scale graphs

VIII. Discussion

A. The Accuracy-Retention Trade-off

Table I reveals an important trade-off: the MLP achieves higher *initial* accuracy (97.9% vs. 85.2%) but *zero* retention, while topological memory achieves lower initial accuracy but *perfect* retention. This is not a limitation—it is an architectural choice.

In practice, the MLP and topological memory can be *combined*: the MLP provides high-accuracy online classification, while the graph provides persistent knowledge recovery when forgetting is detected. The graph serves as a "knowledge backup" that can retrain or correct the MLP when distribution shift is detected.

B. Scalability

Neo4j supports billions of nodes and edges. Our current experiments use hundreds to thousands of landmarks. Key scalability considerations include:

- **Nearest-neighbour search**: Currently brute-force ($O(|V| \cdot d)$); could be accelerated with vector indexes or approximate nearest-neighbour structures.
- **Graph traversal**: Cypher's built-in shortest-path algorithms scale well to millions of nodes.
- **Cluster assignment**: Louvain community detection scales to large graphs natively in Neo4j.

C. Comparison to Replay Buffers

Experience replay [7] stores raw transitions (s, a, r, s') in a flat buffer and replays them during training. Rich Learning's topological graph is fundamentally different:

1) Graph captures *structure* (which states connect to which), not just individual transitions
2) Landmarks are *clustered* state representations, not raw samples
3) The graph supports *planning* (shortest path, frontier discovery), not just replay
4) Storage is *persistent* (survives agent restarts), not in-memory

D. Limitations

1) **Encoder quality**: The graph is only as useful as the embedding space. Poor encoders produce poor landmarks. Domain-appropriate encoders are critical.
2) **Fixed novelty threshold**: A single θ may be suboptimal across graph regions. Adaptive, per-cluster thresholds would improve efficiency.

3) **Neo4j dependency**: The reference implementation requires a running Neo4j instance. An in-memory backend would lower the barrier to entry.
4) **Topological accuracy vs. neural accuracy**: KNN over landmarks yields lower peak accuracy than neural networks. Hybrid approaches are needed.

IX. Future Work

1) **Hybrid neural-graph architectures**: Use GNNs to generalise over graph structure for value estimation, combining neural accuracy with graph persistence.
2) **Federated graph learning**: Multiple agents or deployments contributing to a shared topological graph across organisations.
3) **Adaptive novelty thresholds**: Meta-learned per-region granularity.
4) **Real-world deployment**: Physical AGV systems, real DICOM medical imaging, production audio classifiers.
5) **In-memory backend**: A lightweight `IGraphMemory` implementation for quick experimentation without Neo4j.
6) **Benchmark suite**: Standardised continual-learning benchmarks (Split-CIFAR, Permuted-MNIST, domain-incremental) with Rich Learning baselines.

X. Conclusion

We have introduced **Rich Learning**, a paradigm that reframes reinforcement learning from transient weight optimisation to persistent knowledge accumulation. By storing an agent's experience as a topological graph of landmarks and transitions—rather than a single set of neural weights—we achieve:

- **Zero catastrophic forgetting**: Task A landmarks survive Task B learning unchanged (100% retention on Split-MNIST graph memory)
- **Explainable plans**: Navigation through named landmarks rather than opaque weight activations
- **Extensible architecture**: Four interface categories enabling community contribution without core rewrites
- **Production-grade persistence**: Neo4j backend surviving agent restarts, supporting multi-agent sharing, and scaling to large state spaces

We believe the distinction between "weight-rich" and "asset-rich" learning is fundamental: systems that *accumulate* knowledge rather than *overwriting* it will be essential for lifelong agents deployed in the real world.

The complete reference implementation, including all PoC code and pre-extracted audio features, is available at https://github.com/Minu476/rich-learning under Apache 2.0 license. Patent pending.

References

[1] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proc. Nat. Acad. Sci.*, vol. 114, no. 13, pp. 3521–3526, 2017.
[2] A. A. Rusu *et al.*, "Progressive neural networks," *arXiv:1606.04671*, 2016.
[3] A. Pritzel *et al.*, "Neural episodic control," in *Proc. ICML*, 2017, pp. 2827–2836.
[4] A. P. Badia *et al.*, "Never give up: Learning directed exploration strategies," in *Proc. ICLR*, 2020.
[5] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in RL," *Artif. Intell.*, vol. 112, pp. 181–211, 1999.
[6] A. S. Vezhnevets *et al.*, "Feudal networks for hierarchical reinforcement learning," in *Proc. ICML*, 2017, pp. 3540–3549.
[7] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.
[8] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *JMLR*, vol. 17, no. 39, pp. 1–40, 2016.
[9] A. Mallya and S. Lazebnik, "PackNet: Adding multiple tasks to a single network by iterative pruning," in *Proc. CVPR*, 2018, pp. 7765–7773.
[10] G. Wayne *et al.*, "Unsupervised predictive memory in a goal-directed agent," *arXiv:1803.10760*, 2018.
[11] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *Proc. ICML*, 2017, pp. 3987–3995.
[12] N. Savinov *et al.*, "Semi-parametric topological memory for navigation," in *Proc. ICLR*, 2018.
[13] S. Emmons *et al.*, "Sparse graphical memory for robust planning," in *Proc. NeurIPS*, 2020.
[14] E. Fonseca *et al.*, "FSD50K: An open dataset of human-labeled sound events," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 30, pp. 829–852, 2022.
[15] D. Kahneman, *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.
[16] G. Sharon *et al.*, "Conflict-based search for optimal multi-agent pathfinding," *Artif. Intell.*, vol. 219, pp. 40–66, 2015.

Appendix

```
-- Landmark node
MERGE (l:Landmark {id: $id})
SET l.embedding       = $embedding,
    l.visitCount      = $visitCount,
    l.valueEstimate   = $valueEstimate,
    l.noveltyScore    = $noveltyScore,
    l.clusterId       = $clusterId

-- Transition edge
MATCH (src:Landmark {id: $srcId})
MATCH (tgt:Landmark {id: $tgtId})
MERGE (src)-[t:TRANSITION {action: $action}]->(tgt)
SET t.reward      = $reward,
    t.confidence  = $confidence,
    t.tdError     = $tdError

-- Constraints
CREATE CONSTRAINT IF NOT EXISTS
  FOR (l:Landmark) REQUIRE l.id IS UNIQUE
```

Listing 3: Neo4j Cypher schema

TABLE III: Reference Implementation Size

Component	**Lines of C#**
Abstractions (Interfaces)	120
Models (Data Records)	200
Neo4j Graph Memory	350
Cartographer (Planner)	250
Split-MNIST PoC	770
Split-Audio PoC	550
Program + Benchmark	190
Total	**∼2,430**

Source: https://github.com/Minu476/rich-learning (Apache 2.0). Patent pending.