Learning Cognitive Maps from Transformer Representations for Efficient Planning in Partially Observed Environments

Antoine Dedieu [1] **Wolfgang Lehrach** [1] **Guangyao Zhou** [1] **Dileep George** [1] **Miguel Lázaro-Gredilla** [1]

Abstract

Despite their stellar performance on a wide range of tasks, including in-context tasks only revealed during inference, vanilla transformers and variants trained for next-token predictions (a) do not learn an explicit world model of their environment which can be flexibly queried and (b) cannot be used for planning or navigation. In this paper, we consider partially observed environments (POEs), where an agent receives perceptually aliased observations as it navigates, which makes path planning hard. We introduce a transformer with (multiple) discrete bottleneck(s), `TDB`, whose latent codes learn a compressed representation of the history of observations and actions. After training a `TDB` to predict the future observation(s) given the history, we extract interpretable cognitive maps of the environment from its active bottleneck(s) indices. These maps are then paired with an external solver to solve (constrained) path planning problems. First, we show that a `TDB` trained on POEs (a) retains the near-perfect predictive performance of a vanilla transformer or an LSTM while (b) solving shortest path problems exponentially faster. Second, a `TDB` extracts interpretable representations from text datasets, while reaching higher in-context accuracy than vanilla sequence models. Finally, in new POEs, a `TDB` (a) reaches near-perfect in-context accuracy, (b) learns accurate in-context cognitive maps (c) solves in-context path planning problems.

1. Introduction

Large vanilla transformers (Vaswani et al., 2017) trained for next-token prediction have been successfully applied to a wide range of applications such as natural language processing (Radford et al., 2018; Chowdhery et al., 2022), text-conditioned image generation (Ramesh et al., 2021), reinforcement learning (Chen et al., 2021a), and code writing (Chen et al., 2021b). These large language models (LLMs) also exhibit *emergent* abilities (Wei et al., 2022), among which is their capacity to *in-context learn* (ICL), i.e., to adapt to a new task at inference time given a few examples (Brown et al., 2020). However, these models suffer from shortcomings that prevent them from being used for planning (Valmeekam et al., 2022; Mialon et al., 2023).

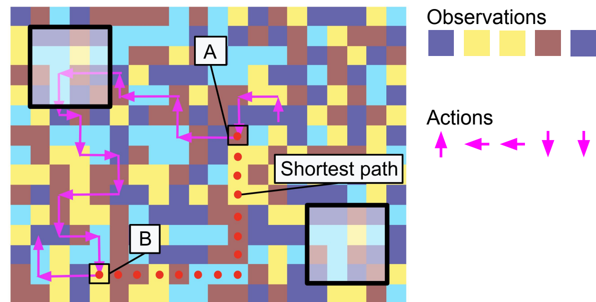

Figure 1. An agent is trained on random walks in an aliased room with no reward and unknown layout. At test time, given a novel random walk (in fuchsia), it has to find a shortest path (in red) between room positions A and B. While a vanilla transformer solves this path planning problem with forward rollouts, which can be exponentially expensive due to aliasing, our transformer variant pairs its learned cognitive map with an external solver.

Herein, we consider a suite of partially observed environments (POEs) (Chrisman, 1992) where the agent receives perceptually aliased observations as it navigates; and cannot deterministically recover its spatial positions from its observations. Path planning in these POEs is hard: the planner has to disambiguate aliasing to locate itself, which requires modeling its history of observations and actions. Path planning in POEs is particularly hard for LLMs, as these models do not learn an interpretable cognitive map (Momennejad et al., 2023; Whittington et al., 2022)—i.e. an action-conditioned latent graph modeling the POE dynamics (George et al., 2021)—which can be flexibly queried.

Example of a path planning problem in a POE that a vanilla transformer cannot solve: We consider the partially observed 15×20 room in Fig.1, which only contains four unique observations. The room also has global aliasing:

[1]Google DeepMind. Correspondence to: Antoine Dedieu <adedieu@google.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

the 4×4 patch with a black border appears twice. An agent executes a series of discrete actions, each action leading to a discrete observation. The agent does not have access to any reward or to the room layout. If an action takes an agent to an invalid location (e.g. the agent hits a wall), the action is executed but the agent remains in place. At test time, the agent wants to find a shortest path (in red) between two room positions (**A** and **B**) it has been to. A transformer trained to predict the next observation given the history of past observations and actions can only perform forward roll-outs, which, due to aliasing, (a) scale exponentially in the ℓ_1 distance between **A** and **B** (b) prevents it from even knowing when it reaches the target **B**.

In this paper, we propose the transformer with discrete bottleneck(s), `TDB`, which adds a single, or multiple, discrete bottleneck(s) (Van Den Oord et al., 2017) on top of a transformer to compress the transformer outputs into a finite number of latent codes. We train `TDB` with an augmented objective, then extract a cognitive map (Whittington et al., 2022; George et al., 2021) from its active bottleneck(s) indices. First, we demonstrate that, on aliased rooms, on aliased cubes with non-Euclidean dynamics, and on visually rich 3D environments (Beattie et al., 2016), these learned cognitive maps (a) disambiguate aliasing (b) nearly recover the ground truth dynamics and (c) can be paired with external solvers to solve path planning problems—like the one in Fig.1. Consequently, `TDB` (a) retains the nearly perfect predictive abilities of vanilla transformers and LSTMs, (b) solves (constrained) path planning problems exponentially faster. Second, we show that a `TDB` extracts an interpretable latent graph from a text dataset (Xie et al., 2021) while achieving higher test accuracies than vanilla sequence models. Finally, when exploring a new POE at test time, `TDB` can (a) in-context predict the next observation given history (b) solve in-context path planning problems (c) learn accurate in-context cognitive maps.

The rest of this paper is organized as follows. Sec.2 discusses related work. Sec.3 details our proposed `TDB` model. Finally, Sec.4 compares our method with vanilla transformer and LSTM in a variety of navigation and text experiments.

2. Related Work

Planning with LLMs: Despite their success on simple benchmarks including maths (Cobbe et al., 2021) and logics (Srivastava et al., 2022), growing evidence suggests that LLMs performance collapse on benchmarks that require stronger planning skills (Valmeekam et al., 2022). While Pallagani et al. (2022) finetune LLMs for planning, Guan et al. (2023) extract an approximate world model with LLMs, then further refine it with an external planner.

Interpretable circuits: Some works try to reverse-engineer LLMs to extract interpretable circuits, i.e., internal structures that drive certain model behaviors (Elhage et al., 2021; Olsson et al., 2022; Nanda et al., 2023). While circuit analysis can be performed at scale (Lieberum et al., 2023), it is labor intensive (Conmy et al., 2023), and does not extract explicit structures to solve planning or navigation tasks.

Decision transformers: Decision transformers (Chen et al., 2021a) abstract reinforcement learning (RL) as sequence modeling: they can play Atari games and stack blocks with a robot arm (Reed et al., 2022). However, they only have been applied to shortest path problems on small graphs with 20 nodes. Decision transformers have also access to a reward, which reveals information about the environment. They would struggle to solve the shortest path in Fig.1, where aliasing is high and no external reward is provided.

Learning spatial representations and world models in POEs: Fraccaro et al. (2018) propose an action-conditioned generative model that uses a spatial memory to store disentangled spatial and visual representations. When tested in 2D and 3D POEs, their model is able to correctly predict future observations given future actions, over hundreds of timesteps. Similarly, Guo et al. (2022) learns a forward model in latent space to predict future latent representations from current latent representations. The authors use the model uncertainty as an intrinsic reward to solve hard exploration tasks in POEs. However, both works (a) do not try to extract interpretable world models of their environments and (b) do not solve planning problems at inference time in highly aliased environments. In a recent work, Lamb et al. (2022) use a discrete bottleneck on top of a visual encoder. The authors showed that a multi-step inverse dynamics loss can be used to learn control-sufficient representations that discard visual distractors. They also build a transition graph from the learned representation they use to solve navigation tasks. However, the authors consider fully visible settings and learn representations for each observation by discarding the history: their approach would not solve the shortest path problem in Fig.1 or in Section 4, where the observations are highly aliased.

Clone-structured causal graphs (CSCGs): CSCG (George et al., 2021) is an interpretable probabilistic model that learns cognitive maps in aliased POEs and solves path planning tasks. However, CSCGs learn a block-sparse transition matrix over a large latent space via the expectation-maximization (EM) algorithm (Dempster et al., 1977). This large transition matrix occupies a large amount of memory and limits CSCGs' scalability—see Appendix B.1 for a detailed discussion. CSCGs would also struggle to solve the ICL experiments in Sec.4.5 without an external algorithm—see Swaminathan et al. (2023). In contrast, the transformer variant we introduce in this paper, `TDB`, builds this same transition matrix sparsely, via local counts. Appendix B.2

discusses a CSCG-inspired variant of our proposed model.

QMDP-Net: The QMDP-Net (Karkus et al., 2017) is a neural network architecture for planning in POE with aliased observations. QMDP-Net assumes that the agent has access to (a) a map of its environment, (b) a goal defined in an unaliased way, (c) an initial aliased observation. During training, QMDP-net has access to expert trajectories that go from the initial state to the known goal. The agent is then trained via behavioral cloning on these expert trajectories. In contrast, our proposed `TDB` never has access to a cognitive map of its unknown environment. It builds a map after training, from the transformer quantized representations. Second, aliasing is never broken for `TDB`: it never has access to unaliased goal states. Third, our `TDB` model is trained to predict the next observation on random walks in an unknown environment. In fact, during training, `TDB` does not know that it will be asked to solve planning problems at test time.

3. A Path Planning-Compatible Transformer

3.1. Problem statement

Problem: We consider an agent executing a series of discrete actions $a_1, \dots, a_{N-1}$ with $a_n \in \{1, \dots, N_{\text{actions}}\}$, e.g. walking in a room. As a result of each action, the agent receives a perceptually aliased observation (Chrisman, 1992), resulting in the stream $x_1, \dots, x_N$. The agent does not have access to any reward at any time. Our goal is to (a) predict the next observation given the history of past observations and actions and (b) build a world model that disambiguates aliased observations and that can be called by an external planner to solve path planning tasks.

Trajectory representation: We represent a trajectory $\tau = (x_1, a_1, \dots, a_{N-1}, x_N)$ by alternating observations and actions. This representation allows us to consider the autoregressive objective of predicting the next observation given the history of past observations and actions:

$$\mathcal{L}_{\text{obs}} = \sum_{n=1}^{N-1} \mathcal{L}_{\text{obs}}(n) = -\sum_{n=1}^{N-1} \log p(x_{n+1} | x_1, a_1, \dots, x_n, a_n) \tag{1}$$

3.2. Predicting the next observation with a transformer

We propose to train a vanilla transformer (Vaswani et al., 2017) to minimize Equation (1). To do so, we first map each categorical observation x_n (resp. action a_n) of τ to a linear embedding $E_{\text{obs}}(x_n) \in \mathbb{R}^D$ (resp. $E_{\text{act}}(a_n) \in \mathbb{R}^D$). Second, we feed the sequence of input embeddings $z = (E_{\text{obs}}(x_1), E_{\text{act}}(a_1), \dots, E_{\text{obs}}(x_N), E_{\text{act}}(a_N))$ to a transformer with causal mask (Radford et al., 2018), resulting in the output sequence $(T_1, \dots, T_{2N})$. We only retain the outputs with *even* indices $(T_2, T_4, \dots, T_{2N})$: $T_{2n} \in \mathbb{R}^D$ is derived after applying the successive self-attention layers to the intermediate representations of $x_1, a_1, \dots, x_n, a_n$. Finally, a linear mapping applied to T_{2n} returns the predicted logits for the next observation x_{n+1}. We display this transformer architecture in Fig.6, Appendix A.

As we show in Sec.4, this transformer almost perfectly predicts the next observation on a suite of perceptually aliased POEs. However, it can only solve the path planning problem in Fig.1 with forward rollouts, which have an exponential cost in the ℓ_1 distance between the room positions **A** and **B**.

3.3. A transformer with a discrete bottleneck

To address these inherent limitations, we add a discrete bottleneck on top of a vanilla transformer, which compresses all the information needed to minimize Equation (1).

After applying the causal transformer from Sec.3.2 to the sequence z of observations and actions embeddings, we now only extract the encodings with *odd* indices, which results in the stream $(T_1, T_3, \dots, T_{2N-1})$: $T_{2n-1} \in \mathbb{R}^D$ is derived from $x_1, a_1, \dots, a_{n-1}, x_n$, but is not aware of the action a_n. We then apply vector quantization (Van Den Oord et al., 2017) and map each vector $T_{2n-1}, \forall n$ to the closest entry in a dictionary of latent codes $\mathcal{D} = (d_1, \dots, d_K)$, where $d_k \in \mathbb{R}^D$, $\forall k$[1]. That is, we introduce the operator

$$\phi(y) = \underset{k=1,\dots,K}{\operatorname{argmin}} \|y - d_k\|_2^2, \ \forall y. \tag{2}$$

The quantized output of T_{2n-1} is $d_{\phi(T_{2n-1})}$. Because T_{2n-1} has not been exposed to a_n, we derive the next observations logits via $f\left(d_{\phi(T_{2n-1})} \oplus E_{\text{act}}(a_n)\right)$, where f is a two layers MLP and $\oplus$ represents the concatenation operator. We refer to our proposed transformer with a discrete bottleneck as `TDB`, and visualize it in Figure 2.

3.4. Augmenting the objective value

The discrete bottleneck in Sec.3.3 introduces the additional objective term $\mathcal{L}_{\text{bot}} = \sum_{n=1}^{N} \mathcal{L}_{\text{bot}}(n)$ with

$$\mathcal{L}_{\text{bot}}(n) = \|d_{\phi(e)} - \text{sg}(e)\|_2^2 + \beta \| \text{sg}\left(d_{\phi(e)}\right) - e\|_2^2, \tag{3}$$

where $e = T_{2n-1}$, $\beta = 0.25$ and sg the stop-gradient operator (Van Den Oord et al., 2017). However, we show in Fig.7, Appendix G, that a `TDB` trained with the simple objective $\mathcal{L}_{\text{obs}} + \mathcal{L}_{\text{bot}}$ fails at disambiguating observations with identical neighbors. To allow the `TDB` to learn richer representations of the observations, we propose to augment the transformer objective via two solutions.

Solution 1: Predicting the next S observations. At each timestep n, instead of only predicting the next observation—which corresponds to $S = 1$—we predict the next S observations and replace the loss $\mathcal{L}_{\text{obs}}(n)$ in Equation (1) with

[1] We explored using Finite Scalar Quantization (FSQ) (Mentzer et al., 2023) as an alternative to vector quantization. While FSQ guarantees dense codebooks, we were unable to solve the challenging in-context learning experiments in Section 4.5.

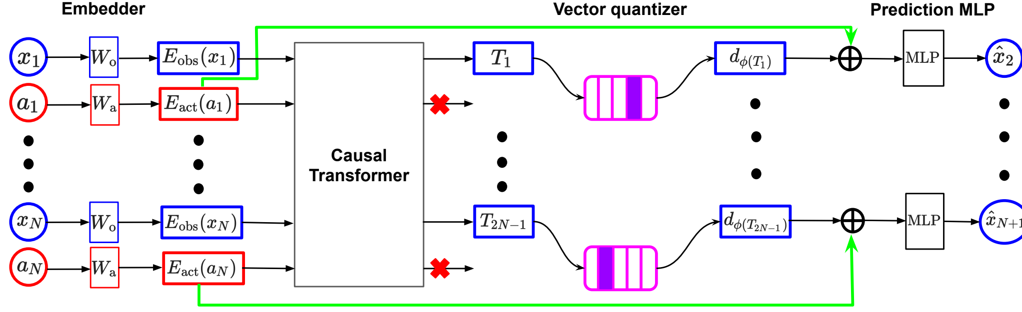

Figure 2. Our proposed transformer with a (single) discrete bottleneck. The respective linear embeddings of observations and actions go through a causal transformer. The observation outputs are compressed by the vector quantizer, then concatenated with the next action embedding in order to predict the next observation. Finally, a cognitive map of the environment is built from the active bottleneck indices.

$$\mathcal{L}_{\text{obs}}^{S}(n) = -\frac{1}{S}\sum_{s=0}^{S-1} \log p(x_{n+s+1}|x_1, a_1, \ldots, x_n, a_n, \ldots, a_{n+s}) \tag{4}$$

The model does not have access to any observation after x_n: it only sees the S actions $a_n, \ldots, a_{n+S-1}$. The logits of the observation x_{n+s+1} are predicted via $f^{(s)}\left(d_{\phi(T_{2n-1})} \oplus E_{\text{act}}(a_n) \oplus \ldots \oplus E_{\text{act}}(a_{n+s})\right)$, where $f^{(s)}$ is a two-layers MLP. In Sec.4, we set $S = 3$.

Solution 2: Predicting the next encoding. Our second solution is to predict future latent representations (Guo et al., 2022). We refer to the TDB described so far as a student network, with weights Θ, and introduce a teacher network (Grill et al., 2020), with the same architecture and whose weights Θ^{teacher} are an exponential moving average of Θ. At each gradient update, we set $\Theta^{\text{teacher}} \leftarrow \alpha\Theta^{\text{teacher}} + (1-\alpha)\Theta$, where α is the decay rate, which we fix to 0.99.

The student network predicts both (a) the next observation and (b) the next teacher network's quantized encoding. This introduces the objective term $\mathcal{L}_{\text{enc}} = \sum_{n=1}^{N-1} \mathcal{L}_{\text{enc}}$ with $\mathcal{L}_{\text{enc}}(n) =$ (5)

$$\left\| \underbrace{\frac{g\left(d_{\phi(T_{2n-1})} \oplus E_{\text{act}}(a_n)\right)}{\left\|g\left(d_{\phi(T_{2n-1})} \oplus E_{\text{act}}(a_n)\right)\right\|_2}}_{\text{using } \Theta} - \text{sg}\left(\underbrace{\frac{d_{\phi(T_{2n+1})}}{\|d_{\phi(T_{2n+1})}\|_2}}_{\text{using } \Theta^{\text{teacher}}}\right)\right\|_2^2$$

where g is a two-layer MLP with same hidden layer as f.

3.5. Extension to multiple discrete bottlenecks

TDB supports $M > 1$ discrete bottlenecks. Each bottleneck induces its own (a) dictionary of latent codes $\mathcal{D}^{(i)} = (d_1^{(i)}, \ldots, d_K^{(i)})$, (b) function $\phi^{(i)}$ which acts as in Equation (2) and (c) objective $\mathcal{L}_{\text{bot}}^{(i)}$. The transformer output T_{2n-1} passes through the discrete bottlenecks in parallel: the ith bottleneck returns the latent code $d^{(i)}_{\phi^{(i)}(T_{2n-1})}$. We get back to the previous case by defining (a) $d_{\phi(T_{2n-1})} = \oplus_{i=1}^{M} d^{(i)}_{\phi^{(i)}(T_{2n-1})}$ and (b) $\mathcal{L}_{\text{bot}} = \sum_{i=1}^{M} \mathcal{L}_{\text{bot}}^{(i)}$. As we show in Sec.4, multiple discrete bottlenecks make training faster and sometimes better. However, Appendix F introduces a disentanglement metric which shows that the representations learned by multiple discrete bottlenecks are highly redundant and not disentangled.

3.6. Learning a cognitive map from bottleneck indices

When we use a single bottleneck, TDB maps each observation x_n to the latent code index $s_n = \phi(T_{2n-1}) \in \{1, \ldots, K\}$. We define a count tensor C which counts the empirical latent transitions: $C_{ijk} = \sum_{(s_n, a_n, s_{n+1})} \mathbf{1}(s_n = i, a_n = j, s_{n+1} = k)$. We then threshold $\mathcal{C}$ to build an action-augmented empirical transition graph $\mathcal{G}$. That is, $\mathcal{G}$ has an edge between the vertices i and k with the action $j^* = \text{argmax}_j C_{ijk}$ iff. $C_{ij^*k} \geq t_{\text{ratio}} \cdot c^*$, where $c^* = \max_{ijk} C_{ijk}$ and t_{ratio} is a threshold. This edge indicates that the action j^* leads from the latent node i to the latent node k a large number of times.

In practice (a) we use multiple discrete bottlenecks (see Sec.3.5) and cluster the bottleneck indices using the Hamming distance with a threshold d_{Hamming} and (b) we map each discarded node of $\mathcal{C}$ to a retained node in $\mathcal{G}$ so that we can solve all the path planning problems in Sec.4. We detail (a) and (b) in Appendix C. As a result, each node in $\mathcal{G}$ is paired with a collection of tuples of bottleneck indices.

$\mathcal{G}$ is a cognitive map of the agent's environment which (a) is interpretable, (b) is action-conditioned, hence, models the agent's dynamics, and (c) can be paired with an external planner to solve path planning problems—see Sec.4. In particular, we show in Sec.4.2 that it can be used to solve the motivation example from Sec.1.

3.7. Why does TDB learn an accurate map?

The TDB latent index $\phi(T_{2n-1})$ associated with an observation x_n given its history corresponds to a node in the latent graph $\mathcal{G}$ in Sec.3.6. For $\mathcal{G}$ to correctly model the environment's dynamics, when this node is active, the agent must always be at the same ground truth spatial position.

Method	TestAccu (%) ↑	ImpFallback (%) ↑	RatioSP ↓	NormGED ↓
Vanilla transformer	99.00 (0.01)	29.53 (0.75)	16.82 (0.92)	—
Vanilla LSTM	98.85 (0.01)	31.97 (0.66)	16.81 (0.83)	—
TDB$(S=1, M=1)$	98.94 (0.05)	6.61 (0.53)	**1.00** (0.00)	0.532 (0.005)
TDB$(S=1, \text{enc}, M=1)$	98.74 (0.04)	99.20 (0.20)	**1.00** (0.00)	**0.118** (0.022)
TDB$(S=3, M=1)$	**99.06** (0.01)	**99.55** (0.13)	**1.00** (0.00)	0.124 (0.019)
TDB$(S=3, M=4)$	99.00 (0.01)	96.59 (0.15)	**1.00** (0.00)	0.186 (0.018)

Table 1. Results averaged over 10 aliased 15×20 rooms with $O = 4$ observations. See main text for a description of the metrics. Arrows pointing up (down) indicate that higher (lower) is better. Our TDB with either multi-step objective or next encoding prediction (a) retains the nearly perfect test accuracy of vanilla sequence models, (b) consistently solves the shortest paths problems when paired with an external solver—while both transformer or LSTM catastrophically fail—(c) learns cognitive maps nearly isomorphic to the ground truth.

$d_{\phi(T_{2n-1})}$ does not know the next action a_n; and compresses all the information needed to minimize the loss at the nth step. When TDB predicts the next observation, $d_{\phi(T_{2n-1})}$ must then encode the next observation that each next possible action leads to. As we show in Appendix G, $d_{\phi(T_{2n-1})}$ may not disambiguate distinct spatial positions with identical neighbors. When TDB predicts the next S observations (Solution 1, Sec.3.4), $d_{\phi(T_{2n-1})}$ must now encode all the observations $x_{n+s}, 1 \leq s \leq S$, that each sequence of S actions leads to. For a large S, the latent index $\phi(T_{2n-1})$ is encouraged to only be active at a unique spatial location. An alternative approach is to augment $d_{\phi(T_{2n-1})}$ so that it compresses all its neighbors' encoding (Solution 2).

4. Computational Results

We assess the performance of our proposed TDB on four experiments: (a) synthetic 2D aliased rooms and aliased cubes, (b) simulated 3D environments, (c) text datasets and (d) mixtures of 2D aliased rooms. Each experiment is run on a 2×2 grid of Tensor Processing Unit v2.

4.1. Methods compared

Each experiment compares the following methods:

• The causal transformer described in Sec.3.2, trained with the autoregressive objective in Equation (1). The architecture considered has 4 layers, 8 attention heads, a context length of 400, an embedding dimension $D = 256$, and an MLP hidden layer dimension of 512. We use relative positional embedding (Dai et al., 2019) and Gaussian error linear units activation functions (Hendrycks & Gimpel, 2016).

• Several TDBs, using the same causal transformer. We refer to each model as TDB(S, M) where (a) $S \in \{1, 3\}$ is the number of prediction steps (Solution 1, Sec.3.4), (b) $M \in \{1, 4\}$ is the number of discrete bottlenecks (see Sec.3.5)—each one contains $K = 1000$ latent codes. We note TDB(S, enc, M) when we predict the next encoding (Solution 2, Sec.3.4). After training, we build a cognitive map as in Sec.3.6 using $d_{\text{Hamming}} = 0.25, t_{\text{ratio}} = 0.1$.

• A vanilla LSTM (Hochreiter & Schmidhuber, 1997) with hidden dimension of 256.

4.2. Random walks in 2D aliased rooms

Problem: We consider a 2D aliased ground truth (GT) room of size 15×20, containing $O = 4$ unique observations, and a 4×4 patch repeated twice—similar to Fig.1. An agent walking selects, at each timestep, a discrete action $a_n \in \{1, 2, 3, 4\}$ and collects the categorical observation $x_n \in \{1, \dots, O\}$. The actions have unknown semantics: the agent does not know that $a_n = 1$ corresponds to moving up. No assumptions about Euclidean geometry are made either. The training and test set contains 2048 random walks of observations and actions of length 400 each.

Training: We train each method in Sec.4.1 with Adam (Kingma & Ba, 2014) for 25k training iterations, using a learning rate of 0.001 and a batch size of 32. For regularization, we use a dropout of 0.1.

Solving path planning problems at test time: For a test sequence x^{test} of length $N = 400$, let pos_n be the GT *unknown* 2D spatial position of the observation x_n. At test time, given a context $C = 50$, the path planning task is to derive a *shortest path*, i.e. a sequence of actions, which leads from pos_C to pos_{N-C} in the GT room. We emphasize that (a) the model only observes x_n and does not know pos_n (b) the path $(a_C, \dots, a_{N-C-1})$ is a solution, referred to as *fallback path*. Fig.1 shows this problem for $N = 40, C = 5$.

For a vanilla sequence model, we autoregressively sample 100 random rollouts of length $N - 2C$ each and collect all the *candidates*, i.e., the observations equal to x^{test}_{N-C}. Because of aliasing, the model cannot know whether a candidate is at the target position pos_{N-C}. We use the tail of the last C observations and actions and only return the candidates estimated to be at pos_{N-C}. For a TDB, after learning the cognitive map $\mathcal{G}$, we map x_C and x_{N-C} to the bottleneck indices $\phi(T_{2C-1})$ and $\phi(T_{2(N-C)-1})$, then to

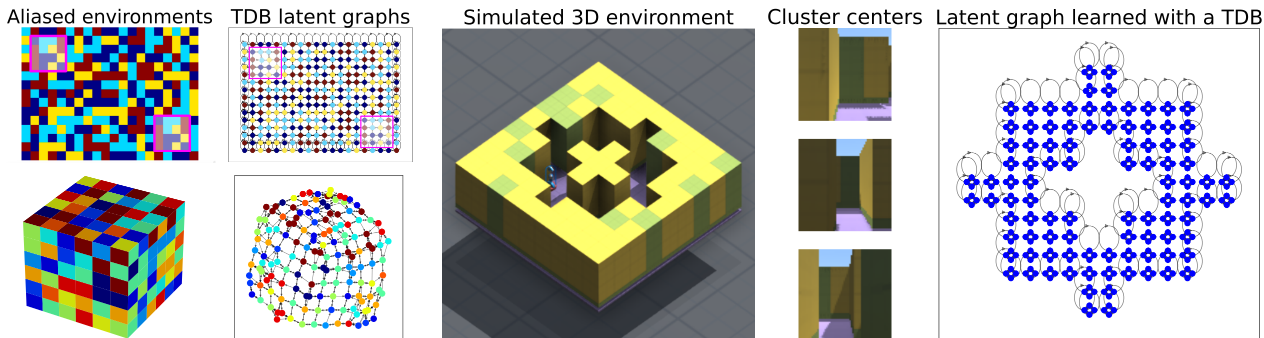

Figure 3. [Left] Top: 2D aliased room of size 15×20 with $O = 4$ unique observations and 2 identical 4×4 patches (in fuchsia). Bottom: Aliased cube of edge size 6 with $O = 12$ and non-Euclidean dynamics. [Center left] Top: cognitive map learned with a TDB($S = 3, M = 1$). For visualization, each latent node is mapped with the observation (resp. is placed at the 2D GT spatial position) with higher empirical frequency when this node is active. Bottom: similar, but we use the Kamada-Kawai algorithm. [Center] Isometric view of a simulated 3D environment. The agent navigates with egocentric actions and collects RGB images. [Center right] Three cluster centers: the cluster indices serve as categorical observations. [Right] Cognitive map learned with a TDB($S = 3, M = 4$): each location is represented by four nodes in the latent graph, corresponding to the four agent's heading directions. Also see Fig.12, Appendix J.

two nodes of $\mathcal{G}$ using the Hamming distance. We then call the external solver `networkx.shortest_path` (Hagberg et al., 2008) to find the shortest path between these two nodes. See Appendix D for details about both procedures.

Metrics: We evaluate each method for four metrics.

• A *prediction* metric, `TestAccu`: next observation accuracy on the test random walks—regardless of whether the agent was trained to predict the $S > 1$ future observations.

• Two *path planning* metrics: we consider 200 shortest paths problems and report (a) `ImpFallback`, the percentage of problems for which a model finds a *valid* path which improves over the fallback path—a path (i.e. a sequence of actions) is valid if it correctly leads from pos_C to pos_{N-C} in the GT room—and (b) `RatioSP`, when a better valid path is found, the ratio between its length and an optimal path length—we derive an optimal path using the GT room.

• Normalized *graph edit distance*, defined for two graphs $\mathcal{G}_1, \mathcal{G}_2$ as $\text{NormGED}(\mathcal{G}_1, \mathcal{G}_2) = \text{GED}(\mathcal{G}_1, \mathcal{G}_2)/(\text{GED}(\mathcal{G}_1, \emptyset) + \text{GED}(\mathcal{G}_2, \emptyset))$ where `GED` is the graph edit distance (Sanfeliu & Fu, 1983) and $\emptyset$ is the empty graph. `NormGED` is lower than 1.0 and is equal to 0.0 iff. $\mathcal{G}_1$ and $\mathcal{G}_2$ are isomorphic. We use a timeout of 15 minutes and empirical positions to accelerate the `GED` computation—see Appendix H.

Results: Table 1 averages the metrics over 10 experiments: each run considers a different GT room. First, we observe that both vanilla LSTM and transformer almost perfectly predict the next observation given the history. However, both sequence models struggle to solve path planning problems with forward rollouts: they only improve over the fallback path $\sim 30\%$ of the same. When they do so, they find shorter paths 16 times longer than an optimal path[3].

Second, all the `TDB`s retain the predictive performance of vanilla models. However, `TDB`($S = 1, M = 1$) fails at path planning: it only finds better valid paths $\sim 7\%$ of the time as it only solves the simplest problems[2]. Indeed, the learned cognitive maps merge the observations with identical neighbors, which introduces unrealistic *shortcuts* in the latent graphs—see Fig.7, Appendix G. In contrast, `TDB` with either (a) multi-step or (b) next encoding objective learns cognitive maps that recover the GT dynamics and reach low `NormGED`—see Fig.3[center left]. They are able to almost consistently find an optimal shortest path. As a result, `ImpFallback` is above 99% while `RatioSP` is 1.00.

Multiple discrete bottlenecks accelerate training: Table 4, Appendix E, reports the average number of training steps to reach a 98% train accuracy: for $S = 3$, this number goes from 8300 for $M = 1$ down to 4000 for $M = 4$. In addition, multiple discrete bottlenecks seem more robust to the environment. Table 8, Appendix I.2, shows that, when the room contains $O = 12$ unique observations, all the `TDB`s with $M = 4$ outperform their $M = 1$ counterparts.

Multiple discrete bottlenecks do not learn a factorized latent space: We test whether multiple discrete bottlenecks learn a disentangled representation by training a logistic regression to predict each bottleneck index given the other indices—see Appendix F for details. If the bottlenecks were independent, test disentanglement accuracy would be 0.10%. However, it is $84.99\%(\pm 0.56\%)$, which shows the strong redundancy in the representations learned by each bottleneck. This is explained by the fact that, the same transformer output, T_{2n-1}, passes through the discrete bottlenecks in parallel (see Sec. 3.5). Each bottleneck then compresses T_{2n-1} in a very similar way, to extract information that help predict the next observation: the bottlenecks are not incentivized to learn a factorized representation.

[2]When `TDB`($S = 1, M = 1$) finds a better valid path, it does finds an optimal path, which explains its `RatioSP` of 1.00

Method	TestAccu (%) ↑	ImpFallback (%) ↑	RatioSP ↓	NormGED ↓
Vanilla transformer	**99.80** (0.00)	26.60 (0.58)	16.38 (0.87)	—
Vanilla LSTM	99.79 (0.00)	26.85 (0.58)	16.38 (0.87)	—
$\texttt{TDB}(S=1, \text{enc}, M=1)$	72.88 (2.86)	2.85 (1.33)	1.32 (0.21)	0.733 (0.082)
$\texttt{TDB}(S=3, M=1)$	58.51 (1.43)	1.35 (0.27)	1.35 (0.13)	0.946 (0.007)
$\texttt{TDB}(S=1, \text{enc}, M=4)$	99.79 (0.01)	97.55 (0.42)	1.03 (0.01)	**0.005** (0.001)
$\texttt{TDB}(S=3, M=4)$	99.79 (0.00)	**99.95** (0.05)	**1.02** (0.00)	0.017 (0.004)

Table 2. Metrics averaged over 10 simulated 3D environments. Training a TDB with a single discrete bottleneck does not converge. In contrast, a TDB with multiple discrete bottlenecks can almost perfectly (a) predict the next observation given history, (b) solve shortest paths problems and (c) learn cognitive maps nearly isomorphic to the ground truth map.

The agent can locate itself after a short context: Appendix I.3 first shows that, on test random walks, accuracy is initially low then quickly increases: it is above 99.50% after the 30th observation—see Fig.9. Second, the latent codes can be divided into two clusters. The first cluster contains codes with low empirical frequencies, which model the agent's uncertainty in the early steps of the random walks. The second cluster contains latent codes with high empirical frequencies, which appear later in the random walks when the agent is confident about its location—see Fig.10. Finally, the agent can "teleportate" in the room but such behavior appears less than 1% of the time after the 60th observation—see Fig.11.

TDB can solve constrained path planning problems: TDB can inject constraints on demand into the external solver. To illustrate this, we consider a path planning problem variant: the new task is to find the shortest path from pos_C to pos_{N-C} which avoids, when possible, a randomly picked color. To avoid a color, we simply call the shortest path solver with a large penalty for all the edges connecting to a node with this color. $\texttt{TDB}(S=3, M=1)$ is still able to almost consistently solve the problem: ImpFallback is still $99.55\%(\pm 0.13\%)$ while RatioSP is still $1.00(\pm 0.00)$. The average length of the shortest paths is now $13.13(\pm 0.23)$, as opposed to $10.73(\pm 0.08)$ when all the colors are allowed.

TDB learn maps for non-Euclidean dynamics: Since TDB does not make any assumptions about its environment or its dynamics, it can be applied to aliased environments with non-Euclidean dynamics. We generate data from a 3D aliased cube with edge size 6 and $O = 12$ unique observations—see Fig.3[bottom left]. Table 7, Appendix I.1 shows that the prediction and planning performance of a TDB are near perfect. Fig.3[bottom - center left] visualizes the learned graph with the Kamada-Kawai algorithm (Kamada et al., 1989) and reveals that TDB recovers the POE non-Euclidean dynamics.

TDB handles noise during training: We explore how the performance of TDB varies as we increase the amount of noise in the training data. We replace a ratio $p_{\text{noise}} \in \{0.1, 0.15, 0.2, 0.25, 0.3\}$ of the training observations by randomly selected observations, and increase the number of sequences in the training set to 8092. As we see in Table 3, even when 30% of the training observations are flipped at random, the test accuracies of both vanilla transformers and TDB are above 90%, which demonstrates that both models are robust to noise. In addition, TDB still significantly outperforms the vanilla transformer for solving planning problems in the presence of noise: it is still able to solve $\sim 95\%$ of the planning problems on aliased environments when 20% of the observations are randomly replaced.

p_{noise}	TestAccu (%)		ImpFallback (%)	
	Transformer	TDB	Transformer	TDB
0.10	98.88 (± 0.04)	**99.08** (± 0.01)	29.50 (± 1.42)	**99.49** (± 0.20)
0.15	98.45 (± 0.05)	**98.85** (± 0.03)	22.66 (± 1.27)	**98.84** (± 0.37)
0.20	97.58 (± 0.10)	**98.30** (± 0.06)	19.87 (± 0.65)	**94.18** (± 1.37)
0.25	95.98 (± 0.16)	**96.98** (± 0.13)	15.83 (± 1.08)	**83.70** (± 3.21)
0.30	91.92 (± 0.33)	**93.92** (± 0.33)	8.11 (± 1.02)	**58.20** (± 2.80)

Table 3. Test accuracies and planning performance of a vanilla transformer and a $\texttt{TDB}(S=3, M=1)$ with increasing amount of noise in the training data.

4.3. Egocentric views in 3D simulated environments

Problem: We consider a suite of visually rich 3D simulated environments (Beattie et al., 2016) with a similar setting as Guntupalli et al. (2023)—see Fig.3[center, center right]. At each step, the agent can take any of three discrete *egocentric* actions—move 1m forward, rotate 90 deg left, rotate 90 deg right—and it sees a new view, which is a 64×64 RGB image. The agent's egocentric views are clustered with a k-means quantizer—with $k = 128$— and the clusters indices are used as categorical observations. The training and test set sizes are identical to Sec.4.2.

Training: We train the same models as in Sec.4.2, with the same parameters, on 10 synthetic 3D rooms.

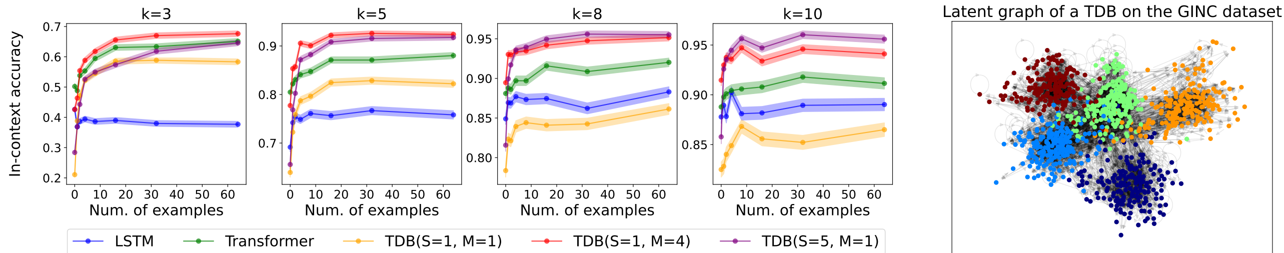

Figure 4. [Left] For all context lengths k, $\texttt{TDB}(S=1, M=4)$ achieves higher in-context accuracies than an LSTM and a vanilla transformer on the GINC test dataset (Xie et al., 2021)—while $\texttt{TDB}(S=5, M=1)$ is the best model for large contexts. [Right] The learned latent graph of $\texttt{TDB}(S=5, M=1)$ exhibits five clusters, each corresponding to a color-coded concept.

Results: Table 2 averages the metrics over 10 3D environments. Despite their excellent predictive performance, vanilla sequence models only find a path better than fallback for $26.60\%(\pm 0.60\%)$ of the problems: these "better" paths are $16.38(\pm 0.87)$ times longer than optimal paths[3]. Compared with Sec.4.2, training TDBs with a single bottleneck do not converge. Consequently, these models reach a low test accuracy and cannot solve the path planning problems. In contrast, for TDBs with $M=4$ discrete bottlenecks, averaged train accuracies are at least 99.70% after 1000 only training steps (see Table 5, Appendix E). These models reach nearly perfect (a) test accuracy and (b) path planning performance. Their latent maps accurately model the POEs dynamics—see Fig.3[right] and Fig.12, Appendix J. However, the multiple bottlenecks learn a highly redundant representation: the test disentanglement accuracy of a $\texttt{TDB}(S=3, M=4)$ is 99.06% $(\pm 0.06\%)$.

4.4. Extension to text via the GINC dataset

Problem: Herein, we extend TDB to text datasets to show that it can extract interpretable latent graphs on this other modality. We consider the text GINC dataset, introduced in Xie et al. (2021) to study in-context learning (ICL). GINC[4] is generated from a uniform mixture of five factorial HMMs (Ghahramani & Jordan, 1995)—referred to as a *concepts*. Each concept has two factorial chains with 10 states each. Each state can emit 50 observations shared across concepts. The training set consists of 1000 training documents with a total of ∼10 million tokens: each document uniformly selects a concept and samples independent sentences from it. The test set consists of in-context prompts. Each prompt has between $n=0$ and $n=64$ examples: each example is of length $k \in \{3, 5, 8, 10\}$. There are 2500 prompts for each setting (k, n). Each prompt uniformly selects a concept, samples $n-1$ examples $x^{(1)}_{:k}, \dots, x^{(n-1)}_{:k}$ of length k, and one example $x^{(n)}_{:k-1}$ of length $k-1$. The in-context task is to infer the most likely last token of the last example, i.e., $\text{argmax}_{x^{(n)}_{k-1}}\, p\left(x^{(n)}_{k-1} | x^{(1)}_{:k}, \dots, x^{(n-1)}_{:k}, x^{(n)}_{:k-1}\right)$.

Since the vocabulary is shared among the concepts, observations in GINC are aliased like in natural language. Solving the task requires the model to disambiguate the aliased observations and correctly infer the latent concepts.

Training: We train the same models as above using the same parameters, except the batch size which we set to 24.

Results: Fig. 4[left] reports the in-context accuracy—defined as the average ratio of correct predictions—for each pair (k, n) of the GINC test set. $\texttt{TDB}(S=1, M=1)$ is outperformed by vanilla sequence models. However, $\texttt{TDB}(S=1, M=4)$ outperforms both transformer and LSTM for all context lengths k. For larger contexts $k \in \{8, 10\}$, the best model is $\texttt{TDB}(S=5, M=1)$—it is however slower to converge (see Table 6, Appendix E). Finally, the highest in-context accuracies for TDB are around 95% and are higher than what was reported in Xie et al. (2021) for transformers. The numerical values are in Table 9, Appendix K.

Fig. 4[right] displays the latent graph learned by $\texttt{TDB}(S=5, M=1)$. We use $t_{\text{ratio}}=0.001$ and the Kamada-Kawai algorithm. Each latent node has a color corresponding to the GT concept with higher empirical frequency when this node is active. The learned latent graph has an interpretable structure: TDB learns five latent subgraphs corresponding to the five concepts in the GINC dataset.

4.5. In-context learning experiments

Problem: Our last experiment explores whether vanilla models and TDB can, in a new test POE, (a) in-context predict the next observation (b) solve in-context path planning problems (c) learn in-context latent graphs. Similar to Sec.4.2, we first generate a 2D aliased room of size 6×8 with $O=4$ unique colors, and with global aliasing: a 3×3 patch is repeated twice. We define the *room partition* as the partition $\mathcal{S}_1, \dots, \mathcal{S}_O$ of the GT room spatial positions such that all the room spatial positions in a set $\mathcal{S}_i$ have the same observation. We then generate many aliased rooms of the same size by preserving the room partition: each room

[3] The transformer and LSTM performance are almost identical because the same random walks are used for both models.

[4] available at https://github.com/p-lambda/incontext-learning/tree/main/data.

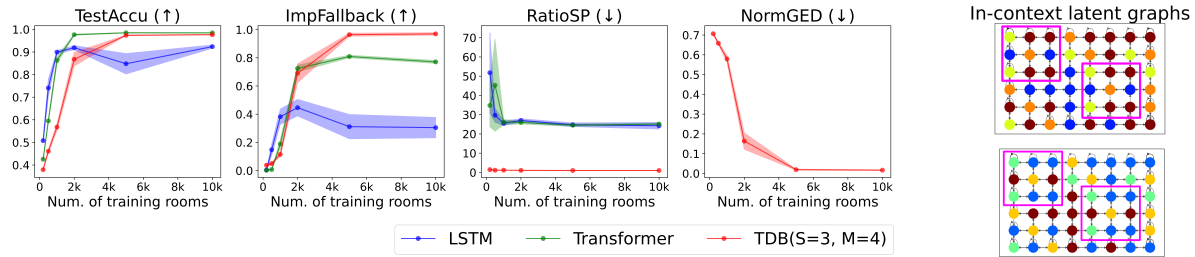

Figure 5. [Left] In *novel* 2D aliased test rooms, $\texttt{TDB}(S=3, M=4)$ perfectly (a) predicts the next observation (b) solves in-context path planning problems. These in-context capacities emerge when the number of training rooms increases. A vanilla transformer only solves the prediction problem (a)—which an LSTM struggles to do. [Right] Two latent graphs in-context learned by the `TDB` on two new test rooms. By design (a) a 3×3 fuchsia-coded patch is repeated twice and (b) the room partitions induced by the colors are the same.

picks 4 colors out of 30 without replacement, and assigns all the room spatial positions in $\mathcal{S}_i$ to the same color. See Appendix L.1 for details.

We train each model for a varying number of training rooms: $200, 500, 1\text{k}, 2\text{k}, 5\text{k}, 10\text{k}$ [5]. The training set contains 8092 sequences: each sequence picks a training room at random, then generates a random walk of length 400 in it. The test set only contains four random walks in each *new* test room. We define in-context path planning problems as before, using a context $C=100$. Given a new test room, `TDB` derives the test bottleneck indices, builds an in-context cognitive map, and uses it to solve the in-context path planning problems.

Training in base: We map a sequence of observations $x=(x_1, x_2, \ldots, x_N)$ in a room—where x_n is one of the four room observations—to a room-agnostic sequence $\tilde{x}=(\tilde{x}_1, \ldots, \tilde{x}_N)$ where (a) $\tilde{x}_n \in \{1, \ldots, O\}$ and (b) $\tilde{x}_n = k$ iff. $\tilde{x}_n$ is the kth lowest observation seen between indices 1 and n[6]. We train each model to take x as input and to predict $\tilde{x}$. As the model predictions are always in the *base* $\{1, \ldots, O\}$, it is encouraged to share structure across rooms—e.g. `TDB` can reuse the same latent codes across rooms. As before, we train for 25k Adam iterations, using a learning rate of 0.001, a batch size of 32 and a dropout of 0.1.

Results: Fig. 5[left] averages the results over 10 runs—the numerical values are in Table 10, Appendix L.2. For `NormGED`, we use a timeout of $20s$. As the number of training rooms increases, all the models display a phase transition where both prediction and path planning performance improve. When the number of training rooms is larger than 5k, in-context learning *emerges* for the transformer and `TDB`: both achieve almost perfect in-context accuracy on the *new* test rooms. In contrast, LSTM reaches lower in-context accuracy. In addition, in-context path planning emerges in the `TDB`: it can nearly perfectly solve the shortest paths problems, which both vanilla models struggle with. The in-context latent graphs reach low `NormGED` and are nearly isomorphic to the GT, and correctly model the room's dynamics—see Fig. 5[right]. $\texttt{TDB}(S=3, M=4)$ trained on 10k rooms has a disentanglement accuracy (see Appendix F) of $94.02\%(\pm 0.27\%)$: again, multiple discrete bottlenecks do not learn a disentangled representation. Finally, Appendix L.4 studies how spatial exposure to base targets in the training data drives in-context performance.

5. Conclusion

We propose `TDB`, a transformer variant that addresses the shortcomings of vanilla transformers for path planning. `TDB` (a) introduces discrete bottlenecks which compress the information necessary to predict the next observation given history then (b) builds interpretable cognitive maps from the active bottlenecks indices. On perceptually aliased POEs, `TDB` (a) retains the near-perfect prediction of transformers, (b) calls an external solver on its latent graph to solve path planning problems exponentially faster, (c) learns interpretable structure from text datasets, (d) exhibits emergent in-context prediction and path planning abilities.

Our approach has two main limitations. First, `TDB` only accepts categorical inputs. Second, though multiple discrete bottlenecks accelerate training, they do not learn a disentangled latent space. To address these points, we want to modify the `TDB` architecture (a) to accept high-dimensional continuous observations (images), and (b) so that different bottlenecks learn non-redundant representations. Furthermore, we would like to extend this work into a framework to build planning-compatible world models in rich environments. To do so, we want to learn disentangled latent dynamics in factored Markov decision processes by (a) extracting knowledge with transformers, (b) using multiple discrete bottlenecks to compress this knowledge and to generate latent nodes—or local latent graphs—and (c) learning factorized transition matrices over these latent nodes.

[5] As there are $30 \times 29 \times 28 \times 27 = 657,720$ possible rooms, we train on at most 1.52% of all the rooms.

[6] $\tilde{x}_n = 1$ may map to different x_n through the sequence.

Acknowledgments

We thank Kevin Murphy, Daan Wierstra and Théophane Weber for useful discussions during the preparation of this manuscript.

Impact Statement

This paper aims to address some of the inherent shortcomings of vanilla transformers for solving path planning and navigation tasks. Consequently, it is possible that a future generation of our proposed model is used to train agents that build a cognitive map of their environment, and that are able to solve real-time navigation tasks in the real world.

References

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Carbonneau, M.-A., Zaidi, J., Boilard, J., and Gagnon, G. Measuring disentanglement: A review of metrics. *arXiv preprint arXiv:2012.09276*, 2020.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Chrisman, L. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, volume 1992, pp. 183–188. Citeseer, 1992.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Conmy, A., Mavor-Parker, A. N., Lynch, A., Heimersheim, S., and Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *arXiv preprint arXiv:2304.14997*, 2023.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1, 2021.

Fraccaro, M., Rezende, D., Zwols, Y., Pritzel, A., Eslami, S. A., and Viola, F. Generative temporal models with spatial memory for partially observed environments. In *International conference on machine learning*, pp. 1549–1558. PMLR, 2018.

George, D., Rikhye, R. V., Gothoskar, N., Guntupalli, J. S., Dedieu, A., and Lázaro-Gredilla, M. Clone-structured graph representations enable flexible learning and vicarious evaluation of cognitive maps. *Nature communications*, 12(1):2392, 2021.

Ghahramani, Z. and Jordan, M. Factorial hidden markov models. *Advances in Neural Information Processing Systems*, 8, 1995.

Grill, J.-B., Strub, F., Altché, F., Tallec, C., Richemond, P., Buchatskaya, E., Doersch, C., Avila Pires, B., Guo, Z., Gheshlaghi Azar, M., et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

Guan, L., Valmeekam, K., Sreedharan, S., and Kambhampati, S. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *arXiv preprint arXiv:2305.14909*, 2023.

Guntupalli, J. S., Raju, R. V., Kushagra, S., Wendelken, C., Sawyer, D., Deshpande, I., Zhou, G., Lázaro-Gredilla, M., and George, D. Graph schemas as abstractions for transfer learning, inference, and planning. *arXiv preprint arXiv:2302.07350*, 2023.

Guo, Z., Thakoor, S., Pîslar, M., Avila Pires, B., Altché, F., Tallec, C., Saade, A., Calandriello, D., Grill, J.-B., Tang, Y., et al. Byol-explore: Exploration by bootstrapped prediction. *Advances in neural information processing systems*, 35:31855–31870, 2022.

Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Kamada, T., Kawai, S., et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

Karkus, P., Hsu, D., and Lee, W. S. Qmdp-net: Deep learning for planning under partial observability. *Advances in neural information processing systems*, 30, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lamb, A., Islam, R., Efroni, Y., Didolkar, A. R., Misra, D., Foster, D. J., Molu, L. P., Chari, R., Krishnamurthy, A., and Langford, J. Guaranteed discovery of control-endogenous latent states with multi-step inverse models. *Transactions on Machine Learning Research*, 2022.

Lieberum, T., Rahtz, M., Kramár, J., Irving, G., Shah, R., and Mikulik, V. Does circuit analysis interpretability scale? evidence from multiple choice capabilities in chinchilla. *arXiv preprint arXiv:2307.09458*, 2023.

Mentzer, F., Minnen, D., Agustsson, E., and Tschannen, M. Finite scalar quantization: Vq-vae made simple. *arXiv preprint arXiv:2309.15505*, 2023.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023.

Momennejad, I., Hasanbeig, H., Vieira, F., Sharma, H., Ness, R. O., Jojic, N., Palangi, H., and Larson, J. Evaluating cognitive maps and planning in large language models with cogeval. *arXiv preprint arXiv:2309.15129*, 2023.

Nanda, N., Chan, L., Liberum, T., Smith, J., and Steinhardt, J. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T., Mann, B., Askell, A., Bai, Y., Chen, A., et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Pallagani, V., Muppasani, B., Murugesan, K., Rossi, F., Horesh, L., Srivastava, B., Fabiano, F., and Loreggia, A. Plansformer: Generating symbolic plans using transformers. *arXiv preprint arXiv:2212.08681*, 2022.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pp. 8821–8831. PMLR, 2021.

Reed, S., Zolna, K., Parisotto, E., Colmenarejo, S. G., Novikov, A., Barth-Maron, G., Gimenez, M., Sulsky, Y., Kay, J., Springenberg, J. T., et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Sanfeliu, A. and Fu, K.-S. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (3):353–362, 1983.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Swaminathan, S., Dedieu, A., Raju, R. V., Shanahan, M., Lazaro-Gredilla, M., and George, D. Schema-learning and rebinding as mechanisms of in-context learning and emergence. *arXiv preprint arXiv:2307.01201*, 2023.

Valmeekam, K., Olmo, A., Sreedharan, S., and Kambhampati, S. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.

Van Den Oord, A., Vinyals, O., et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

Whittington, J. C., McCaffary, D., Bakermans, J. J., and Behrens, T. E. How to build a cognitive map. *Nature neuroscience*, 25(10):1257–1272, 2022.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080*, 2021.

A. Vanilla transformer architecture

Fig.6 presents the vanilla transformer detailed in Sec.3.2. In our numerical experiments, Sec.4, we train this model to minimize the autoregressive objective Equation (1).

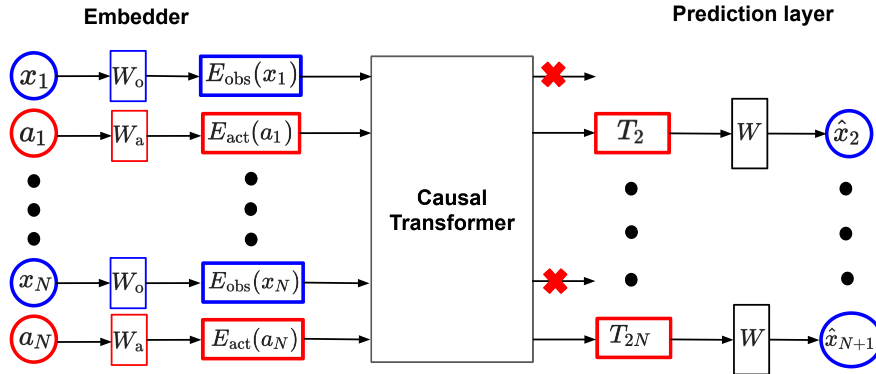

Figure 6. A vanilla transformer with causal mask takes as inputs the respective linear embeddings of the observations and actions. A linear layer on top of the action representations returns the next observation logits.

B. On the connection between TDB and CSCG

B.1. Computational and space complexity

A clone-structured causal graph (CSCG) (George et al., 2021) is a hidden Markov model variant used to model sequences of categorical observations. The latent space of a CSCG is partitioned, such that all the latent variables in one set of the partition deterministically emit the same observation. These latent variables are the *clones* of the observation. Because of its latent structure, CSCG models a transition between two observations as a transition between their respective clones—as opposed to a transition between all the latent variables.

We compare below the complexity between a vanilla CSCG and a vanilla self-attention layer when running inference on a sequence of length N.

Computation complexity: For a vanilla CSCG with M clones per observation, the computational cost for computing the forward and backward messages is $O(M^2N)$. In contrast, for a vanilla self-attention layer with a D-dimensional embedding, the forward pass cost is $O(N^2D)$. In this paper, we use $N = 400$, $D = 256$. CSCG is then more expensive when the number of clones per observation satisfies $M \gg \sqrt{ND} = 320$.

Space complexity: The memory cost of storing a vanilla CSCG with M clones per observation and E different observations is $O(M^2E^2)$. This large dense memory matrix can grow quickly as the number of observations increases, and prohibit vanilla CSCGs from scaling to very large latent space. In contrast, a vanilla self-attention layer with a D-dimensional embedding requires a $O(N^2 + ND)$ memory. The quadratic dependence on the sequence length is problematic for large corpora, but not for the problem we study here, where $N = 400$.

B.2. A transformer with clone-structured discrete bottleneck

We propose herein a variant of the `TDB` architecture inspired by the CSCG model, which injects a "clone-structured" inductive bias into its discrete bottlenecks. We consider the dictionary of latent codes introduced in Sec.3.3: $\mathcal{D} = (d_1, \dots, d_K)$, where $d_k \in \mathbb{R}^D$, $\forall k$. We now allocate a subset of the latent codes for each observation x in the vocabulary V. That is, we first define a partition of the indices $\{\mathcal{C}(x)\}_{x \in V}$ such that:

$$\bigcup_{x \in V} \mathcal{C}(x) = \{1, \dots, K\} \quad ; \quad \mathcal{C}(x) \cap \mathcal{C}(y) = \emptyset, \ \forall x \neq y \quad ; \quad \mathcal{C}(x) \neq \emptyset, \ \forall x.$$

Second, we generalize the operator ϕ in Equation (2) so that it now depends on both (a) the transformer output and (b) the observation itself. That is, we define:

$$\tilde{\phi}(y, x) = \underset{k \in \mathcal{C}(x)}{\operatorname{argmin}} \|y - d_k\|_2^2, \ \ \forall y \in \mathbb{R}^d, \ \forall x \in V \tag{6}$$

For the observation x_n which leads to the transformer output T_{2n-1}, the discrete quantizer now returns the latent code of $\mathcal{D}$ with index $\tilde{\phi}(T_{2n-1}, x_n)$. That is, each observation can now only activate a subset of the latent codes. We refer to this new quantizer module as *clone-structured discrete bottleneck*.

Computational gain: This architecture change decreases the computational cost of the discrete bottleneck by a linear term $O(E)$—which can be of interest when the number of observations E is large. However, the overall computational cost is still dominated by the quadratic cost of computing the attention matrices in the causal transformer.

Performance gain: Overall, we did not find any major benefit from replacing the discrete bottleneck of a `TDB` with its clone-structure variants. Our most promising result was that we were able to solve the 3D synthetic environments in Sec.4.3 using a single clone-structured discrete bottleneck—where a `TDB` needs more than one discrete bottleneck. However, a transformer with clone-structured bottlenecks would struggle to solve the in-context learning experiments in Sec.4.5. Indeed, a new test room with observations $5, 6, 7, 8$ would, by definition, use different latent codes than a training room with observations $1, 2, 3, 4$, which prevents the model from learning to share structure across rooms. This convinced us to not use the clone-structured discrete bottleneck in our experiments.

C. Clustering steps to build the transition graph

In this section, we detail the clustering steps that we apply to learn a cognitive map from the bottleneck indices in Sec.3.6.

Step 1. Cluster the bottlenecks indices with Hamming distance: This first clustering step only applies in the case where the `TDB` has multiple $M > 1$ discrete bottlenecks. In this case, `TDB` maps each observation x_n to a tuple of latent code indices:

$$s_n = \left(\phi^{(1)}(T_{2n-1}), \dots, \phi^{(M)}(T_{2n-1})\right) \in \{1, \dots, K\}^M.$$

Before building the cognitive map, we first collect in a set $\mathcal{J}$ all the unique tuples of latent indices appearing in the training set. For two tuples $s = (s^{(1)}, \dots, s^{(M)}) \in \mathcal{J}$ and $\ell = (\ell^{(1)}, \dots, \ell^{(M)}) \in \mathcal{J}$, we compute their Hamming distance:

$$H(s, \ell) = \frac{1}{M} \sum_{j=1}^{M} \mathbf{1}(s^{(j)} \neq \ell^{(j)}) \in [0, 1].$$

We introduce a distance threshold d_{Hamming}—which we set to 0.25 in Sec.4—and merge any pair of tuples such that $H(s, \ell) \leq d_{\text{Hamming}}$. This means that we will not distinguish ℓ and s when we build the count tensor C—see Sec.3.6. Consequently, these two tuples will be mapped to the same node on the cognitive map $\mathcal{G}$—and any node in $\mathcal{G}$ will be associated with a *collection of tuples* of bottleneck indices.

Intuitively, this means that when the entries of two tuples of bottleneck indices are "almost all the same", then we assume that their associated observations are at the same spatial position.

Step 2. Merge identical nodes: Second, we build the count tensor C (after this first clustering step), and threshold it with t_{ratio} to build an action-augmented transition graph $\mathcal{G}$—as detailed in Sec. 3.6. We only retain nodes that have at least two incoming and outgoing neighbors. Finally, our second clustering step merges every pair of nodes in $\mathcal{G}$ that are connected to the same set of neighbors.

Step 3. Map each discarded node to its closest retained node: Thresholding the count tensor $\mathcal{C}$ may discard some (tuple of) bottleneck indices with low empirical counts. In particular, as we show in Fig.10, Appendix I.3, many bottleneck indices that are active early in the sequences are discarded. This may be problematic for solving path planning problems, as it means that the corresponding observations cannot be mapped to nodes in $\mathcal{G}$. Indeed, when `TDB` activates a discarded (tuple of) bottleneck indices, it cannot locate itself in the learned cognitive map $\mathcal{G}$.

To remedy this, we divide the set of unique (tuples of) latent indices $\mathcal{J}$, between the elements that are discarded—i.e. not mapped to a node in $\mathcal{C}$—and retained—i.e. mapped to a node in $\mathcal{C}$. Our last clustering step maps any discarded (tuple of) latent indices $s_{\text{discarded}}$ to its closest retained latent index for the ℓ_1 distance, defined as:

$$\hat{s}_{\text{retained}} \in \underset{s_{\text{retained}}}{\operatorname{argmin}} \sum_{a=1}^{N_{\text{actions}}} \sum_{\ell=1}^{K} \left| p(\ell \mid s_{\text{retained}}, a) - p(\ell \mid s_{\text{discarded}}, a) \right|.$$

After this step, $s_{\text{discarded}}$ and $\hat{s}_{\text{retained}}$ are mapped to the same node in $\mathcal{C}$, and all the entries in $\mathcal{J}$ are retained.

D. Solving path planning problem at test time on partially observed environments

We detail herein how a vanilla sequence model and a TDB can solve the test shortest path problem presented in Sec.4.2.

Defining the shortest path problem: We consider a test sequence of observations $x = (x_1, \dots, x_N)$ and actions $a = (a_1, \dots, a_N)$ in a POE—the environment can either be the 2D aliased rooms of Sec.4.2 and Sec.4.5, the aliased cube of Sec.4.2, or the egocentric 3D synthetic environments of Sec.4.3. We also denote $\text{pos} = (\text{pos}_1, \dots, \text{pos}_N)$ the *unobserved* sequence of room spatial positions associated with each observation. Note that, because of aliasing, we cannot deterministically infer pos_n from x_n.

For the 2D aliased rooms (Sec.4.2), the spatial position of an observation is its 2D coordinates. For the egocentric 3D synthetic environments (Sec.4.3), the spatial position of an observation is a 3D vector: the first two entries correspond to the 2D coordinates of the observation, while the last entry corresponds to the agent's heading direction, which is one of $\{0°, 90°, 180°, 270°\}$

Let C be a context length—we set $C = 50$ or $C = 100$ in practice. The path planning problem we consider in Sec.4 consists of finding the shortest path, i.e., the shortest sequence of actions, which leads from the room position pos_C (associated with the observation x_C) to the room position pos_{N-C} (associated with the observation x_{N-C}). Naturally, the sequence $(a_C, \dots, a_{N-C-1})$ is a path of length $N - 2C$ from pos_C to pos_{N-C}, which we refer to as the *fallback path*. In addition, for evaluating a model's performance, we use an *optimal shortest path*, which solves the shortest path problem in the ground truth room—note that the agent does not know the ground truth layout. While the optimal shortest path is not unique, all the optimal shortest paths have the same length.

Fig.1 in the main text illustrates this path planning problem on a 2D aliased room—we use $N = 40, C = 5$ for the sake of visualization.

We detail below our proposed procedure for solving this shortest path problem with a TDB; and with a vanilla transformer or a vanilla LSTM. As mentioned in the main text, a vanilla sequence model finds this challenging because (a) it can only perform forward rollouts—without being able to collapse redundant visitations of the same spatial position—and (b) it cannot tell whether it has reached its destination. In contrast, a TDB solves this problem by querying its learned latent graph.

We are interested in measuring (a) whether each model can derive a path that improves over the fallback path and (b) when a model derives such a better path, how much worse it is compared to the optimal shortest path.

Deriving shortest paths with TDB: When TDB is evaluated on the test sequences x and a, it maps each observation x_n to either (a) a single latent code index $s_n = \phi(T_{2n-1}) \in \{1, \dots, K\}$ when we have a single discrete bottleneck or (b) a tuple of latent codes indices $s_n = \left(\phi^{(1)}(T_{2n-1}), \dots, \phi^{(M)}(T_{2n-1})\right)$ when we have multiple discrete bottlenecks.

As a reminder, each node in the learned cognitive map $\mathcal{G}$ is associated with a collection of (tuples of) bottleneck indices—see Appendix C. However, the (tuples of) bottleneck indices s_C and s_{N-C} may not be associated with a node in $\mathcal{G}$. Consequently, we find two tuples of bottleneck indices $\tilde{s}_C$ and $\tilde{s}_{N-C}$ (a) with lowest Hamming distance from s_C and s_{N-C} and (b) which are associated with two nodes n_C and n_{N-C} in $\mathcal{G}$. Finally, we call the external `networkx.shortest_path` solver to find a shortest path between the nodes n_C and n_{N-C}.

Deriving shortest paths from rollouts with vanilla sequence models: A naive option that finds the shortest path between pos_C and pos_{N-C} with a vanilla sequence model consists in exploring all the possible sequences of observations and actions. For a number of evaluations exponential in the length of the optimal shortest path, this approach is guaranteed to find an optimal shortest path.

We propose herein an alternative approach, which (a) is more computationally efficient, (b) is not guaranteed to find the optimal shortest path, and (c) in practice, often improves over the fallback path. Beforehand, let us define the subsequences of observations context $x_{\text{context}} = (x_1, \dots, x_C)$ and actions context $a_{\text{context}} = (a_1, \dots, a_{C-1})$. We also define the subsequences of observations tail $x_{\text{tail}} = (x_{N-C+1}, \dots, x_N)$ and actions tail $a_{\text{tail}} = (a_{N-C}, \dots, a_{N-1})$.

First, we generate $N_{\text{random walks}}$ random sequences of actions, each of length $N - 2C$: we denote $a^{(i)} = (a^{(i)}_C, \dots, a^{(i)}_{N-C-1})$ the ith random sequence—we initialize the indices at C. In Sec.4.3, we set $N_{\text{random walks}} = 10$.

Second, we augment each sequence of actions (of length $N - 2C$) into two sequences of observations and actions, of respective lengths $N - 2C$ and $N - 2C - 1$, as follows. The first C observations and $C - 1$ actions are the shared context

x_{context} and a_{context}. The next $N-2C$ actions correspond to $a^{(i)}$. The corresponding sequence of observations corresponds to the sequence model's autoregressive predictions. That is, for $n = C, \ldots, N-C-1$, we iteratively define $\hat{x}_{n+1}^{(i)}$ as:

$$\hat{x}_{n+1}^{(i)} \in \operatorname*{argmax}_x p \left(x \;\middle|\; \underbrace{x_1,\ a_1, \ldots, a_{C-1},\ x_C}_{\text{shared context}},\ a_C^{(i)},\ \hat{x}_{C+1}^{(i)},\ a_{C+1}^{(i)}, \ldots, \hat{x}_n^{(i)},\ a_n \right)$$

For each generated random walk, we look for all the observations equal to x_{N-C}, and define the set of *candidates*:

$$\mathcal{C} = \left\{ (i,n) : n \geq C \text{ and } \hat{x}_n^{(i)} = x_{N-C} \right\}.$$

Estimating whether the target is reached via tail evaluation: As we mentioned in the main text, because of aliasing, the vanilla sequence model cannot know whether it has reached the target position when it observes a candidate. That is, it does not know whether the spatial position associated with the candidate is equal to pos_{N-C}.

We could return all the candidates and evaluate them using the ground truth map. Instead, we propose herein to estimate whether a candidate (i,n) is at the target position pos_{N-C} by evaluating it—with its history—on the tail subsequences of observations x_{tail} and actions a_{tail}. That is, for a candidate (i,n) we define, for $k = 1, \ldots, C$:

$$\hat{x}_{\text{tail},k}^{(i)} \in \operatorname*{argmax}_x p \left(x \;\middle|\; \underbrace{x_1,\ a_1, \ldots, a_{C-1},\ x_C}_{\text{shared context}},\ \underbrace{a_C^{(i)},\ \hat{x}_{C+1}^{(i)},\ a_{C+1}^{(i)}, \ldots, \hat{x}_n^{(i)} = x_{N-C}}_{\text{autoregressive random walk}},\ a_{N-C}, x_{N-C+1}, \ldots, a_{N-C+k-1} \right)$$

We estimate that a candidate is at the target position pos_{N-C} when $(\hat{x}_{\text{tail},1}^{(i)}, \ldots, \hat{x}_{\text{tail},C}^{(i)}) = x_{\text{tail}}$. Note that, because of aliasing, even when the tail evaluation succeeds, the candidate may not be at the target position.

Evaluating valid paths: For the `TDB`, we return a single shortest path proposal, which is estimated by the external solver. For the transformer and the LSTM, we return all the path proposals that are estimated to be at the target position.

For each model, we first evaluate whether the proposed paths are *valid*. That is, we test whether the proposed sequence of actions correctly leads from pos_C to pos_{N-C} in the ground truth environment. For a vanilla sequence model, when multiple returned paths are valid, we only retain the shortest one. We then compute two metrics **(a)** a binary indicator of whether the method has found a valid path and **(b)** if (a) is correct, the ratio between the length of the valid found path and the optimal path length. Finally, `ImpFallback` averages (a) while `RatioSP` averages (b) over 200 shortest paths problems.

Tail evaluation improves the quality of the paths returned: For a transformer trained on 2D aliased rooms in Sec.4.2, if we return all the candidates, then the ratio of *valid* paths among the paths returned is $0.78\%(\pm 0.03\%)$. Indeed, most of the candidates are not at the target spatial position. When we estimate whether we are at the target position via tail evaluation, this ratio goes up to $38.76\%(\pm 1.17\%)$—prediction errors prevent it from being at 100%. Similarly, on the 3D synthetic environments of Sec.4.3, a transformer without tail evaluation returns $1.75\%(\pm 0.20\%)$ of valid paths while a transformer with tail evaluation returns $25.78\%(\pm 1.17\%)$. Finally, on the ICL experiments of Sec.4.5, for a transformer trained on $10k$ rooms, this same ratio goes from $2.58\%(\pm 0.05\%)$ without tail evaluation up to $51.10\%(\pm 1.81\%)$ with it. Overall, tail evaluation drastically improves the quality of the paths returned by sequence models.

E. Multiple discrete bottlenecks accelerate training

Tables 4, 5 and 6 below illustrate that multiple discrete bottlenecks accelerate training.

2D aliased rooms: For each TDB trained on the 2D aliased rooms and reported in Sec.4.2, we compute, for every 1000 training steps, its training accuracy on the entire training set. We define a new metric, TrainingAbove98, which reports the first evaluation when this training accuracy is higher than 98%. Table 4 averages this metric over the 10 experiments run in Sec.4.2, for various TDB models.

Method	TrainingAbove98 $\downarrow$
TDB$(S=1, M=1)$	9400 (290)
TDB$(S=1, M=4)$	4300 (145)
TDB$(S=1, \text{enc}, M=1)$	9300 (425)
TDB$(S=1, \text{enc}, M=4)$	4500 (158)
TDB$(S=3, M=1)$	8300 (284)
TDB$(S=3, M=4)$	4000 (0)

Table 4. On 2D aliased rooms, training a TDB with $M=4$ discrete bottlenecks converges faster than training its counterpart with a single discrete bottleneck.

3D synthetic environments: As discussed in Sec.4.2, we need $M=4$ discrete bottlenecks for the TDB training to converge in 3D synthetic environments. Table 5 illustrates this by reporting TrainingAfter1k—the training accuracy after 1000 training steps—for the different models reported in Table 2.

Method	TrainingAfter1k (%) $\uparrow$
TDB$(S=1, M=1)$	41.40 (0.09)
TDB$(S=1, M=4)$	99.72 (0.03)
TDB$(S=1, \text{enc}, M=1)$	40.81 (0.93)
TDB$(S=1, \text{enc}, M=4)$	99.71 (0.04)
TDB$(S=3, M=1)$	28.05 (1.88)
TDB$(S=3, M=4)$	99.63 (0.06)

Table 5. On the 3D synthetic environments, training a TDB with $M=4$ discrete bottlenecks converges faster than training its counterpart with a single discrete bottleneck.

GINC text dataset: Finally, on the GINC dataset, we also evaluate the training accuracy of each model reported in Sec.4.5 at every 1000 training steps. Here, TrainingAbove98 reports the first iteration when the training accuracy becomes higher than 98% of the highest value it reaches during training.

Method	TrainingAfter1k (%) $\uparrow$
TDB$(S=1, M=1)$	16000
TDB$(S=1, M=4)$	6000
TDB$(S=3, M=1)$	13000
TDB$(S=3, M=4)$	5000
TDB$(S=5, M=1)$	11000
TDB$(S=5, M=4)$	5000

Table 6. On the GINC dataset, training a TDB with $M=4$ discrete bottlenecks converges faster than training its counterpart with a single discrete bottleneck.

F. Do multiple discrete bottlenecks learn a disentangled representation?

Predicting a discrete bottleneck index given the other indices: We propose herein a metric to test whether multiple discrete bottlenecks learn a disentangled representation of the observations.

If the discrete bottlenecks learn of a disentangled representation, then (a) they have to be independent (b) each one has to explain a distinct factor of the data (Carbonneau et al., 2020). Given a TDB with M discrete bottlenecks, we propose to estimate (a), that is, whether the bottlenecks are independent. To do so, we train a logistic regression estimator to predict each bottleneck index given the other $M-1$ bottleneck indices.

Given a sequence of observations $(x_1, \dots, x_N)$, a TDB with M discrete bottlenecks maps each observation x_n to the latent codes $\left(s_n^{(1)}, \dots, s_n^{(M)}\right)$ where $s_n^{(i)} = \phi^{(i)}(T_{2n-1}) \in \{1, \dots, K\}$—see Sec.3.5.

We denote $e_n^{(i)} = \left(\mathbf{1}(s_n^{(i)} = k)\right)_{1 \le k \le K}$ the one-hot encoding of $s_n^{(i)}$.

Let us now assume that we want to predict the Mth bottleneck index given the first $M-1$ bottleneck indices.

We note $z = \left(e_n^{(1)}, \dots, e_n^{(M-1)}\right) \in \{0,1\}^{(M-1)K}$ and $y = e_n^{(M)} \in \{0,1\}^K$, where we drop the dependency over n. We also introduce a matrix $W \in \mathbb{R}^{K \times (M-1)K}$. The logistic regression objective at timestep n is:

$$\mathcal{L}^{(M)}_{\text{disentanglement}}(n) = -\sum_{i=1}^{K} y_i \, \log\left(\frac{\exp(W_i^T z)}{\sum_{j=1}^{K} \exp(W_j^T z)}\right)$$

We also define $\mathcal{L}^{(M)}_{\text{disentanglement}} = \sum_{n=1}^{N} \mathcal{L}^{(M)}_{\text{disentanglement}}(n)$. Note that we do not use an intercept term or a regularization term.

Training: Given a training and test set from Sec.4, each consisting of N_{seq} test sequences of length N, we first compute all the active bottleneck indices, which gives us two tensors, $\mathcal{T}_{\text{train}}$ and $\mathcal{T}_{\text{test}}$, each of dimensions $N_{\text{seq}} \times N \times M$.

We train M logistic regression estimators on $\mathcal{T}_{\text{train}}$: the ith estimator is trained to minimize $\mathcal{L}^{(i)}_{\text{disentanglement}}$, that is, to predict the ith bottleneck index given the other $M-1$. For training, we use Adam (Kingma & Ba, 2014) for $5,000$ iterations, with a learning rate of 0.001 and a batch size of 32.

Disentanglement metric: Our proposed disentanglement metric is the averaged test accuracy of the M logistic regression estimators on the test tensor $\mathcal{T}_{\text{test}}$.

Results: In Sec.4, we use $K = 1000$ latent codes for each discrete bottleneck: if the bottleneck representations are independent, the test disentanglement accuracy would be 0.10%. However, all the test disentanglement accuracies reported in the main paper, are above 85%, which shows that multiple discrete bottlenecks learn highly redundant representations.

G. TDB with single-step objective merges nodes with identical neighbors

In Sec.3.7, we make the argument that a `TDB` only trained to predict the next observation is not encouraged to disambiguate observations with identical neighbors in the ground truth (GT) room and, as a result, may merge their representations. We use this same argument in Sec.3.4 to justify the need for augmenting the loss via either (a) a multi-step objective or (b) next encoding prediction.

Fig.7 demonstrates the argument. We consider a 2D aliased GT room from Sec. 4.2 with $O = 12$ unique observations—the numerical results are presented in Table 8, Appendix I.2. Similar to Fig.1, the GT room contains a 4×4 patch with black borders repeated twice. Each 4×4 patch contains an inner 2×2 patch with fuchsia borders. For each node inside this inner patch, all four actions in the GT room (move up, down, left, and right) lead to the same neighbors, regardless of whether the node in the top-left 2×2 patch or the bottom-right one.

Fig.7[right] shows the latent graph learned by a $\texttt{TDB}(S = 1, M = 4)$ with single step prediction and four discrete bottlenecks. Because this model is only trained to predict the next observation given the history, it learns the same representation for each pair of nodes with same location inside the inner 2×2 patch. This results in the presence of four *merged nodes* in the latent graph—colored in fuchsia. When a merged node is active, the agent can (a) perfectly predict the next observation but (b) it cannot know which one of the two possible 2×2 patches it is in. In addition, the merged nodes introduce *unrealistic shortcuts* in the learned latent graph: nodes that are far apart in the GT room are close in the latent graph. Because the learned cognitive map does not accurately model the ground truth dynamics, the external solver will propose shortest paths that are invalid in the GT room. As a consequence, as indicated in Table 8, Appendix I.2, $\texttt{TDB}(S = 1, M = 4)$ can only solve $59.37\%(\pm 3.93\%)$ of the path planning problems.

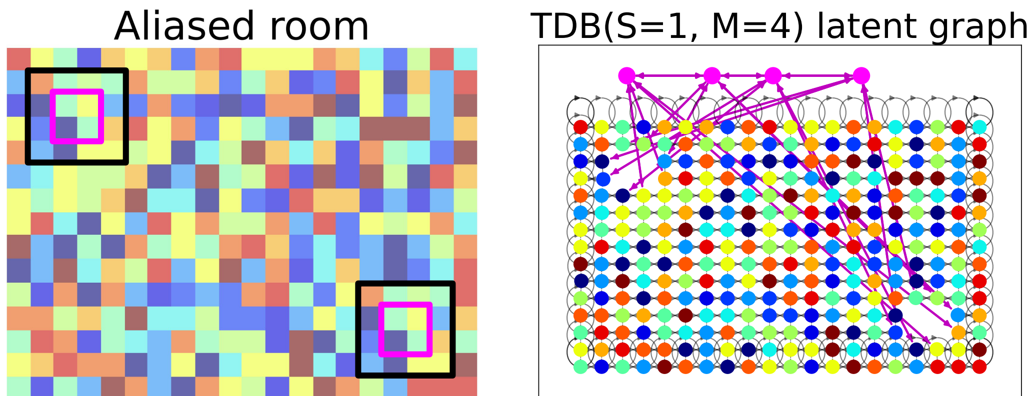

Figure 7. [Left] A 2D aliased room with $O = 12$ unique observations: for each observation inside the 2×2 patches with fuchsia boundaries, all four actions (move up, down, left, right) lead to the same neighbors, regardless of whether the node is in the top-left 2×2 patch or the bottom-right one. [Right] The latent graph learned by a `TDB` with single step prediction fails at learning the ground truth dynamics. `TDB` learns the same representation for the two elements of each pair. This results in four merged nodes—colored in fuchsia—which are active at two spatial locations. While these merged nodes do not affect test prediction performance, they introduce unrealistic shortcuts in the latent graph, which fool the external planner.

For aliased rooms with 4 colors: When $O = 4$, a higher frequency of observations have identical neighbors. The learned latent graph merges these observations, which introduces a larger number of unrealistic shortcuts. This explains the poor performance of $\texttt{TDB}(S = 1, M = 1)$ in Table 1, which only improves $6.61\%(\pm 0.53\%)$ of the time over the fallback path. Because of this inherent failure, $\texttt{TDB}(S = 1, M = 4)$ similarly only improves over the fallback path $6.21\%(\pm 0.62\%)$ of the time. In fact, `TDB` is only able to solve the easiest path planning problems, for which the optimal shortest path consists of only a small number of actions. Indeed, the average length of the optimal shortest paths solved by $\texttt{TDB}(S = 1, M = 4)$ is 2.15 (± 0.07) while the average length of the optimal shortest paths over all the path planning problems is 10.80 (± 0.08).

For 3D synthetic environments: Fig.8[middle] shows that, for a 3D environment from Section 4.3, the cognitive map learned with a $\texttt{TDB}(S=1, M=4)$ merges three pairs of nodes. These merged nodes introduce unrealistic shortcuts on the learned latent graph: as a result, $\texttt{TDB}(S=1, M=4)$ fails at path planning and only improves $65.30\%(\pm 4.24 \pm \%)$ of the time over the fallback path.

Fig.8[right] shows the latent graph learned when the objective is augmented to predict the next three steps, and highlights the three pairs of nodes merged in the middle image. Note that each node is mapped to the observation with higher empirical frequency when this node is active. As before, we see that the merged nodes have identical neighbors[7].

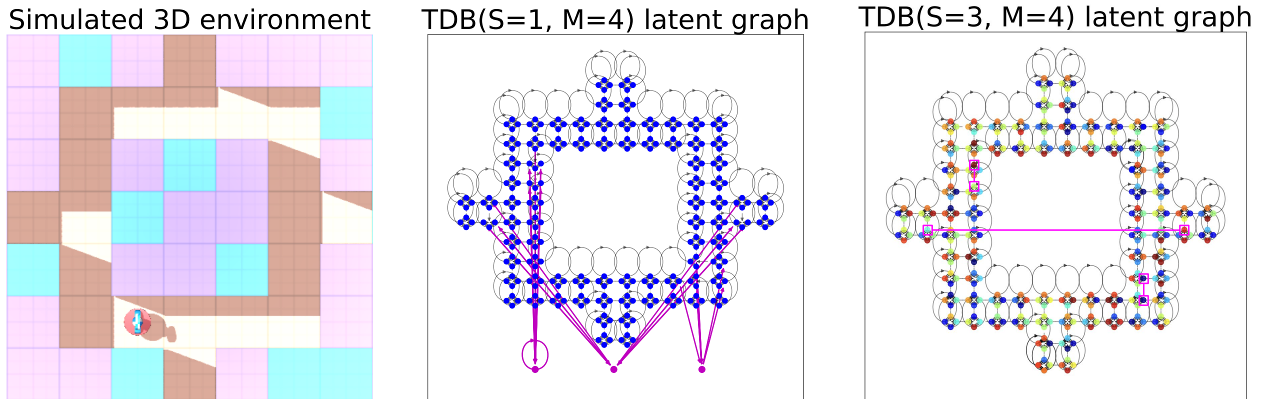

Figure 8. [Left] Top-down view of a 3D simulated rooms used in Sec.4.3. [Middle] Latent graph learned with a $\texttt{TDB}(S=1, M=4)$ with single-step objective: each pair of observations with identical neighbors are represented with the same node, which introduces unrealistic shortcuts. [Right] Color-coded latent graph learned with a multi-step objective.

[7] Because there are 128 clusters, the reader cannot precisely infer the observation from the colors: we looked at the graph indices to make sure that the neighbors of the merged nodes are identical.

H. Accelerating the graph edit distance

The graph edit distance (GED) (Sanfeliu & Fu, 1983) between two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a graph similarity measure defined as minimum cost of edit path—i.e. sequence of node and edge edit operations—to transform the graph $\mathcal{G}_1$ into a graph isomorphic to $\mathcal{G}_2$.

Computing the exact GED is NP-complete. We therefore rely on the networkx.optimize_graph_edit_distance (Hagberg et al., 2008) function to return a sequence of approximations. This method accepts as argument a function to indicate which nodes (resp. edges) of $\mathcal{G}_1$ and $\mathcal{G}_2$ should be considered equal during matching.

To accelerate GED, we say that a node n_1 of $\mathcal{G}_1$ and a node n_2 of $\mathcal{G}_2$ have to be considered equal — and we note $n_1 \approx n_2$ if the ground truth spatial positions with higher empirical frequency when n_1 (resp. n_2) is active are the same. Similarly, we say that an edge of $\mathcal{G}_1$ connecting the nodes (n_1^s, n_1^t) and an edge of $\mathcal{G}_2$ connecting the nodes (n_2^s, n_2^t) are considered to be equal if either (a) $n_1^s \approx n_2^s$ and $n_1^t \approx n_2^t$ or (b) $n_1^s \approx n_2^t$ and $n_1^t \approx n_2^s$.

We use a timeout of 900s to compute GED in all the experiments but the ICL one, for which we use 20s (due to the large number of normalized graph edit distances being computed).

I. Additional materials for aliased cube and for 2D aliased room cubes

I.1. Table of results for non-Euclidean aliased cube

Table 7 reports the prediction and path planning metrics for an aliased cube with edge size 6, similar to Fig. 3[bottom-left]. All the settings and parameters are the same as in Section 4.2. We average the results over 10 experiments: each experiment considers a different cube.

Method	TestAccu (%) ↑	ImpFallback (%) ↑	RatioSP ↓	NormGED ↓
Vanilla transformer	**99.76** (0.00)	43.82 (1.17)	22.74 (0.84)	—
Vanilla LSTM	**99.76** (0.00)	43.82 (1.17)	22.74 (0.84)	—
TDB($S=1, \text{enc}, M=1$)	99.36 (0.06)	99.07 (0.88)	**1.02** (0.00)	**0.334** (0.016)
TDB($S=3, M=1$)	99.76 (0.00)	**99.05** (0.23)	1.03 (0.00)	0.304 (0.011)

Table 7. Results averaged over 10 aliased cubes of edge size 6. Our TDB with either multi-step objective or next encoding prediction (a) retains the nearly perfect test accuracy of vanilla sequence models (b) consistently solves the shortest paths problems when paired with an external solver—while both transformer or LSTM catastrophically fail—(c) learns cognitive maps almost isomorphic to the ground truth.

I.2. Table of results for 12 unique observations

Table 8 reports the prediction and path planning metrics for an aliased room with $O = 12$ unique observations and global aliasing—as a 4×4 patch is repeated twice. All the settings and parameters are the same as in Section 4.2.

First, during training, a TDB with a single discrete bottleneck either (a) is stuck in a local optimum or (b) converges after a long time. As a consequence, contrary to Table 8 which considers $O = 4$, all the TDBs with a single bottleneck are here outperformed by their counterparts with four bottlenecks. TDBs with a single bottleneck also learn worse cognitive maps and solve a lower frequency of path planning problems.

Second, regardless of the number of bottleneck used, a TDB with single-step objective cannot disambiguate the global aliasing and introduces unrealistic shortcuts—see Appendix G—which result in a drop in planning metrics.

In contrast, TDBs with $M = 4$ bottlenecks and either multi-step or next encoding objective (a) achieves nearly perfect prediction accuracy, (b) are able to almost consistently find the optimal shortest path, and (c) learn cognitive maps that recover the GT dynamics.

Method	TestAccu (%) ↑	ImpFallback (%) ↑	RatioSP ↓	NormGED ↓
Vanilla transformer	**99.75** (0.00)	29.89 (0.94)	17.45 (0.57)	—
Vanilla LSTM	**99.75** (0.00)	29.89 (0.94)	17.45 (0.57)	—
TDB($S=1, M=1$)	89.46 (2.48)	17.31 (4.52)	**1.00** (0.00)	0.265 (0.053)
TDB($S=1, M=4$)	**99.75** (0.00)	59.37 (3.93)	**1.00** (0.00)	**0.033** (0.002)
TDB($S=1, \text{enc}, M=1$)	92.57 (3.04)	58.21 (10.98)	**1.00** (0.00)	0.145 (0.038)
TDB($S=1, \text{enc}, M=4$)	99.71 (0.00)	96.58 (2.13)	1.01 (0.00)	0.157 (0.016)
TDB($S=3, M=1$)	99.33 (0.01)	97.09 (2.06)	**1.00** (0.00)	0.157 (0.010)
TDB($S=3, M=4$)	99.73 (0.01)	**99.75** (0.14)	1.01 (0.00)	0.160 (0.017)

Table 8. Prediction and path planning metrics averaged over 10 aliased 15×20 rooms with $O = 12$ observations. Training a TDB single bottleneck ($M = 1$) is slow and sometimes does not converge. A TDB with single-step objective cannot disambiguate global aliasing, fails at recovering the ground truth dynamics. TDB with multiple discrete bottlenecks ($M = 4$) outperform their single bottleneck counterpart ($M = 1$) for both prediction and path planning performance.

I.3. The agent can locate itself after a short context

Test accuracy by timestep: Fig.9 shows the average prediction accuracy as a function of the observation index for our best $\texttt{TDB}(S = 3, M = 1)$, reported in Sec.4.2 for the 2D aliased room with $O = 4$ observations.

Because of aliasing, at the start of a test random walk, the model cannot know its location in the 2D room: the prediction accuracy for the first timesteps is low. The agent then quickly locates itself and test accuracy rapidly increases: it is above 99.50% after only 30 observations.

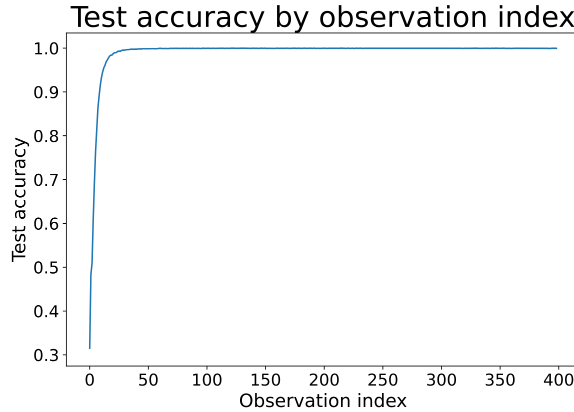

Figure 9. After a short context of 30 observations, $\texttt{TDB}(S = 3, M = 4)$ predicts the next observations 99.50% of the time.

Some latent codes model early uncertainty while others represent late confidence: Fig.10 displays, for each latent code active on the test data of a $\texttt{TDB}(S = 3, M = 1)$ (a) the average timestep at which this latent code is active—on the horizontal axis—and (b) its number of appearances on the entire test data—on the vertical axis. In addition, each bottleneck is colored in green (resp. red) to indicate that it is retained (resp. discarded) when we threshold the empirical count tensor C to build the cognitive map—as detailed in Sec.3.6.

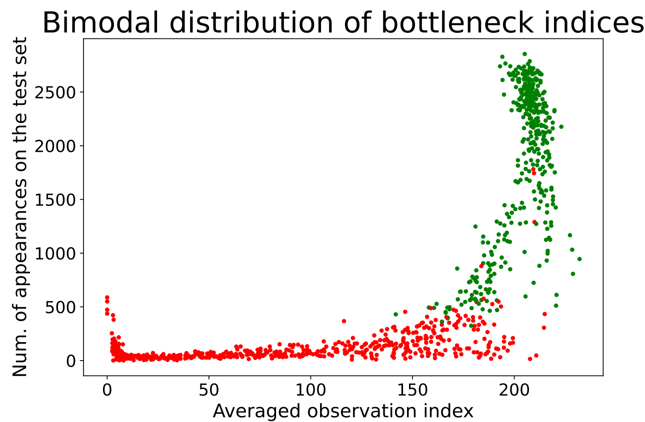

Figure 10. A first cluster of latent indices, in red, has low appearance counts and models the early agent's uncertainty in the test sequences. A second cluster, in green, has high appearance counts and represents the agent's confidence when it knows its location.

The figure displays two clusters, which highlight the two roles played by the latent codes. The first cluster represents a large number of codes with low appearance counts, which appear early in the test sequences. These codes model the agent's uncertainty when it does not know its location.

The latent codes in the second cluster have a high number of appearances and appear later in the test sequences. They represent the agent's high confidence when it knows its locations. In addition, because we use a high count threshold

$t_{\text{ratio}} = 0.1$ (see Sec.3.6) to build the cognitive map, only the transitions between the latent codes of this second cluster are used to build the graph. As a result, a vast majority of the nodes of the second (resp. first) cluster are green (resp. red). Consequently, our third clustering step, detailed in Appendix C, maps the discarded (red) latent indices to their closest retained (green) latent index.

The agent can "teleportate": We finally study how $\texttt{TDB}(S = 3, M = 1)$ moves in the learned latent graph. In particular, we say that the agent "teleportates" when the node of the transition graph where it estimates its location at the nth timestep is not a neighbor of its estimated position at the $n-1$th timestep. That is, teleportation means that the shortest path between two consecutive nodes is larger than one.

Fig.11 looks at the average number of teleportation as a function of the observation index. Around the 20th observation, the agent still teleportates 5% of the time. As the agent becomes confident about its location, the teleportation rate drops below 1% after 100 observations and stays around 0.75%.

Interestingly, Fig.11[right] shows a subsequence of 10 observations $(x_{10}, \ldots, x_{19})$ on a test random walks, with perfect next observation accuracy. Nonetheless, the agent still "teleportates" five times out of ten.

Here, teleportation is an artifact of the clustering procedures that we use when we build the cognitive map—see Appendix. C. For instance, our third clustering step may incorrectly map some discarded bottleneck indices to their closest bottleneck index. As a consequence, when the agent activates a discarded bottleneck index, it may be mapped to an invalid room location. If we use a lower threshold t_{ratio}, then teleportations should disappear but the learned latent graph would not accurately model the environment's dynamics.

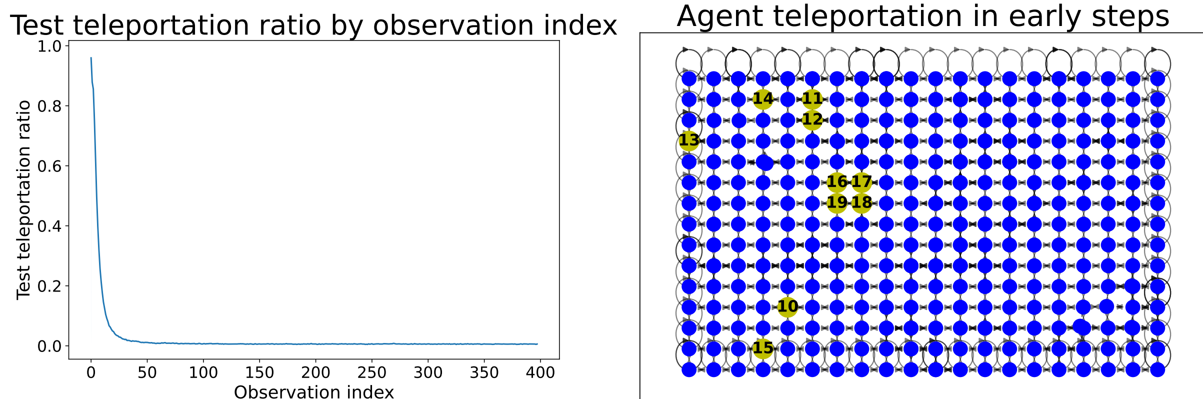

Figure 11. [Left] The agent frequently teleportates, i.e., moves between nodes that are not neighbors early in the test sequences. Teleportation rate drops below 5% after 20 observations, and below 1% after 100 observations. [Right] Despite perfectly predicting the next observation between indices 10 and 20, the agent also teleportates five steps out of ten. Here, teleportation is explained by incorrect clustering when we build the cognitive map.

J. Additional materials for simulated 3D environment

Latent graphs visualizations: Second, Fig.12 presents the isometric views of four 3D simulated rooms, as well as the respective cognitive maps learned with a TDB($S = 3, M = 4$).

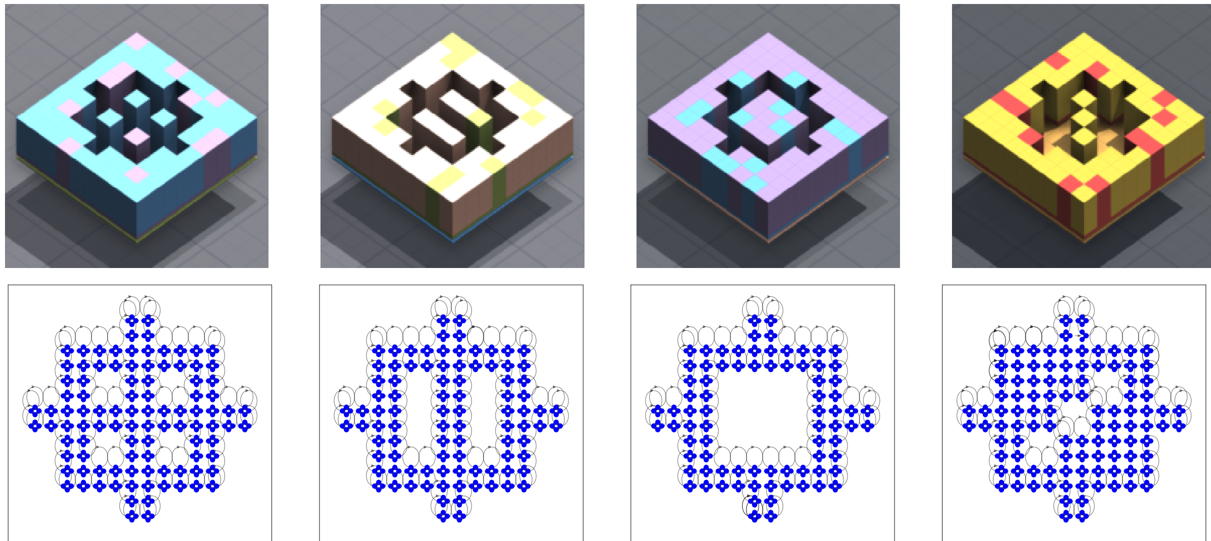

Figure 12. [Top] Isometric views of four 3D simulated rooms used in Sec.4.3. [Bottom] Corresponding cognitive maps learned with a TDB($S = 3, M = 4$): each location in the room is represented by four nodes in the latent graph, corresponding to the four possible heading directions of the agent.

K. Additional materials for GINC dataset

Table 9 presents numerical results associated with Fig.4[left].

Context length	Num. of examples	LSTM	Transformer	TDB($S=3, M=4$)	TDB($S=5, M=1$)
3	0	0.426 (0.01)	**0.502** (0.01)	0.426 (0.01)	0.284 (0.009)
	1	0.369 (0.01)	**0.488** (0.01)	0.464 (0.01)	0.369 (0.01)
	2	0.387 (0.01)	0.538 (0.01)	**0.554** (0.01)	0.443 (0.01)
	4	0.394 (0.01)	0.553 (0.01)	**0.588** (0.01)	0.526 (0.01)
	8	0.386 (0.01)	0.595 (0.01)	**0.618** (0.01)	0.549 (0.01)
	16	0.390 (0.01)	0.631 (0.01)	**0.655** (0.01)	0.574 (0.01)
	32	0.380 (0.01)	0.633 (0.01)	**0.670** (0.009)	0.618 (0.01)
	64	0.377 (0.01)	0.651 (0.01)	**0.676** (0.009)	0.646 (0.01)
5	0	0.692 (0.009)	**0.805** (0.008)	0.777 (0.008)	0.656 (0.01)
	1	0.742 (0.009)	0.822 (0.008)	**0.853** (0.007)	0.768 (0.008)
	2	0.754 (0.009)	0.831 (0.008)	**0.857** (0.007)	0.802 (0.008)
	4	0.749 (0.009)	0.841 (0.007)	**0.905** (0.006)	0.872 (0.007)
	8	0.761 (0.009)	0.847 (0.007)	**0.901** (0.006)	0.882 (0.006)
	16	0.756 (0.009)	0.871 (0.007)	**0.922** (0.005)	0.908 (0.006)
	32	0.767 (0.008)	0.871 (0.007)	**0.926** (0.005)	0.915 (0.006)
	64	0.758 (0.009)	0.88 (0.006)	**0.924** (0.005)	0.917 (0.006)
8	0	0.849 (0.007)	0.881 (0.006)	**0.895** (0.006)	0.816 (0.008)
	1	0.869 (0.007)	0.888 (0.006)	**0.930** (0.005)	0.899 (0.006)
	2	0.869 (0.007)	0.886 (0.006)	**0.930** (0.005)	0.917 (0.006)
	4	0.877 (0.007)	0.897 (0.006)	0.933 (0.005)	**0.936** (0.005)
	8	0.873 (0.007)	0.897 (0.006)	0.935 (0.005)	**0.940** (0.005)
	16	0.875 (0.007)	0.916 (0.006)	0.942 (0.005)	**0.950** (0.004)
	32	0.862 (0.007)	0.909 (0.006)	0.948 (0.004)	**0.956** (0.004)
	64	0.883 (0.006)	0.920 (0.005)	0.952 (0.004)	**0.955** (0.004)
10	0	0.878 (0.007)	0.888 (0.006)	**0.915** (0.006)	0.858 (0.007)
	1	0.889 (0.006)	0.898 (0.006)	**0.930** (0.005)	0.926 (0.005)
	2	0.878 (0.007)	0.901 (0.006)	**0.938** (0.005)	0.935 (0.005)
	4	0.902 (0.006)	0.904 (0.006)	0.936 (0.005)	**0.945** (0.005)
	8	0.881 (0.006)	0.906 (0.006)	0.947 (0.004)	**0.956** (0.004)
	16	0.882 (0.006)	0.908 (0.006)	0.934 (0.005)	**0.947** (0.004)
	32	0.889 (0.006)	0.918 (0.005)	0.946 (0.005)	**0.960** (0.004)
	64	0.89 (0.006)	0.911 (0.006)	0.941 (0.005)	**0.956** (0.004)

Table 9. In-context accuracy for vanilla sequence models and `TDB`s, for each each pair (k, n) of context length k and number of examples n of the GINC test set. Our proposed `TDB`s consistently reach the highest in-context accuracies.

L. Additional materials for in-context learning

L.1. Defining the in-context learning problem

Preserving the room partition: We consider a first 2D aliased room of size 6×8 with $O = 4$ observations, $c_1, \dots c_O$ such that $c_i \in \{1, \dots, 30\}$. We represent this room by a matrix $R \in \{c_1, \dots c_O\}^{6 \times 8}$, where, for $x \in \{1, \dots, 6\}$ and $y \in \{1, \dots, 8\}$, R_{xy} is the room observation at the spatial position (x, y).

We define the *room partition* induced by the room colors as the partition $\mathcal{S}_1, \dots, \mathcal{S}_O$ of the 2D room spatial positions such that, for each element $\mathcal{S}_i$ of the partition, all the spatial positions in $\mathcal{S}_i$ have the same observation.

Formally, the room partition is defined as

$$\bigcup_{i=1}^{O} \mathcal{S}_i = \{(x, y): \ 1 \leq x \leq 6, \ 1 \leq y \leq 8\} \quad ; \quad \mathcal{S}_i \cap \mathcal{S}_j = \emptyset, \ \forall i \neq j \quad ; \quad \mathcal{S}_i \neq \emptyset, \ \forall i,$$

and satisfies

$$\forall i \leq O, \ \forall (x, y) \in \mathcal{S}_i, \ R_{xy} = o_i.$$

For each injective mapping $\phi : \{1, \dots, O\} \to \{1, \dots, 30\}$, we can build a new room $\tilde{R}$ with (a) observations $\phi(1), \dots, \phi(O)$ and which (b) *preserves the room partition*, by assigning all the 2D spatial positions in $\mathcal{S}_i$ to the observation $\phi(i)$. That is

$$\forall i \leq O, \ \forall (x, y) \in \mathcal{S}_i, \ \tilde{R}_{xy} = \phi(i).$$

The training sets in Sec.4.5 contain at most 10k such training rooms, each one preserving the room partition. Note that, for a given partition, the number of possible rooms which preserves the room partition is the number of injective mapping from $\{1, \dots, O\}$ to $\{1, \dots, 30\}$, which is $\prod_{i=0}^{O-1} 30 - i$.

Learning in base: As discussed in the main text, given a sequence of inputs observations $x = (x_1, \dots, x_N)$ in a room defined as above, each model is trained to predict targets in the *room-agnostic base*, $\tilde{x}_n \in \{1, \dots, O\}$, such that $\tilde{x}_n = k$ iff. $\tilde{x}_n$ is the kth lowest observation seen between indices 1 and n.

In particular, let us define the permutation $i_1, \dots, i_O$ of $1, \dots, O$ such that $\phi(i_1) \leq \dots \leq \phi(i_O)$. Let n^* be such a large enough integer such that all the O different room observations have been seen between x_1 and x_{n^*}. Then, for $n \geq n^*$, the target $\tilde{x}_n = k$ corresponds to the kth highest color of the room, which is equal to $\phi(i_k)$.

L.2. Table of results

Table 10 reports the numerical results displayed in Fig. 5[left].

First, we observe that for each model, in-context capacities emerge when we increase the number of rooms.

Second, while vanilla LSTM performs best for a small number of training rooms, it cannot reach near-perfect in-context accuracies when the number of training rooms increases—which a vanilla transformer can do. We believe that attention-like mechanisms are important for in-context capacities to fully emerge in novel test rooms.

Third, as observed in other experiments, vanilla sequence models cannot solve path planning problems. Indeed, for a large number of training rooms, a transformer improves at best 77% of the times over the fallback path. However, when it does so, it finds paths that are on average 25 times longer than the optimal paths. In contrast, our best $\texttt{TDB}(S = 3, M = 4)$ (a) almost matches the nearly perfect in-context predictive performance of a vanilla transformer, (b) almost perfectly solves in-context path planning problems and (c) learns in-context latent graphs nearly isomorphic to the ground truth.

Finally, our TDB$(S = 3, \text{enc}, M = 4)$—which also predicts the next latent encoding—performs worse than the other models. This is explained by its large variances across the different experiments. In fact, we observe that for two out of ten experiments, its training accuracy remains very low—below 40%—as the model struggles to learn to predict the next observation. For the remaining eight runs, prediction and path planning performance are near optimal and the model competes with our best $\texttt{TDB}(S = 3, M = 4)$. We could try increasing the number of bottlenecks and training a TDB$(S = 3, \text{enc}, M = 16)$ to mitigate this issue.

Metric	Num. training rooms	LSTM	Transformer	TDB($S=3, M=4$)	TDB($S=1, \text{enc}, M=4$)
TestAccu (%) ↑	200	**0.508** (0.009)	0.426 (0.005)	0.38 (0.004)	0.379 (0.002)
	500	**0.741** (0.039)	0.595 (0.006)	0.461 (0.009)	0.481 (0.007)
	1000	**0.899** (0.006)	0.864 (0.016)	0.568 (0.015)	0.58 (0.03)
	2000	0.919 (0.009)	**0.977** (0.003)	0.868 (0.03)	0.8 (0.056)
	5000	0.848 (0.045)	**0.985** (0.001)	0.974 (0.001)	0.784 (0.091)
	10000	0.924 (0.007)	**0.985** (0.001)	0.977 (0.001)	0.912 (0.059)
ImpFallback (%) ↑	200	0.005 (0.001)	0.002 (0.001)	**0.038** (0.005)	0.032 (0.004)
	500	**0.148** (0.04)	0.007 (0.002)	0.05 (0.004)	0.064 (0.004)
	1000	**0.382** (0.054)	0.189 (0.032)	0.115 (0.014)	0.124 (0.016)
	2000	0.447 (0.059)	0.725 (0.024)	**0.690** (0.067)	0.495 (0.092)
	5000	0.312 (0.086)	0.809 (0.009)	**0.964** (0.011)	0.605 (0.132)
	10000	0.305 (0.072)	0.771 (0.011)	**0.970** (0.009)	0.782 (0.084)
RatioSP ↓	200	51.79 (20.83)	34.79 (11.07)	1.46 (0.09)	**1.33** (0.06)
	500	29.73 (3.67)	45.26 (23.87)	**1.17** (0.05)	1.27 (0.06)
	1000	25.52 (1.19)	26.14 (1.34)	**1.18** (0.04)	1.20 (0.04)
	2000	27.01 (0.89)	25.98 (0.83)	**1.07** (0.03)	1.11 (0.04)
	5000	24.79 (0.96)	24.53 (0.63)	**1.01** (0.0)	1.06 (0.05)
	10000	24.27 (1.73)	25.13 (0.45)	**1.01** (0.0)	**1.01** (0.0)
NormGED ↓	200	—	—	**0.707** (0.005)	0.723 (0.006)
	500	—	—	**0.658** (0.006)	0.660 (0.008)
	1000	—	—	**0.579** (0.018)	0.602 (0.033)
	2000	—	—	**0.164** (0.042)	0.29 (0.073)
	5000	—	—	**0.019** (0.003)	0.294 (0.121)
	10000	—	—	**0.017** (0.002)	0.118 (0.079)

Table 10. In-context metrics for vanilla sequence models and TDBs on the new test rooms. For a large number of training rooms, our best TDB($S=3, M=4$) (a) reaches nearly perfect in-context accuracy, close to the best performing transformer, (b) almost perfectly solves the in-context path planning problems, and (c) learns in-context latent graphs nearly isomorphic to the ground truth.

L.3. In-context accuracy by timestep

Fig.13 shows the average in-context accuracy on the new test rooms, as a function of the observation index, for our best TDB($S=3, M=4$) trained on 5k rooms. Interestingly, in-context accuracy drops after a few iterations. Indeed, as the model sees new observations in the new test rooms, the mapping between the base labels $\tilde{x}_n$ and the observations x_n may change—which may confuse the model. After a few tenths of iterations, TDB is able to locate itself in the new test room. After 70 iterations, in-context accuracy becomes higher than 99%.

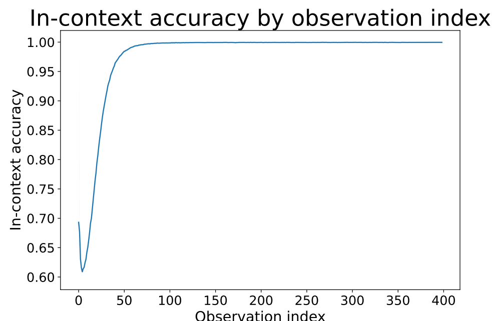

Figure 13. After 70 timesteps, in-context accuracy for a TDB($S=3, M=4$) trained on 5k training rooms becomes higher than 99%.

L.4. In-context learning is driven by spatial exposure to base targets in the training data

Restricting training rooms: For this experiment, we build training rooms such that the kth highest color (almost) never appears at any of the spatial positions indicated by the set $\mathcal{S}_k$ of the room partition. With the notations of Appendix L.1, each injective mapping ϕ used to generate a training room has to satisfy the rule

$$\mathcal{R} = \{\forall k \leq O: \; i_k \neq k\}.$$

For instance, the mapping $\phi(1) = 5, \; \phi(2) = 13, \; \phi(3) = 20, \; \phi(4) = 8$ cannot be used to generate a training room. Indeed, when we rank the entries we get $i_1 = 1, \; i_2 = 4, \; i_3 = 2, \; i_4 = 3$ and $i_1 = 1$ violates $\mathcal{R}$. Similarly $\phi(1) = 20, \; \phi(2) =$

$2,\ \phi(3) = 15,\ \phi(4) = 11$ violates $\mathcal{R}$ as $i_3 = 3$.

Given a training sequence x, let n^* be such a large enough integer such that all the O different room observations have been seen between x_1 and x_{n^*}. For $n \geq n^*$, let $k \leq O$ be such that the spatial position pos_n of the agent at timestep n belongs to the set $\mathcal{S}_k$ of the room partition. Then, by definition, the base target $\tilde{x}_n$ satisfies $\tilde{x}_n \neq k$. Note that, by definition, when $n \leq n^*$ and we have not yet observed all the O different room observations, we may have $\tilde{x}_n = k$ when $\mathrm{pos}_n \in \mathcal{S}_k$.

Consequently, for each room spatial position, there is a position-specific base target that the agent will never see (when $n \geq n^*$) at this position on the training set.

We generate 5k such training rooms and train a $\texttt{TDB}(S = 3, M = 4)$ as before.

In-context accuracies: We consider four families of test rooms:
(A): Test rooms where $\mathcal{R}$ is respected.
(B): Test rooms where $\mathcal{R}$ is violated once.
(C): Test rooms where $\mathcal{R}$ is violated twice.
(D): Test rooms where $\mathcal{R}$ is violated four times. These are the rooms satisfying $\phi(0) \leq \phi(1) \leq \phi(2) \leq \phi(3)$.

On test rooms **(A)**, in-context accuracy is $98.05\%(\pm 0.07\%)$.
On test rooms **(B)**, in-context accuracy is $56.97\%(\pm 0.82\%)$.
On test rooms **(C)**, in-context accuracy is $50.25\%(\pm 0.51\%)$.
On test rooms **(D)**, in-context accuracy is $46.20\%(\pm 0.73\%)$.

This drastic drop in in-context accuracy demonstrates that—in addition to the number of training rooms—in-context capacities are also driven by spatial exposure to base targets during training.